Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

SparseX: Βιβλιοθήκη για τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα σε πολυπύρηνες αρχιτεκτονικές

Διπλωματική εργασία

Αθηνά Ελαφρού

Αθήνα,
Ιανουάριος, 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

SparseX: Βιβλιοθήκη για τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα σε πολυπύρηνες αρχιτεκτονικές

Διπλωματική εργασία

Αθηνά Ελαφρού

Επιβλέπων καθηγητής: Νεκτάριος Κοζύρης

Εγκρίθηκε από την τριμελή επιτροπή την 31η Ιανουαρίου 2014.

Νεκτάριος Κοζύρης Καθηγητής, Ε.Μ.Π.	Δημήτριος Σούντρης Επίκ. Καθηγητής, Ε.Μ.Π.	Δημήτριος Τσουμάκος Επίκ. Καθηγητής, Ιόνιο Παν.

Αθήνα,
Ιανουάριος, 2014

Αθηνά Ελαφρού
Διπλωματούχος Εθνικού Μετσοβίου Πολυτεχνείου

Περίληψη

Η διπλωματική αυτή εργασία εστιάζει στην υλοποίηση μίας βιβλιοθήκης ανοιχτού λογισμικού για τον πολλαπλασιασμό αραιού πίνακα με διάνυσμα (SpMV) σε σύγχρονες πολυπύρηνες αρχιτεκτονικές υπολογιστών, με χρήση της δομής αποθήκευσης αραιών πινάκων Compressed Sparse eXtended (CSX). Προηγούμενη έρευνα έχει δείξει ότι η δομή CSX μπορεί να παράσχει σημαντική βελτίωση της επίδοσης του πυρήνα SpMV σε μία πληθώρα διαφορετικών πινάκων και πολυπύρηνων αρχιτεκτονικών, διατηρώντας μία σημαντική σταθερότητα στην επίδοση. Η μοναδική του αυτή ικανότητα το καθιστά εξαιρετικό υποψήφιο για HPC εφαρμογές και, συνεπώς, θεωρούμε ότι η ενσωμάτωσή του στην ιεραρχία λογισμικού επίλυσης υπολογιστικών προβλημάτων θα έχει σημαντική επίδραση σε ένα μεγάλο εύρος επιστημονικών εφαρμογών. Με αυτό το σκοπό, παρουσιάζουμε και αξιολογούμε την βιβλιοθήκη SparseX, μια συλλογή από ρουτίνες για χρήση σε βιβλιοθήκες επίλυσης γραμμικών συστημάτων αλλά και ανεξάρτητες εφαρμογές.

Λέξεις κλειδιά: high performance computing; sparse matrix-vector multiplication; multicore; SpMV; SpMV library; CSX; HPC

Abstract

This thesis focuses on the implementation of an open-source library for performing the Sparse Matrix-by-Vector Multiplication (SpMV) kernel on modern multicore architectures, using the Compressed Sparse eXtended (CSX) format for storing sparse matrices. Previous research has demonstrated that CSX achieves significant improvements in the performance of this kernel on a wide variety of matrices and multicore architectures, by drastically reducing the memory footprint of the sparse matrix. Its unique performance stability makes it an excellent candidate for HPC applications and, thus, we believe CSX's integration in the numerical software stack will have an important impact on many scientific applications, that are sensitive to the performance of the SpMV kernel. To this end, we present and evaluate SparseX, a collection of low-level primitives for use by solver libraries and standalone applications.

Keywords: high performance computing; sparse matrix-vector multiplication; multicore; SpMV; SpMV library; CSX; HPC

Contents

List of Figures

List of Tables

List of Algorithms

1

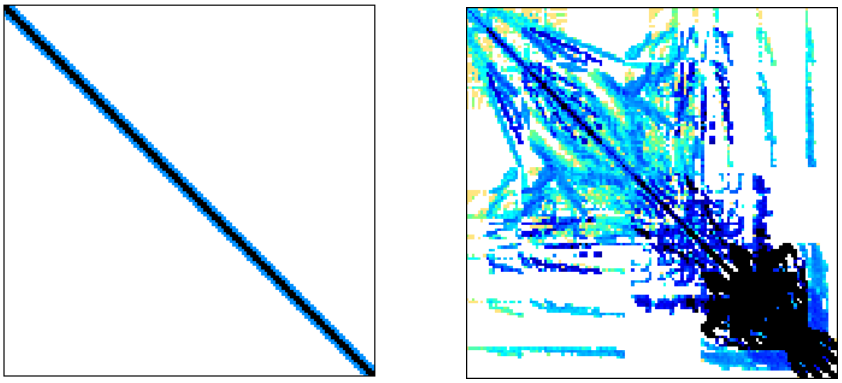
Introduction

Research in sparse linear systems has been active since the early days of computing systems, when a first realization was made that one can take advantage of "sparsity" to design solution methods that can be quite economical. This triggered the development of a wide variety of methods, both direct and iterative, over the last decades, whose performance is vital, nowadays, to a wide variety of HPC applications. Although the grand challenges of these applications are diverse, they share an intersection in terms of sparse linear system solvers, spending, in many cases, most of the execution time on solving such systems. In this chapter, we briefly present the different methods for solving large sparse linear systems and designate the importance of the Sparse Matrix-Vector Multiplication kernel, which is the focus of this diploma thesis.

1.1 Sparse matrices

A sparse matrix is defined as a matrix populated primarily by zeros. The sparsity of a matrix is defined as the number of zero elements divided by the total number of elements in the full matrix. For example, a sparsity value of 90% means that 90% of the elements in the matrix are zeros. In many cases the sparsity value is a little over 99%.

Sparse matrices arise in scientific applications that use finite difference, finite element or finite volume discretizations of partial differential equations (PDEs) in problems with underlying 2D or 3D geometry coming from computational fluid dynamics, electromagnetics, thermodynamics, materials, semiconductor devices, model reduction and structural mechanics, but also in other application domains where problems do not seem to have such a geometry, as in electrical circuit simulation, chemical process simulation, economic and financial modeling, power networks and graphs. For example, in circuit simulation, the elements of a sparse matrix may represent circuit voltages, currents, impedances and power sources; in model reduction the



(a) *boneS10* **(b)** *offshore*

Figure 1.1: A structured sparse matrix on the left and an unstructured sparse matrix on the right from the University of Florida Sparse Matrix Collection [Davis and Hu, 2011]. In (a), *boneS10* is part of a second order system that results from applying a three-dimensional serial reconstruction technique to model the porous bone micro-architecture and in (b), *offshore* is a finite-element system matrix derived from the 3D transient electric field diffusion equation. The matrix results from the discretization of offshore conductivity structures using tetrahedral elements.

elements of a sparse matrix may represent the porous micro-architecture of a human bone, after applying a finite element method; in web-related graph problems the elements of the matrix may represent the existence of a hyperlink between two webpages.

There are two broad types of sparse matrices: structured and unstructured [Saad, 2003]. A structured matrix is one whose nonzero entries form some sort of regular pattern, often along a small number of diagonals or block diagonals. When the nonzero entries are confined exclusively to a diagonal band, comprising the main diagonal and zero or more diagonals on either side, the matrix is called a band or banded matrix. On the other hand, unstructured matrices comprise of elements which are irregularly located. Figure 1.1 shows a matrix of each category. This distinction between the two types of matrices is important to the performance of matrix-by-vector products, which are the focus of this diploma thesis. As will be shown later on, their performance may significantly differ, depending on whether the matrix is structured or not.

Even within the same category, matrices may vaslty differ in terms of their structure. For example, matrices arising in circuit simulation (and other network-related domains) differ greatly from matrices arising from the discretization of 2D and 3D physical domains. Computational fluid dynamics

matrices differ from structural engineering matrices, and both are vastly different from matrices arising in linear programming or financial portfolio optimization. A sample set of sparse matrices from The University of Florida Sparse Matrix Collection [Davis and Hu, 2011], that highlights the complexity and diversity of matrices that arise in real-life applications is available in Appendix A.

1.2 Sparse linear systems

Sparse matrices are usually involved in the solution of large linear systems of the form

$$A \cdot x = b$$

where A denotes the coefficient matrix, b is the right-hand side vector, and x is the unknown vector. There are two basic classes of methods for solving linear systems of the above form.

The first class is represented by direct methods, which theoretically yield an exact solution in a (predictable) finite number of steps. In practice, of course, the solution obtained will be contaminated by the round-off error that is involved with the arithmetic being used. Such methods include the Gaussian Elimination Method, the LU Decomposition Method and the Cholesky Method [Duff et al., 1989; Barrett et al., 1987; Davis, 2006]. These methods rely on the factorization of the coefficient matrix as a product of three matrices $A = L \cdot D \cdot U$, where L and U are lower and upper triangular respectively and D is diagonal. Then, the solution of the system comes down to solving two easily invertible triangular systems $L \cdot y = b$ and $U \cdot x = D^{-1} \cdot y$. Forward elimination followed by backward substitutions complete the solution process.

Direct methods are important because of their generality and robustness. Indeed, for "tough" linear systems arising in some applications (e.g., circuit simulation) they are the only feasible solution. Furthermore, they provide an effective means of solving multiple systems with the same coefficient matrix A but different right-hand side vectors, because the factorization only needs to be performed once. On the downside, the asymptotic time complexity of all dense direct methods is $O(n^3)$ for the factorization and $O(n^2)$ for solving the system based on the precomputed factorization. When the dimension of the coefficient matrix is in the order on 10^5, 10^6 or more, the total cubic complexity is prohibitive. Furthermore, the elimination process of these methods may introduce fill-in, i.e., matrices L and U may have nonzero elements in locations where the original matrix A has zeros, complicating the solution of sparse systems as well; by necessitating a fill-in minimization step during the

factorization process in order to increase the sparsity of matrices L and U and by introducing the need for a storage format that supports the insertion of new elements [Saad, 2003].

The second class of methods for solving linear systems includes iterative techniques that try to find an approximate solution. There are two broad categories of iterative methods: stationary methods and projection methods.

Stationary methods begin with a given approximate solution of the system and iteratively modify the components of the approximation, one or a few at a time until convergence is reached. These modifications, called relaxation steps, usually aim to annihilate some component(s) of the residual vector $b - A \cdot x$. The most common stationary methods are the Jacobi, Gauss-Seidel and the Successive Overrelaxation (SOR) methods. The convergence of these methods cannot be guaranteed for all matrices and therefore they are rarely used separately [Saad, 2003].

Projection methods try to extract an approximate solution x of a linear system from a subspace of $\mathbb{R}^n$. If $\mathcal{K}_m$ is this subspace of candidate approximants where m denotes its dimension, then, in general, m constraints must be imposed on the residual vector $b - A \cdot x$ to be able to extract such an approximation. More specifically, the residual vector must be orthogonal to m linear independent vectors, which form the subspace of constraints L_m. At each step of a projection method a new pair of $\mathcal{K}_m$ and $\mathcal{L}_m$ subspaces is used with an initial guess x_o equal to the approximation obtained from the previous step.

Some of the most important iterative techniques for solving large linear systems are the projection methods known as the Krylov subspace methods. A Krylov subspace method is a method for which the subspace K_m is the Krylov subspace:

$$Km(A, r_0) = span\{r_0, A \cdot r_0, A^2 \cdot r_0, ..., A^{m-1} \cdot r_0\}$$

where $r_0 = b - A \cdot x_0$, is the residual of the initial guess x_o. Different versions of Krylov subspace methods arise from different choices of the subspace $\mathcal{L}_m$ and from different preconditioning techniques applied to the original system. Preconditioning involves applying a transformation, called the preconditioner (which is usually a direct solution method), on a system in order to make it more suitable for numerical solution by improving its convergence characteristics. It is the key ingredient to the success of Krylov subspace methods when applied on systems derived from large "real-world" problems, whose convergence ratio would otherwise be very low.

Preconditioned Krylov subspace methods, such as the Generalized Minimum Residual (GMRES) method [Saad and Schultz, 1986] and the Conjugate Gradient (CG) method [Hestenes and Stiefel, 1952], are widely used nowadays for solving large sparse linear systems. From a computational point

Method	Vector updates	Dot products	Matrix-by-Vector products
GMRES	i+1	i+1	1
CG	3	2	1

Table 1.1: Summary of operations for iteration i of the GMRES method and the CG method.

of view, and setting aside the preconditioning process, these methods rely mainly on the following computational kernels:

Vector Updates. Operations of the form

$$y = y + a \cdot x$$

where a is a scalar and y and x are vectors, are known as vector updates or SAXPY operations (Scalar Alpha X Plus Y according to the naming conventions of the Basic Linear Algebra Subprograms package, or simply BLAS [Lawson et al., 1979]).

Dot Products. The dot product is the inner product of two vectors x and y:

$$t = x^T \cdot y$$

Matrix-by-Vector Products. This is the product of a sparse matrix A (the coefficient matrix of the system) and a vector x:

$$y = A \cdot x$$

Table 1.1 gives a summary of the above operations for each iteration of the GMRES and the CG methods.

In order to obtain good performance of the iterative method, one must efficiently implement the above computational kernels on the target architecture. When it comes to multicore architectures, which are of interest in this thesis, the implementation of a vector update is pretty straightforward, since each processor may execute the operation on a specific range of the vectors and all processors may work independently. A dot product, on the other hand, is more complicated to implement, since the reduction step required to calculate the final result has limited parallelism that may hinder performance. Finally, a matrix-by-vector product, which we will henceforth call a Sparse Matrix-Vector Multiplication kernel (SpMV), is probably the most intriguing operation involved. The performance of this kernel is intimately associated with the data structures used to store the sparse matrix. It seems to be the most expensive operation involved in iterative solvers, as shown in Figure 1.2, and, therefore, an efficient implementation is essential to their acceleration.

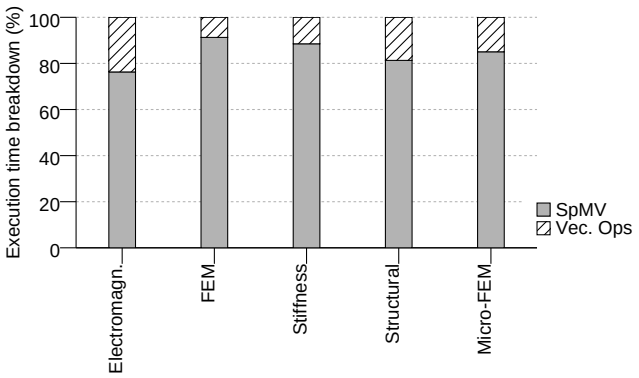


Figure 1.2: Execution time breakdown for the non-preconditioned CG iterative method for different problem categories [Karakasis, 2012].

1.3 The SpMV kernel

The Sparse Matrix-Vector Multiplication kernel (SpMV) is considered to be one of the most important computational kernels in science and engineering. Since 1970s, plenty of researches have been dedicated to optimizing SpMV performance for its fundamental importance. However, conventional implementations have historically been reported to perform poorly on modern microprocessors (e.g., 10% of peak performance [Vuduc and Moon, 2005]) due to a number of issues concerning the algorithm itself, the storage formats, and the sparsity patterns of the matrices. Each of the following sections will address one of these issues.

1.3.1 The algorithmic nature of the kernel

A high level approach of the SpMV kernel is given in Algorithm 1.1. In total, the algorithm performs $O(n^2)$ operations on $O(n^2)$ amount of data, which means that the ratio of floating point operations to memory accesses (flop:byte ratio, also known as arithmetic intensity [Harris, 2005; Williams et al., 2009]) is $O(1)$. Figure 1.3 shows the arithmetic intensity of a number of HPC kernels. Some of them have an arithmetic intensity that scales with the problem size (FFTs, Dense Linear Algebra, Naive Particle Methods) while others have a constant arithmetic intensity, including the SpMV kernel. A constant flop:byte ratio designates a lack of temporal locality, which subsequently suggests that if the memory subsystem cannot provide data to the CPU in a comparable speed, then for large working sets (that do not fit in the system's cache hierarchy) the kernel will be memory bound. Even in a multithreaded environment, where the algorithm exhibits ample paral-

lelism [Buluc et al., 2011], due in particular to the lack of data dependencies between different rows, the kernel doesn't scale over a specific number of threads depending on the system's available memory bandwidth.

1: **procedure** SPMV(A::in, x::in, y::out)
   A: matrix
   x: input vector
   y: output vector
2: **foreach** row_i **in** A **do**
3: $y_i \leftarrow {row_i}^T \cdot x$
4: **end for**

Algorithm 1.1: A high level approach of the SpMV kernel.

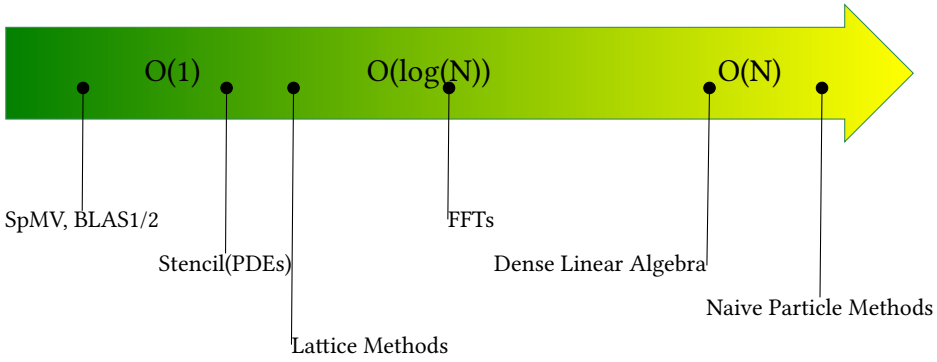


Figure 1.3: Arithmetic intensity of some representatative HPC kernels.

1.3.2 Conventional storage formats

When multiplying a large sparse matrix A by a dense vector x, the memory bandwidth for reading A can limit overall performance [Goumas et al., 2009]. Consequently, most algorithms to compute $A \cdot x$ store A in a compressed format.

The simplest storage format that has been proposed is the so called *Coordinate (COO)* format [Pooch and Nieder, 1973; Tewarson, 1973; Duff and Reid, 1979; Saad, 1992]. It uses one floating-point array `values[nnz]` and two integer arrays `rowind[nnz]` and `colind[nnz]`, where `nnz` is the number of nonzero elements in the matrix. The first array stores the values of the nonzero elements, while the `rowind` and `colind` arrays store the row

and column indices respectively of each element in the `values` array. Figure 1.4 shows a typical implementation of the COO format, while Algorithm 1.2 shows an implementation of the SpMV kernel using this format.

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$

```
rowind: ( 0  0  0  0  1  1  1  2  2  2  3  3  4  4  5  5 )
colind: ( 0  1  2  3  0  1  2  0  1  2  0  3  4  5  4  5 )
  val:  ( 7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1 )
```

Figure 1.4: The Coordinate (COO) sparse matrix storage format.

1: **procedure** SpmvCoo(A::in, x::in, y::out)
   A: matrix in COO format
   x: input vector
   y: output vector
2: **for** $i \leftarrow 0$ **to** NNZ **do**
3: $temp_i \leftarrow rowind[i]$
4: $y[temp_i] \leftarrow y[temp_i] + values[i] \cdot x[colind[i]]$
5: **end for**

Algorithm 1.2: SpMV implementation using the COO format.

The use of this format in the SpMV kernel yields a low performance due to an unsatisfyingly large memory footprint. Assuming 4-byte integers for indexing and 8-byte double precision floating point values, a typical layout in iterative solvers, the COO format sums up to $4nnz + 4nnz + 8nnz = 16nnz$ bytes. The corresponding flop:byte ratio of the SpMV kernel in this case can be calculated as

$$\frac{2nnz}{16nnz + 16n} = \frac{1}{8 + 8\frac{n}{nnz}} \tag{1.1}$$

that leads to an arithmetic intensity of 0.125 when $nnz \gg n$, which is the case in most sparse matrices.

A more compact format that is widely used in scientific computing is the *Compressed Sparse Row (CSR)* format [Tinney and Walker, 1967; Pooch and

Nieder, 1973; Duff and Reid, 1979; Saad, 1992]. This format maintains the `values` and `colind` arrays of the COO format but replaces the `rowind` array with a new integer array `rowptr[n+1]`, where n is the number of rows. Instead of explicity storing the row index of each nonzero element, this format further reduces the indexing information by using pointers to the beginning of each row, i.e., `rowptr[i]` contains the index of the first element of row i inside the `values` and `colind` arrays. The last entry points to the end of the `values` and `colind` arrays. Figure 1.5 gives an example of storing a sparse matrix in the CSR format, while Algorithm 1.3 shows an implementation of the SpMV kernel using this format.

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$

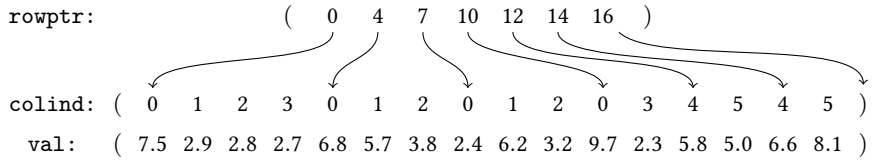


Figure 1.5: The Compressed Sparse Row sparse matrix storage format.

```
1: procedure SpmvCsr(A::in, x::in, y::out)
   A: matrix in CSR format
   x: input vector
   y: output vector
2:     for i ← 0 to N do
3:         for j ← rowptr[i] to rowptr[i + 1] do
4:             y[i] ← y[i] + values[j] · x[colind[j]]
5:         end for
6:     end for
```

Algorithm 1.3: SpMV implementation using the CSR format.

Assuming, again, 4-byte integers for indexing and 8-byte double precision floating point values, the CSR format sums up to $4(n+1)+4nnz+8nnz = 4n + 12nnz + 4$ bytes. In the case of most sparse matrices $nnz \gg n$, which

leads to a $12nnz$ footprint, a major improvement over the COO format. The corresponding flop:byte ratio of the kernel can be calculated as

$$\frac{2nnz}{12nnz + 16n} = \frac{1}{6 + 8\frac{n}{nnz}} \tag{1.2}$$

which approximately equals to 0.167. This may surpass the arithmetic intesity of the kernel implemented with the COO format, but the fact remains that the flop:byte ratio is very low compared to other computational kernels (again see Figure 1.3).

Using a more compact storage format, like the CSR format described previously, creates a number of additional performance issues. The indexing structures (`rowptr` and `colind` in the case of CSR) introduce indirect memory references that increase the number of load operations, creating additional cache interference [Pinar and Heath, 1999]. Furthermore, access to the right-hand-side vector x is no longer sequential and depends on the irregularity of the matrix structure, limiting spatial locality. Therefore, even though the memory footprint of the matrix is reduced and, thus, the total memory access time, the computational load of the processor seems to be increased. This, however, does not pose a problem in a modern multicore system, where the memory bandwidth bottleneck allows a number of time-consuming operations to take place without being exposed to the total execution time of the kernel.

1.3.3 The effect of the sparsity pattern

Many sparse matrices contain a large number of rows with a short length, leading to a small average rowsize, $\frac{nnz}{n}$. According to equations (1.1) and (1.2) the flop:byte ratio of the kernel is proportional to the average rowsize, which means that the smaller the average rowsize the lower the flop:byte ratio. From a more practical point of view, by inspecting Algorithm 1.3 one can observe that a small average rowsize means that the inner loop runs over only a few iterations. This small trip count may introduce a significant loop overhead [White and Sadayappan, 1997]. Furthermore, since the rows of the matrix usually have variable lengths, the inner loop is almost impossible to unroll as is, forcing the potential loop overhead to be performed for every short row, further degrading performance (this problem can actually be overcome by employing a register blocking technique; see Section 1.4 for an example).

1.4 Alternative storage formats

Even though the CSR format is the most popular format for storing sparse matrices, it keeps a lot of redundant information. Usually, sparse matrices arising in the solution of large linear systems consist of small dense substructures, e.g., horizontal,vertical or diagonal sequences, 2-D blocks etc. Therefore, one could take advantage of these dense substructures to further reduce the matrix size, by using one column index for each one that is encountered in the matrix. Many storage formats that focus on the minimization of the indexing structures have been proposed in the past. For example, the *Blocked Compressed Sparse Row (BCSR)* format [Im and Yelick, 2001] stores the matrix as a sequence of fixed-size $r \times c$ dense blocks. It uses one column index per block, reducing the indexing storage by a factor of $\frac{1}{rc}$. Fixed-size blocks favor loops optimizations, such as loop unrolling, and enable register-level tiling. However, this format introduces zero-padding in order to create these fixed-size blocks and, thus, its effectiveness depends on the structure of the matrix. Other formats exploit different types of substructures, e.g., the *Variable Block Length (VBL)* format detects 1-D variable sized horizontal blocks, the *Variable Block Row (VBR)* format exploits variable sized blocks, the *Blocked Compressed Sparse Diagonal (BCSD)* format uses fixed-size diagonal blocks etc [Pinar and Heath, 1999; Saad, 1994; Agarwal et al., 1992]. The problem with the aforementioned formats is that each of them accelerates the SpMV kernel over CSR for matrices in which the detected substructure holds a sufficient number of instances, while exhibiting a significant performance degradation otherwise.

The *Compressed Sparse eXtended (CSX)* format has been proposed as an alternative storage format that offers a stable high performance across most types of matrices and different platforms (including symmetric shared memory and NUMA multicore architectures) [Kourtis et al., 2011; Karakasis et al., 2013], by further reducing the memory footprint of the matrix with a series of compression techniques that will be explained in detail in the following chapter. The implementation of a functional library interface that will allow CSX to be easily exploited by the HPC community in the context of solving sparse linear systems forms the contribution of this diploma thesis. Thus, we introduce the SparseX library, an API written in the C programming language that provides the means to developers of solver libraries and of scientific and engineering applications to easily attain better performance in modern multicore architectures than the one exhibited by conventional storage formats. In order to demonstrate SparseX's advantage, it is compared to other popular libraries providing the same functionality, including Intel® Math Kernel Library (Intel® MKL) [Intel® Coorporation, 2013a], a well established commercial product, and the parallel Optimized Sparse Ker-

nel Interface (pOSKI) library [Vuduc et al., 2005; Ankit, 2008], which has been developed by the Berkeley Benchmarking and Optimization (BeBOP) group (http://bebop.cs.berkeley.edu/).

1.5 Outline

The remainder of this diploma thesis is organised as follows:

Chapter 2 introduces in detail the Compressed Sparse eXtended (CSX) format, which has been proposed as an alternative sparse matrix storage format that improves the performance of the SpMV kernel on modern multicore architectures.

Chapter 3 follows with a discussion on the design goals and decisions concerning the SparseX library, accompanied by an overview of the provided functionality along with some use cases.

Chapter 4 continues with a performance evaluation of different aspects of SparseX and a thorough comparison to other popular libraries that provide similar functionality.

Chapter 5 concludes this diploma thesis by summarizing its contribution and giving some insight on future directions.

2

The Compressed Sparse eXtended Format

As pointed out earlier, the key to the optimization of the SpMV kernel is the minimization of the memory footprint for storing the sparse matrix. The most successful format towards this direction is the *Compressed Sparse eXtended (CSX)* format [Kourtis et al., 2011]. The distinguishing feature of this format is that it encodes the matrix into a multitude of substructure types, unlike other CSR alternatives, which usually focus on a single type of substructure. This feature, in conjunction with a number of aggressive compression techniques, allows CSX to accomplish a significant minimization of the column index information, reaching in many cases the maximum possible compression ratio [Karakasis et al., 2013].

2.1 The CSX data structures

The CSX format uses a single data structure to store all the required indexing information. This structure is based on a variant of the CSR format, called *CSR with Delta Units (CSR-DU)* [Kourtis et al., 2008b], which effectively compresses the `colind` array by applying delta indexing on the column indices, thus minimizing the storage needs of each element. In order to fully understand the CSX indexing scheme, one must first explore the CSR-DU format.

2.1.1 The CSR with Delta Units (CSR-DU) storage format

CSR-DU divides the matrix into areas, called *delta units*. Each delta unit comprises of a variable number of (not necessarily contiguous) non-zero elements, which form an horizontal regularity. The detection of these units is based on the following delta-encoding scheme: first, we run through every non-zero element of the matrix and calculate its *delta value*, i.e., the column distance from the previous non-zero element in the same row. When all the delta values have been calculated, the elements of each row are grouped in sequences according to the number of bits needed to store their delta value

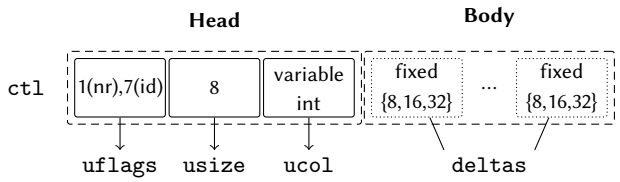


Figure 2.1: The `ctl` structure employed by CSR-DU. Optional fields are denoted with a dotted bounding box.

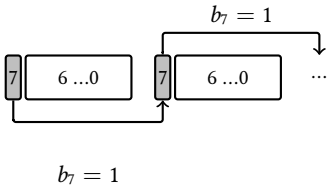


Figure 2.2: The variable-length integer encoding employed by both CSR-DU and CSX. Only the seven lower bits are used for storing an integer; numbers larger 127 are stored in 7-bit chunks with bit 8 serving as a continuation marker.

(8, 16, 32, 64). Each storage size corresponds to a different unit type (*d8*, *d16*, *d32*, *d64*). For example, consecutive elements whose delta value is less than $2^8 = 256$ need 8 bits of storage and are therefore stored in a group of type *d8*. The indexing information of each unit is stored in a single byte-array, called `ctl`, while the corresponding values are stored in the `values` array, as usual. Thus, the `colind` array of CSR is replaced by the `ctl` array.

In the implementation of CSR-DU presented in [Kourtis et al., 2011] each unit is represented by a *header* and a *body* (see Figure 2.1). The header includes unit information in the form of two 1-byte fields, called `usize` and `uflags`, and a variable-length integer, called `ucol`. A variable-length integer is essentially an integer stored in the minimum number of 7-bit chunks, reserving the most significant bit of each chunk as a link to the next one, as illustrated in Figure 2.2. The `usize` field represents the number of elements in the unit, while the `uflags` field stores the unit type (d8, d16, d32 or d64) and a 1-bit flag that indicates whether the unit is at the beginning of a new row. The `ucol` field stores the distance of the unit's column index from the previous one. The body stores the delta values of the `usize`-1 last elements of the group.

2.1.2 The CSX extension

The CSX format enriches the notion of units employed by CSR-DU, in order to enable the encoding of other regularities that are usually encountered

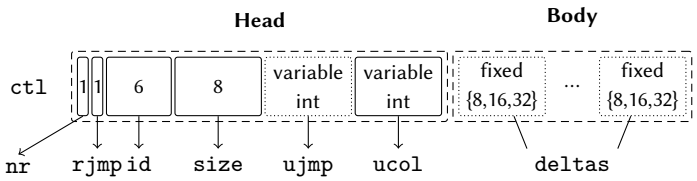


Figure 2.3: The CSX extension of the `ctl` structure. Optional fields are denoted with a dotted bounding box.

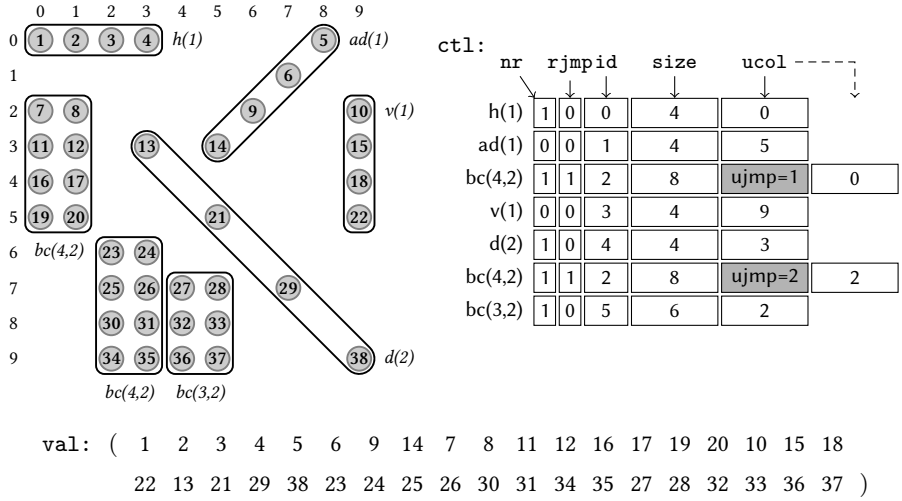


Figure 2.4: Example of the CSX format.

in sparse matrices. In CSX's terminology a unit can also be a *substructure unit*, representing one of the multiple substructure types detected, including horizontal, vertical, diagonal, anti-diagonal and two-dimensional blocks, which will be described in detail in the following section. Figure 2.3 shows the extended version of the `ctl` structure employed by CSX. Since CSX also encodes non-horizontal substructures, it is possible for substructure units to group together all the elements of subsequent rows, leaving them empty. In order to be able to skip the empty rows when executing the multiplication, CSX introduces the `rjmp` bit to indicate the existence of empty rows and the `ujmp` field to store the number of empty rows that follow. The latter is present only when the `rjmp` bit is set.

As for the values of the non-zero elements, CSX keeps the `values` array of the CSR format, but changes the order in which the elements are stored. Since the matrix is encoded in units, the ordering of the values must agree with the unit ordering. Within a single unit the values are stored in a substructure-dependent order, as illustrated in Figure 2.4.

col. indices:	**1**	**10**	**11**	**12**	**13**	**14**	**21**	**41**	**61**	**81**	...
delta values:	1	9	1	1	1	1	7	20	20	20	...
			$d = 1$					$d = 20$			

Figure 2.5: Example of the run-length encoding process.

2.2 Detection and encoding of substructures

CSX encodes a wide variety of substructure types with the indexing scheme described in the previous section. To facilitate the process of detecting and encoding different substructures, CSX uses a special internal representation, which is a hybrid of the COO and CSR formats. More specifically, CSX's internal representation stores *generic elements*; a generic element is either a single non-zero element or a substructure, stored as a (i, j, v) tuple. In case of a single element, (i, j) represent its coordinates and v its value, while in case of a substructure, (i, j) correspond to the coordinates of the first element and v to an array of values. The `rowptr` array of CSR is also kept for fast row accessing. This internal representation is constructed during the loading of the matrix.

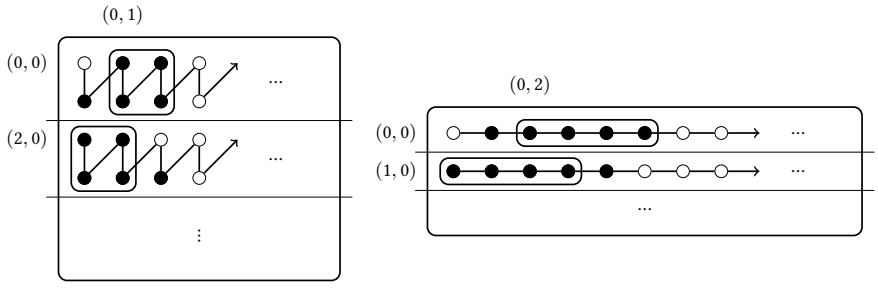
CSX extends the notion of a sequence to include elements with a constant distance greater than one, i.e., n consecutive elements with column indices $\{a, a + d, a + 2d, a + 3d, ..., a + (n - 1)d\}$, where a is the column index of the first element and $d \in \mathbb{N}$, form a sequence. The detection of substructures in CSX involves finding such sequences by applying *run-length encoding* on the (sorted) column indices: first, the delta values (i.e. column distances) are computed for every generic element of the internal representation that hasn't been already encoded and, afterwards, consecutive elements with identical delta values are grouped together in *runs*, with each run forming a single unit. To avoid the potential overhead of very small runs, only runs of size greater or equal to 4 are recognised as CSX units. Also, since the maximum value that can be stored in the `usize` field of the `ctl` structure is 255, large runs are split into 255-chunks. An example of the run-length encoding process is visualized in Figure 2.5 and described in Algorithm 2.1, while the whole detection process of a specific substructure type is given in Algorithm 2.2.

2.2.1 Detecting one-dimensional substructures

The detection of horizontal substructures in CSX occurs by simply applying Algorithm 2.2, since the elements are arranged in a row-wise manner and are lexicographically sorted by default. In order to detect one-dimensional, non-horizontal substructures, however, i.e., vertical, diagonal and anti-diagonal, coordinate transformations are first applied on the elements of the internal

```
1:  function RunLengthEncode(colind::in)
    colind: sorted column indices
2:      deltas ← DeltaEncode(colind)
3:      d ← deltas[0]
4:      f ← 1
5:      rle ← (d, f)
6:      for i ← 1 to N do
7:          if deltas[i] = d then
8:              f ← f + 1
9:          else
10:             if f ≥ min_run_legth then
11:                 rle ← rle ∪ (d, f)
12:             d ← deltas[i]
13:             f ← 1
14:     end for
15:     return rle
```

Algorithm 2.1: Run-length encoding of the column indices. The DeltaEncode() function performs a delta encoding on the column indices and returns the sequence of delta values. The *min_run_length* is currently set to 4.



(a) Original matrix. **(b)** Transformed matrix.

Figure 2.6: Detection of row-aligned blocks with 2-width bands (black dots denote non-zero elements).

representation, so that vertical, diagonal or anti-diagonal sequences are converted to horizontal ones. These transformations are given in Table 2.1. After sorting the indices in the transformed space, units of each substructure type are detected in a similar manner.

2.2.2 Detecting two-dimensional blocks

The process of detecting two-dimensional blocks is a little more complicated, since we need to 'linearize' the coordinates of the elements. First the matrix is partitioned into fixed-width bands, either row- or column-aligned. That is, if we are searching for row-aligned blocks of size k, a $N \times N$ matrix is partitioned into $\frac{N}{k}$ horizontal bands. Afterwards, a space-filling transformation is applied, so that every new line corresponds to a single band (giving $\frac{N}{k}$ lines in total) and every new column in a line corresponds to the position of an element (taking into account both zero and non-zero ones) when the band is run through in a column-wise order (giving $k \times N$ columns in every new line). An example of this transformation for row-aligned bands of size 2 is given in Figure 2.6. One must observe that even though the sequence $\{(0,1),(0,2),(0,3),(0,4),(0,5)\}$ seems to form a unit this is not actually true. For a unit to be a valid block, it must begin at a column index (in the transformed space), which is a multiple of the band width. In this example, $(0,1)$ has a column index which is not a multiple of 2, thus this element is excluded. Regardless, after the transformations have been applied, the resulting sequence is passed to a variant of the RunLengthEncode() function (see Figure 2.1), which takes into account the restriction noted above. In a similar manner, column-aligned blocks come from a vertical partitioning of the matrix and an analogous space-filling transformation that runs through the elements of each band in a row-wise instead of a column-wise order. The CSX format currently supports segmentation bands with width $k \in [2,8]$ (see Table 2.1), leading to seven block-row and seven block-column substructure types.

An minor drawback of these transformations, is that they pose a restriction on the blocks that can be detected by CSX. To be more specific, blocks that cross over bands are not detected. In case of k-width bands, either row- or column-aligned, a block whose first element has a column index (in the transformed space) $j \in \{c \times k + r\}$ where $c \in \mathbb{N}$, $r \in \{1, ..., k-1\}$, is left out. However, this does not seem to limit CSX's block-detecting capabilities. On the contrary, its ability to encode block substructures without padding with explicit zeros, gives CSX an important advantage over existing fixed block formats, such as BCSR and its variants.

2.2.3 Selecting substructures for final encoding

The use of coordinate transformations has allowed CSX to support a total of 18 different substructure types and allows a straightforward expansion to other classes of substructures, e.g., diagonal blocks. We must note here, that

Substructure Type	Transformation
horizontal	$(i', j') = (i, j)$
vertical	$(i', j') = (j, i)$
diagonal	$(i', j') = (N + j - i, min(i, j))$
anti-diagonal	$(i', j') = \begin{cases} (i + j + 1, N - j), & i + j > N - 1 \\ (i + j, i), & i + j \leq N - 1 \end{cases}$
block(row-aligned)	$(i', j') = (\lfloor \frac{i}{k} \rfloor, k \cdot j + mod(i, k))$
block(column-aligned)	$(i', j') = (\lfloor \frac{j}{k} \rfloor, k \cdot i + mod(j, k))$

Table 2.1: The coordinate transformations employed by CSX for the detection of each substructure type (zero-based indexing assumed).

```
1:  procedure DetectSubstructure(matrix::in,stats::inout)
    matrix: CSX's internal representation, lexicographically sorted
    stats: substructure statistics
2:      colind ← ∅                          ▷ Column indices that will be encoded
3:      foreach row in matrix do
4:          foreach generic element e(i, j, v) in row do
5:              if e is not a substructure then
6:                  colind ← colind ∪ e.j
7:                  continue
8:              enc ← RunLengthEncode(colind)
9:              UpdateStats(stats,enc)      ▷ Update statistics for this encoding
10:             colind ← ∅
11:         end for
12:         enc ← RunLengthEncode(colind)
13:         UpdateStats(stats, enc)         ▷ Update statistics for this encoding
14:         colind ← ∅
15:     end for
```

Algorithm 2.2: Detecting substructures in CSX's internal representation. The UpdateStats() function keeps track of the statistics for each substructure type, that will lead to the selection of the most suitable substructure for encoding.

in CSX there is a distinction between a substructure type and its actual instantiation in the sparse matrix. For example, the horizontal substructures with delta values $d = 1$ and $d = 20$ in Figure 2.5 belong to the same substructure type (horizontal), but are different instantiations. A single substructure type, therefore, may have indefinitely many instantiations in a sparse matrix, adding great flexibility to the CSX format.

After the detection process has collected statistics for all the supported substructure types, a filtering process takes place that intends to discard instantiations that fail to meet certain criteria (see Algorithm 2.2). In the current implementation, if a substructure instantiation fails to encode more than 5% of the total non-zero elements of the matrix, it is filtered out. Afterwards, the most suitable type for encoding the matrix is selected and the algorithm proceeds with the encoding. The whole procedure is repeated until no substructure type can be selected for encoding (see Algorithm 2.3). We must note here that the encoding that takes place in this phase is performed on CSX's internal representation. The final data structures of the CSX format (`ctl` and `values`) are constructed after the whole detection-selection-encoding phase has ended.

The fitness metric for selecting the best substructure type for encoding in each step of Algorithm 2.3 tries to maximize the gain in size when replacing the original CSR's `colind` structure with CSX's `ctl` structure. Assuming that we keep a single, full column index per detected substructure (ignoring delta units), the size of the `ctl` structure when encoding a specific substructure type will be:

$$S_{ctl} = N_{units} + NNZ - NNZ_{encoded} \tag{2.1}$$

where N_{units} is the total number of encoded substructure units and $NNZ_{encoded}$ is the number of the non-zero elements encoded by the substructure type. Therefore, the gain in the CSR's `colind` size will be roughly:

$$\begin{aligned} Gain &= S_{colind} - S_{ctl} \\ &= NNZ - (N_{units} + NNZ - NNZ_{encoded}) \\ &= NNZ_{encoded} - N_{units} \end{aligned} \tag{2.2}$$

This metric ensures that the substructure type that is chosen for encoding in every step (SelectType()) of Algorithm 2.3 will be the one that achieves the maximum compression of the matrix at that point. It represents a classic greedy strategy that makes a locally optimal choice and, therefore, doesn't ensure that the final selection of substructure types will be the globally optimal one.

```
1:  procedure EncodeMatrix(matrix::inout)
    matrix: CSX's internal representation
2:      repeat
3:          stats ← ∅
4:          for all available substructure types t do
5:              Transform(matrix, t)
6:              LexSort(matrix)
7:              DetectSubstructure(matrix, stats)
8:              Transform^-1(matrix, t)
9:          end for
10:         FilterStats(stats)          ▷ Filter out instantiations that encode less
                                          than 5% of the non-zero elements
11:         selected ← SelectType(stats)
12:         if selected ≠ NONE then
13:             Transform(matrix, selected)
14:             LexSort(matrix)
15:             EncodeSubstructure(matrix, selected)  ▷ Encode the selected
                                                        substructure
16:     until selected = NONE
```

Algorithm 2.3: Detection, selection and encoding of the substructures in CSX. The process is divided into two phases: (a) gathering of statistics and (b) selection and encoding. Each time the matrix is transformed (Transform()) to a specific iteration order, a lexicographic sort (LexSort()) of the non-zero elements is needed. The $\text{Transform}^{-1}()$ function transforms the matrix back into the original, horizontal iteration order.

We must note here that a selection based exclusively on the size of the final matrix representation can be far from optimal in a number of cases. More precisely, in systems where the matrix size is close to the last level cache size or the number of threads selected for execution is small (1 or 2), the computational part of the kernel is more exposed and therefore, substructure types that do not favor the SpMV execution (i.e. diagonal) may undermine its performance. This phenomenon is even more pronounced in NUMA architectures, where the increased memory bandwidth provided to a processor for accesess to its local memory node, further exposes computationally intensive loads.

The current implementation of the CSX format, does not favor computation friendly substructure types. It does, however, take into account the underlying architecture; in SMP systems, the fitness metric depends solely on

the prediction of the `colind` size reduction, as described previously, while in NUMA systems it also tries to minimize the total number of encoded substructure instantiations, which affects the computational part of the kernel in a way that will become clear in the following section.

2.3 Generating the SpMV code

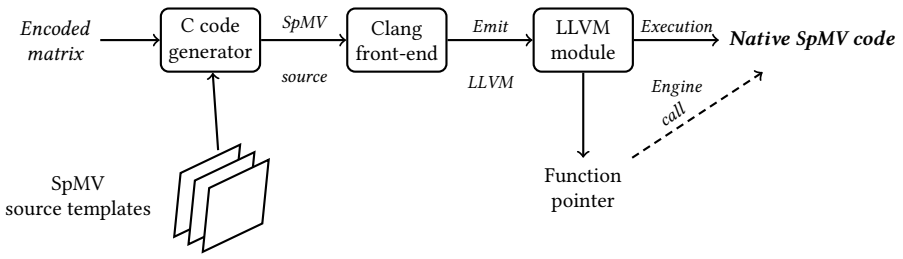
CSX uses a Just-In-Time (JIT) compilation framework in order to generate at runtime the final SpMV code, according to the substructure instantiations encoded in the matrix. This framework is based on the LLVM compiler infrastructure [Lattner and Adve, 2004; Lattner, 2011], a collection of modular and reusable compiler and toolchain technologies, and Clang [Lattner, 2011], a "LLVM native" C/C++/Objective-C compiler that works as a front-end for the LLVM infrastructure. Clang can be used to parse normal C/C++/Objective-C source code, convert it to the LLVM's Intermediate Representation (IR) and pass it to the LLVM back-end for the generation of the native code.

Figure 2.7 gives an overview of the code generation procedure followed by CSX. After the encoding of the matrix, the compilation framework undertakes the task to fill in the missing components, called *hooks*, of the top-level template of the SpMV kernel, which can be viewed in Algorithm 2.4. The __body_hook() is replaced by a switch statement, switching on the unit ID and executing the corresponding SpMV routine. The __new_row_hook() is responsible for advancing the current position y_{curr} in the output vector. We must note here that the switch statement introduces a performance overhead (due to branch mispredictions) that increases with the number of encoded substructure instantiations. This overhead can be noticeable in a multithreaded environment with ample memory bandwidth per processor (e.g, NUMA architectures) where the computational part of the kernel is more exposed. This is the reason why, in NUMA architectures, the heuristic employed by CSX when selecting the substructure types to encode tries to minimize the number of switches.

2.4 Parallelization

The complete process of storing a sparse matrix with the CSX format can be summed up to the following steps:

1. Loading of the matrix and conversion to CSX's internal representation
2. Substructure detection and selection
3. Matrix encoding



(a)

Figure 2.7: The Just-In-Time compilation framework of CSX.

```
1:  procedure CsxSpmv(ctl::in, values::in, x::in, y::out)
2:      y_curr ← y                                  ▷ Current position in y vector
3:      x_curr ← x                                  ▷ Current position in x vector
4:      yr ← 0                                      ▷ Local accumulator
5:      repeat
6:          flags ← *ctl
7:          size ← *(ctl + 1)
8:          ctl ← ctl + 2
9:          if TestBit(flags, 7) then               ▷ Check if nr bit is set
10:             y_curr ← y_curr + yr
11:             yr ← 0
12:             __new_row_hook()                    ▷ Advance y_curr
13:             x_curr ← x
14:         x_curr ← x_curr + DecodeColumn(ctl)
15:         id ← GetID(flags)                       ▷ Retrieve the ID of the next unit
16:         __body_hook()                           ▷ Unit specific SpMV code
17:     until ctl ends
```

Algorithm 2.4: The SpMV kernel template used by the CSX storage format. The __new_row_hook() and __body_hook() are filled at runtime, during the code generation process described in Figure 2.7.

4. SpMV code generation

In order to accelerate both the preprocessing phase and the SpMV kernel in a multithreaded environment, the original matrix is statically split into sub-matrices, one for each available thread, during the construction of the internal representation in step 1 (the actual number of threads employed is user-defined). A naive approach would split the matrix into partitions of $\frac{N}{n}$ rows, where N is the dimension of the matrix and n is the number of threads.

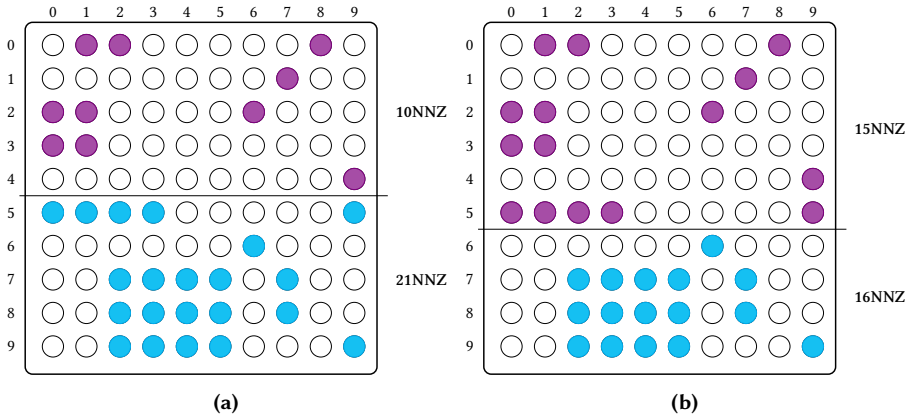


Figure 2.8: Splitting a sparse matrix into sub-matrices in a multithreaded environment. Figure (a) shows the naive one-dimensional row-wise partitioning and figure (b) gives the corresponding one-dimensional row-wise partitioning employed by CSX.

However, this partitioning scheme can lead to a significant load imbalance among threads, especially in matrices where the distribution of the non-zero elements is rather irregular. This derives from the fact that the computational load of each thread is directly proportional to the number of non-zero elements it contains and therefore, a significant divergence in the number of non-zero elements assigned to each thread may lead to idle threads and an increase in the overall execution time. For this reason, the actual partitioning scheme is a one-dimensional row-wise decomposition that tries to maintain the same number of non-zero elements per partition ($\approx \frac{NNZ}{n}$, where NNZ is the total number of non-zero elements). Figure 2.8 exhibits the superiority of this partitioning scheme over the naive approach.

After the sub-matrices have been created, each thread proceeds independently with steps 2 and 3, producing its own CSX sub-matrix. Actually, CSX offers a mechanism that uses statistical sampling to reduce the completion time of step 2 and more specifically the detection of substructure types, without sacrificing performance in the execution of the SpMV kernel [Kourtis et al., 2011; Karakasis et al., 2013]. Finally, step 4 is executed in a single-threaded mode concluding the preprocessing phase. The SpMV kernel may subsequently be executed in a multithreaded manner.

2.5 CSX for symmetric matrices

2.5.1 Exploiting symmetry in the SpMV kernel

Symmetric sparse matrices arise in a large number of real-life applications. These matrices introduce a great opportunity when it comes to reducing the total matrix size, since only the lower (or upper) half of the matrix and its main diagonal suffice to describe the whole matrix. Especially in the case of the SpMV kernel, which is memory bound in most cases, one might be tempted to grasp this opportunity in order to improve performance. This might be the case in a single-threaded execution, but the nature of the symmetric SpMV kernel complicates things in a multithreaded environment.

The most common storage format for symmetric sparse matrices is the *Symmetric Sparse Skyline (SSS)* format [Eisenstat et al., 1982; Saad, 1992]. SSS stores the lower triangular matrix in the CSR format and introduces a new n-size array for the elements of the main diagonal, called *dvalues*, where n is the matrix dimension. Assuming 4-byte integers for indexing and 8-byte double precision floating point values, the SSS format sums up to $4(n+1) + 4\frac{nnz-n}{2} + 8\frac{nnz-n}{2} + 8n = 6(nnz+n) + 4$ bytes. A serial implemetation of the symmetric SpMV kernel using the SSS format is given in Algorithm 2.5.

```
1: procedure SerialSpmvSSS(A::in, x::in, y::out)
   A: matrix in SSS format
   x: input vector
   y: output vector
2:     for r ← 0 to N do
3:         y[r] ← dvalues[r] · x[r]
4:         for j ← rowptr[r] to rowptr[r + 1] do
5:             c ← colind[j]
6:             y[r] ← y[r] + values[j] · x[c]
7:             y[c] ← y[c] + values[j] · x[r]
8:         end for
9:     end for
```

Algorithm 2.5: Symmetric SpMV implementation with the SSS format.

In order to parallelize this algorithm, one could split the lower triangular matrix in a row-wise manner as usual. However, each thread will no longer write exclusively on the range of the output vector y corresponding to the rows it contains, as was the case in the non-symmetric version, but it will also contribute to the computations of its symmetric counterpart.

Thus, each thread may write on any entry $y[i]$ of the output vector where $i \in [0, last\_row_{thread}]$. This leads to race conditions in the output vector, which are particularly increased in matrices with larbge bandwidths. They could be resolved either with the use of locks or with the use of local output vectors per thread. The cost of locks is prohibitive in such a case, leaving us with the local vectors approach as a more viable solution. In this approach, each thread performs Algorithm 2.5 on its own partition writing the results on a local buffer. A reduction step follows in order to compute the final output vector, which can be easily parallelized by assigning each thread the reduction of a specific range of the output vector. Figure 2.9b illustrates a naive implementation of the local vectors method. The main problem with this approach is that the working set of the kernel increases linearly with the number of threads, since each thread produces a local vector of size n. As the number of threads increases, the total size of the local vectors becomes comparable to the matrix size, incurring a significant overhead. The method of *effective ranges* proposed by [Batista et al., 2010] tries to overcome this problem, by restricting the local vectors to "possibly conflicting" regions and performing the rest of the updates directly on the final output vector. The boudaries of these regions are defined based on the following observations:

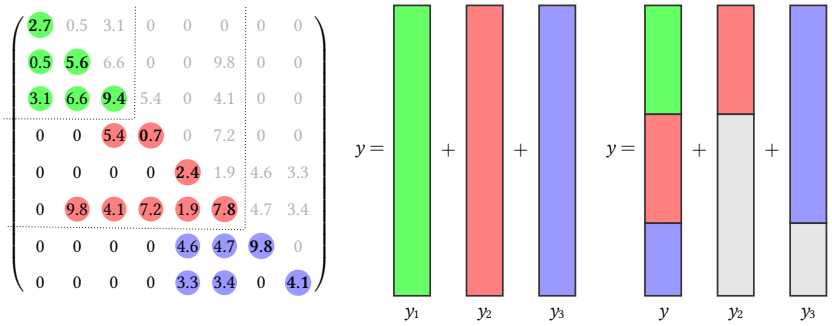
- Thread i never performs updates below the end_i row.
- Thread i may directly perform writes in the $[start_i, end_i]$ range of the final output vector.

In the example of Figure 2.9c, thread 2 (red color) can write the values in positions $(3,2)$, $(3,5)$, $(4,5)$, $(5,1)$, $(5,2)$, $(5,3)$ and $(5,4)$ directly on the final output vector, while those in $(1,5)$, $(2,3)$ and $(2,5)$ fall in the "possibly conflicting" region of the thread and are, therefore, directed to its local vector.

The reduction overhead may be halved with this method, but the main problems still remains: the working set continues to grow linearly to the number of threads.

2.5.2 The CSX-Sym format

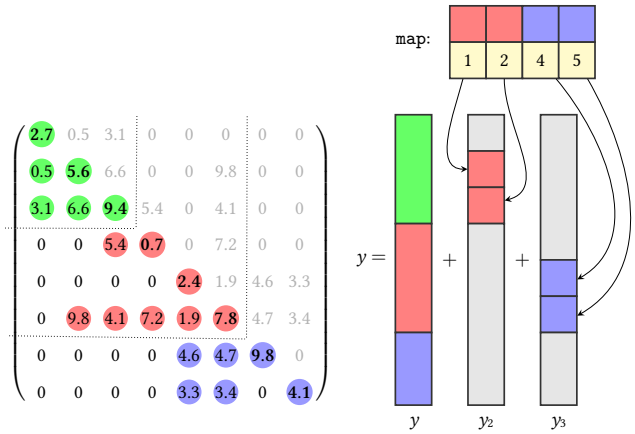
CSX-Sym is a variant of the CSX format that aims to optimize the symmetric SpMV kernel [Gkountouvas et al., 2013]. The detection and encoding of submatrices is performed in a similar manner to CSX, but only on the lower triangular part of the matrix and with a restriction that all of the vector updates of an encoded substructure must be directed either to the original output vector or the local vector; not to both of them. The values of the main diagonal are stored in a new array, called *dvalues*, as in the SSS format. The CSX-Sym format further refines the effective ranges of local vectors method, with the



(a) Sample matrix. **(b)** Naive. **(c)** Effective ranges.

Figure 2.9: Local vector methods for the reduction phase of the symmetric SpMV kernel. The naive method in (b) uses $p = 3$ local buffers that are reduced to the final output vector. The effective ranges method in (c) uses $p - 1$ local vectors ($y2$ and $y3$) writing only to the possibly conflicting regions.

so called *local vectors indexing* scheme. In a nutshell, this scheme introduces an indexing structure, called `map`, that points only to the really conflicting elements of the local vectors. With this indexing scheme the working set of the reduction phase is now dependent on the density of the effective regions of the local vectors. Since the density of the effective regions decreases when the thread count grows, the workload overhead of the reduction step tends to stabilize after a specific number of participating threads [Karakasis, 2012; Gkountouvas et al., 2013]. An example of the CSX-Sym format is given in Figure 2.10.



(a) Sample matrix. **(b)** Local vectors indexing.

(c) Encoded substructures.

Figure 2.10: Local vector method for the reduction phase of the symmetric SpMV kernel in CSX-Sym. In (b), CSX-Sym's indexing scheme uses $p-1$ local vectors and a map pointing only to the actually conflicting elements. In (c), notice that the horizontal substructure at row 6 is not encoded, since its symmetric vertical substructure crosses the thread partition boundary, i.e., values 7.2 and 1.9 are directed to the final output vector while values 9.8 and 4.1 to the local vector. The symmetric counterpart of the block on rows 7-8, on the other hand, is directed entirely to the local vector, thus it is encoded. The actual map will store a pointer only to the first element of this block.

3

The SparseX library

In this chapter, we present the SparseX library, a collection of low-level primitives that provide solver libraries and applications with an implementation of the SpMV kernel using the CSX storage format under an open-source software license. We discuss basic elements of the library design and provide some general implementation details. We also describe the exported BLAS-like user API and illustrate its use with some examples.

3.1 Library design

3.1.1 Goals and motivation

Sparse iterative solver libraries for linear systems that focus on high performance computing, such as the AztecOO and Belos packages of the Trilinos project [Heroux et al., 2005], or lower-level software frameworks that implement the *Basic Linear Algebra Subprograms (BLAS)* functionality for sparse matrices, such as PSBLAS [Filippone and Colajanni, 2000], usually include standard matrix storage formats, such as the CSR and COO formats, thus exhibiting poor performance in many cases. However, some recent projects, including the open-source Portable Extensible Toolkit for Scientific computation (PETSc) [Balay et al., 2013] and Intel's commercial Math Kernel Library (MKL) [Intel® Coorporation, 2013a], have expanded their suite of sparse matrix representations with more elaborate formats, such as the BCSR format, which are much more efficient for particular types of problems.

The CSX format, along with its symmetric variant, are currently among the most highly-optimized sparse matrix storage formats for performing matrix-by-vector multiplications on multicore architectures, especially in the context of iterative methods for the solution of large-scale linear systems [Kourtis et al., 2011; Karakasis et al., 2013]. Its unique performance stability across a wide variety of problems makes CSX an excellent candidate for HPC applications and, thus, we believe CSX's integration in the numerical

software stack would have an important impact on several scientific applications whose overall performance is sensitive to that of the SpMV kernel. To this end, we provide the means to facilitate this process through a low-level interface. Key goals and aspects of our interface sum up to the following:

Provide simple and clear semantics. Every well-designed API should be easy to use correctly and difficult to use incorrectly. A user-friendly syntax reduces the time and intellectual overhead required to develop user software as well as making the development process less error prone. Of course, this is a universal objective in interface design and achieving it depends, to a significant degree, in minimizing the number of things the user must remember in order to effectively use the interface. In the present context, this implies:

- the number of function names the user must remember should be small.
- to the extent possible, the information that functions require as input parameters should be limited to information that the user would necessarily know.

Our interface tries to reflect the above "guidelines" as well as being as *intuitive* as possible; that is, usage of the interface should follow the user's natural train of thought on solving the problem. This objective is usually complicated by the desire to serve users with different levels of expertise.

Facilitate integration to large scale sparse solver libraries. Even though the library can be used directly in applications that involve sparse matrix-by-vector multiplications, its ultimate goal is to integrate readily into application-level libraries that provide high-level sparse kernel support (including iterative solution methods of sparse linear systems), such as the aforementioned PETSc library. Integration in such systems has a number of advantages, including the ability to hide data structure details from the user and, of course, the large potential user base that will assist in the better dissemination of the CSX format.

Transparently adjust to the target platform. The CSX format currently supports symmetric shared memory (SMP) and non-uniform memory access (NUMA) multicore architectures. Any architecture specific optimization is performed transparently during the installation process of the library, eliminating any need for the user to provide information on the hardware platform.

Allow for user inspection and control of the tuning process. Tuning refers to the preprocessing phase of the CSX format, i.e, converting the original sparse matrix into CSX (see Chapter 2.2). A number of parameters

can be explicitly set by the user in order to control different aspects of this phase, such as the number of threads used, choosing between a full or sampling-based detection of substructures, defining specific substructure types to search for in the matrix, selecting between CSX or CSX-Sym as the target format and so on. This adds a lot of flexibility to the tuning process and also affects performance in a direct manner. For example, if the user is aware of the sparsity structure of the matrix (e.g, consisting mainly of blocks) she can guide the detection process by selecting the block substructure types, reducing the execution time of this phase. Of course, a "poor" selection may result in a significant performance degradation, thus the user is advised to rely on the autotuning capabilities of CSX and only override an option when prior knowledge is available, as in the example described previously.

Useful feedback, such as the collected statistics of the detection process and the actual substructures encoded, is provided to help the user gain a better insight into the preprocessing phase of CSX.

3.1.2 Design decisions

The SparseX library is implemented in layers, namely the *core library* and the *user API*. The core library is written in the C++ language, while the user API is written in C. C++ was selected as a programming language that facilitates expressing and implementing an enormous variety of designs in a direct and efficient manner. The decision to export a C API, on the other hand, was driven by the fact that most widely-used iterative solvers for sparse linear systems require APIs in C.

Following an object oriented approach

The library is written in an object-oriented fashion at both development layers. The core library is implemented in a language that by default embraces the object-oriented paradigm, while the user interface makes use of *handles*, the equivalent of an object instance in object-oriented programming terminology.

An object-oriented design is usually defined by the following complementary principles: data encapsulation, polymorphism and inheritance.

Data encapsulation refers to creating objects (data structures) so that application code cannot directly access underlying data in the object. Rather, the application code can modify the data only through the public interface of the object, which consists of a number of subroutine calls. SparseX uses strong data encapsulation in both the sparse matrix and vector objects used

to perform the SpMV kernel. Thus, application-level access to the matrix and vector data is obtained through specific function calls. Encapsulation is vivid throughout both development layers.

Polymorphism refers to techniques that allow the user to call the same function to perform a specific operation regardless of the underlying data structure used to store the data internally. Polymorphism (both runtime and compile-time) exists in the core library of SparseX. For example, when converting the input matrix into the CSX format, the same function is used either the sparse matrix is loaded from a file or stored in the CSR format. This is possible by implementing the function with the use of templates (as a function template) and instantiating it with a different template parameter, which in this case represents the matrix input type. This forms an example of compile-time polymorphism. Runtime polymorphism, on the other hand, exists, for example, in the form of virtual functions in the custom memory allocator employed throughout the core library for efficient data allocations.

Inheritance refers to the process of creating objects either by combining properties of several different types of objects or adding properties to an object that is already defined. This principle was employed in the design of the logging interface of the library, in the context of defining different output policies (logfile, console).

Using templates

When it comes to implementing a numerical library, using a language with generic programming constructs, such as templates in C++, offers a significant advantage: supporting multiple arithmetic types is straightforward and occurs in a typesafe manner. Template classes and functions allow us to use different data types with a single definition, thus reducing source code size. At the same time, they are typesafe, because the data type of the template on which it operates is known and, therefore, checked at compile time (this is known as static or compile-time polymorphism). Also, templates enable code inlining by nature, and thus, allow the compiler to eliminate function calls in many cases, leading to performance benefits. Furthermore, templates can be used instead of virtual functions in many occasions, eliminating the overhead of dynamic polymorphism. On the downside, templates expose their implementation in header files, which means that template code is being compiled in each translation unit that uses it, leading to longer build times and larger binaries. Especially during the development process, this can be particularly "painful", since a single change in a template source code requires a complete rebuild of all project pieces that depend on it.

Using the Boost library

From a programmer's perspective, when it comes to implementing solutions to common problems, one should avoid "reinventing the wheel" and, instead, use existing (efficient and tested) libraries that provide the desired functionality whenever possible. Use of such libraries speeds up initial development, results in fewer bugs and cuts long-term maintenance costs. The Boost library project (http://www.boost.org) provides an elegant and efficient platform- and compiler- independent open-source solution to a wide variety of needed services. Being one of the most highly regarded C++ library projects, Boost tends to become a *de facto* standard and many programmers are already familiar with it. Thus, using Boost when developing a number of features in our project seemed appropriate.

API naming convention

The naming convention for the public interface routines of the SparseX library has the following form:

(return value type) spx_**object**_*function*(...)

Every routine starts with the **spx** prefix, which stands for **Sp**arse**X**, followed by the name of the object it is associated with (usually in a condensed form) and a name that describes its functionality.

3.1.3 Software architecture

Implementing the user interface involved, for the most part, wrapping C around object-oriented C++ code. The core functionality was exposed with the use of the *facade* design pattern. Specifically, the C++ code was built into an internal library (hence the "core library" terminology), including a single C++ module that acts as a facade to the API along with a header file. Access to the core library is, therefore, gained exclusively through the functions declared in this header file. We must note here that since the core library uses templates to support multiple indexing and value types, the facade module must provide explicit template instantiations for the desired template parameters. Currently, the interface is generated for a single combination of indexing and value types, which is chosen at library build-time. A simplified overview of the entire software architecture is given in Figure 3.1.

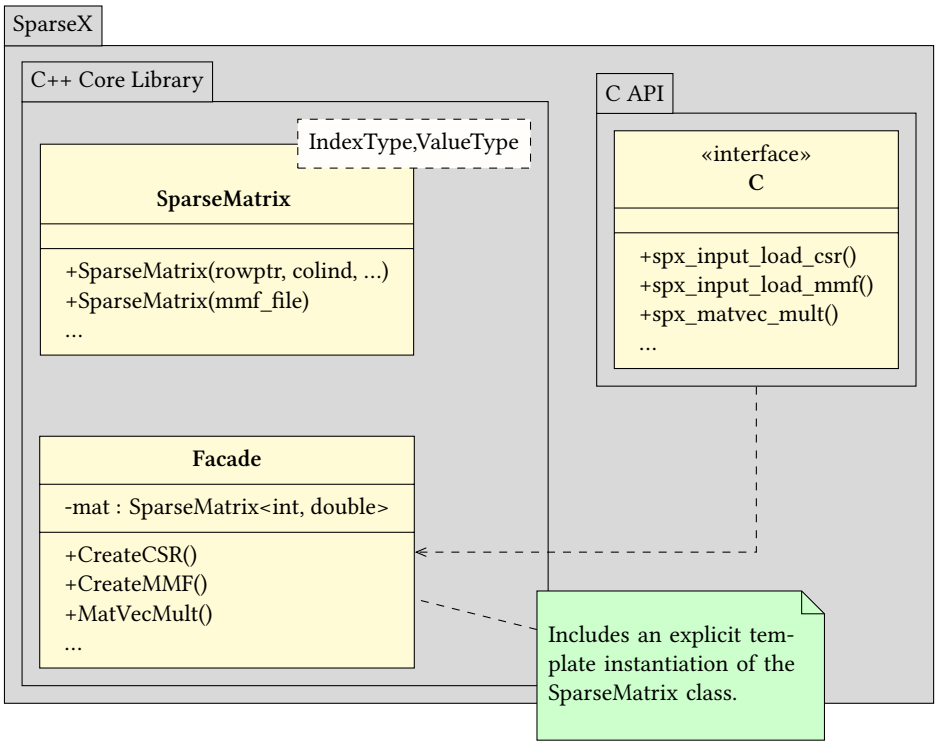


Figure 3.1: An simplified view of the software architecture of SparseX. The facade design pattern is not actually implemented as a class, but as a simple C++ module including all the desired core functionality along with a C/C++ header file that exposes the defined wrapper functions.

3.2 API overview

The API primitives of the SparseX library fall into two broad categories: the *auxiliary routines* and the *computational routines*. The auxiliary routine set includes

- sparse matrix and vector creation and update;
- sparse matrix tuning into the CSX format;
- sparse matrix and vector reordering;
- CSX update;
- CSX I/O.

The computational routine set follows the "look-and-feel" of the BLAS, including

- sparse matrix-by-vector product;
- vector operations (scale, add, subtract, multiply).

The following sections proceed with a thorough presentation of the available routines, both auxiliary and computational, and describe essential ingredients of the user API, including the major *objects* that are intergal parts of it. Some source code examples are provided to give a more practical view of how the API may be used. The complete C bindings appear in Appendix B.

3.2.1 Auxiliary routines

Creating input matrices and vectors

Matrices and vectors are represented by handles in the SparseX interface. The use of handles complements the object-oriented approach of the core library and enables information hiding. Once created, a matrix or vector is referenced only by its handle. The available handles for manipulating matrices and vectors are defined by the following data types:

- `spx_input_t`, which represents a handle to the input matrix;
- `spx_matrix_t`, which represents a handle to the matrix in the CSX format;
- `spx_vector_t`, which represents a handle to a dense vector object.

We must note here that since our API aims at exporting utilities for the CSX format alone, matrices stored in different formats are only treated as "input" matrix representations, hence the distinction between the `input_t` and `matrix_t` handles.

The interface currently supports loading a sparse matrix from the following formats:

Matrix Market File (MMF) format. In dealing with issues of I/O, SparseX is currently designed to support reading a sparse matrix from a file in the MMF format [Boisvert et al., 1996]. File input is embedded as another form of a sparse matrix constructor. A MMF file consists of four parts:

1. **Header line**: this is the first line in the file and contains an identifier and four text fields in the following form

   %%MatrixMarket *object format field symmetry*

   where *object* is either **matrix** (this is the case we will consider here) or **vector**, *format* can be either **coordinate** for sparse matrices or **array** for dense matrices, *field* is either **real**, **double**, **complex**, **integer** or **pattern** and, finally, *symmetry* is either **general**, **symmetric**, **skew-symmetric** or **hermitian**.
2. **Comment lines**: begin with a percent sign and allow a user to store information and comments.

3. **Size line**: specifies the number of rows, columns and nonzero elements in the matrix.
4. **Data lines**: specify the location of the matrix entries and their values. When the matrix is sparse, the location of the matrix entries is given in the *coordinate* format using one-base indexing in a column-wise ordering.

Since CSX operates on the elements of a sparse matrix in a row-wise order, the column-wise ordering of the MMF format creates the need to sort the elements when loading the matrix. Thus, the format has been extended in order to also support row-wise ordering of the elements and zero-base indexing by introducing two optional fields in the header line called *indexing* which can be either **0-base** or **1-base** and *ordering* which can be either **column** or **row**. Furthermore, a simplified version of the MMF format is supported, which drops the header line and includes only the size line and data lines. In this case, however, the data must be arranged in a row-wise order.

Compressed Sparse Row (CSR). An input handle can also be created from an existing user-allocated, pre-assembled matrix in the CSR format. The CSR data structures (`rowptr`, `colind` and `values`) are "shared" in this case with the library, thus the user must guarantee they will not be freed or reallocated before the destruction of the input handle. Both zero-based and one-based indexing is supported by setting the appropriate argument in the corresponding routine.

Dense vector objects of type `vector_t` can be either wrappers of user-defined arrays or they can be created and initialized explicitly by the user. Partitioning a vector among threads is performed transparently during the creation process and, thus, cannot be controlled by the user. The only responsibility of the user is to provide the split points to the vector creation routine through an object of type `spx_partition_t`, which is constructed either implicitly during the tuning phase of CSX and subsequently extracted with the `spx_mat_get_partition()` routine or explicitly through the `spx_partition_csr()` routine. This will become more clear in the following section.

Table 3.1 summarizes the available routines for input matrix and vector creation.

Tuning

The user may convert the input matrix into the CSX format simply by calling the `spx_mat_tune()` routine. However, in order to fully exploit the capa-

Routine	Description
`spx_input_load_mmf`	Creates an input object from a file on disk in the MMF format.
`spx_input_load_csr`	Creates an input object from a matrix in the CSR format.
`spx_input_destroy`	Frees an input object.
`spx_vec_create`	Creates a vector object.
`spx_vec_create_from_buff`	Creates a vector object as a wrapper of a user-defined array.
`spx_vec_create_random`	Creates a randomly filled vector object.
`spx_vec_destroy`	Destroys a vector object.

Table 3.1: Available routines for creating and destroying input and vector objects.

bilities of CSX and achieve the best performance, the user is advised to make use of the `spx_set_option()` routine. There are options for controlling: (a) the runtime environment, (b) the preprocessing phase of CSX and (c) the CSX format itself. The available options with their default values for every category are given in Table 3.2.

The options for setting the runtime environment include the number of participating threads (`spx.rt.nr_threads`) and the processor affinity (`spx.rt.cpu_affinity`). If these options are not explicitly set by the user the library automatically detects the optimal configuration, i.e., the number of threads is set to the number of available cores and the CPU affinity is adjusted according to a 'share-nothing' core-filling policy, which assigns threads to cores so that the least resource sharing is achieved.

The user can also control different aspects of the preprocessing phase. For one, she may select specific substructure types to be encoded through the `spx.preproc.xform` option. She can also enable/disable the use of statistical sampling in the detection process (`spx.preproc.sampling`). As pointed out in the previous chapter, the preprocessing cost of CSX can be significantly halved by enabling the use of sampling. The default values for the portion of the matrix that will be sampled and the number of sampling windows used per thread can be overridden with the `spx.preproc.sampling.portion` and `spx.preproc.sampling.nr_samples` options respectively. The last two parameters are used to compute the window size according to

Option	Default value
spx.rt.nr_threads	1
spx.rt.cpu_affinity	0
spx.preproc.xform	none
spx.preproc.sampling	none
spx.preproc.sampling.nr_samples	10
spx.preproc.sampling.portion	0.01
spx.matrix.symmetric	false
spx.matrix.split_blocks	true
spx.matrix.full_colind	true (NUMA) false (SMP)
spx.matrix.min_unit_size	4
spx.matrix.max_unit_size	255
spx.matrix.max_coverage	0.1

Table 3.2: Available options for configuring the preprocessing phase of CSX.

the following equation:

$$\begin{aligned} window\_size &= \frac{sampling\_portion \cdot nr\_nzeros_{per\_thread}}{nr\_samples} \\ &= \frac{sampling\_portion \cdot nr\_nzeros_{total}}{nr\_samples \cdot nr\_threads} \end{aligned} \tag{3.1}$$

The above equation designates that the window size has an inverse relation to the number of threads for fixed values of the sampling portion and the number of samples. As the number of threads increases it is possible that the sampling window will become too small and limit/constrain the detection capabilities of CSX. For example, if the window contains only a single row then only horizontal and delta units can be detected. Thus, when the number of participating threads is large, the user is advised to set a smaller number of samples, especially in case of small matrices. The user might of course achieve a similar effect by increasing the sampling portion. We plan to address this issue by further refining the heuristic employed for computing the window size.

From the last group of options, the most important for the user is the `spx.matrix.symmetric` option which enables the use of the symmetric variant of the CSX format. The rest of the options are configured by default according to a 'best practice' policy, which also takes into account the underlying architecture. For a description of these options refer to Appendix B.

Reordering

When the matrix structure is very irregular, resulting in an equally irregular access pattern in the right-hand side vector, a significant amount of cache misses is introduced. Additionally, when the matrix suffers from an heterogeneous sparsity pattern (see *G3_circuit* in Appendix A) a varying flop:byte ratio is exhibited among the participating threads, due to fluctuations in the nonzero density across the matrix, leading to significant load imbalances. Reordering the matrix using a bandwidth-reduction technique has been proposed as a solution to the aforementioned problems. This involves applying row and column permutations in order to bring the nonzero elements of the matrix as close as possible to the main diagonal. This is beneficial to a typical SpMV implementation, since the homogenization of the nonzero element distribution leads to a better access pattern and load balance. For the symmetric version of the kernel (using SparseX with the `spx.matrix.symmetric` option enabled), the obvious effect of this nonzero rearrangement in a mutlithreaded execution is the minimization of the reduction phase overhead.

Our user API currently provides an option for applying the *Reverse Cuthill McKee (RCM)* reordering algorithm that has been proposed for structurally symmetric matrices [Cuthill and McKee, 1969]. Reordering is defined as an optional argument of the `spx_mat_tune()` routine, but it can also be performed explicitly with the `spx_input_reorder()` routine, while the `spx_mat_get_perm()` and `spx_vec_reorder()` routines allow the user to extract and apply the corresponding permutation on vector objects.

Changing matrix nonzero values

The nonzero pattern of the input matrix determines the nonzero pattern of the `spx_matrix_t` handle, i.e., the CSX-tuned matrix, but the nonzero values can be modified by the user. If the input matrix contains explicit zeros, the library treats them as "logical nonzeros" whose values can be modified later. To change individual entries in the tuned matrix, the user may call the `spx_mat_set_entry()` routine. The corresponding routine for obtaining a single value is the `spx_mat_get_entry()`. Both routines assume one-base indexing in the supplied coordinates of the matrix entry.

Storing and retrieving CSX

Although significantly optimized, the preprocessing cost of CSX is still non-negligible. This motivated us to implement an I/O feature that will allow the user to avoid tuning a matrix that has already been converted to the CSX format in a previous session. This feature involves serializing the tuned

matrix handle and storing it in a binary file, so it can be used in a future session to reconstruct an equivalent handle and directly perform the SpMV kernel. This achieves a considerable speedup over the optimized preprocessing phase of CSX (a detailed evaluation is available in Chapter 4). Our user API provides the `spx_mat_save()` routine for storing the matrix and the `spx_mat_restore()` routine for loading it from the binary file.

3.2.2 Computational routines

This module includes two routines, shown in Table 3.3, to perform the sparse matrix-by-vector multiplication. The most generic version of the kernel was implemented, i.e. $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$, where A is the sparse matrix, x and y are vectors and *alpha* and *beta* are scalars. The first routine is equivalent to the BLAS Level 2 routine for matrix-by-vector multiplication. Parameter ordering follows the BLAS convention. The second routine, on the other hand, is a higher-level implementation of the kernel that hides the preprocessing phase of CSX. This last routine can be efficiently used in a loop, since only the first call will convert the matrix into the CSX format and every subsequent call will use the previously tuned matrix handle.

Use of the aforementioned multiplication routines is illustrated with a couple of examples, that also highlight different aspects of our API. The sequence of operations in a minimal application, that simply performs a loop of multithreaded matrix-by-vector multiplications with the use of the `spx_matvec_mult_from_csr()` routine, is shown in Listing 3.1, while Listing 3.2 gives an example that includes reordering and exposes the tuning phase.

Routine	Description
`spx_matvec_mult`	Performs the SpMV kernel $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$ with a tuned matrix handle as input.
`spx_matvec_mult_from_csr`	Performs the SpMV kernel $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$ with the CSR format as input.

Table 3.3: Available routines for sparse matrix-by-vector multiply.

Listing 3.1: A usage example the hides tuning and executes in a multithreaded mode. In this example the higher-level multiplication routine is selected for executing 128 iterations of the SpMV kernel in a multithreaded mode. Specifically, 2 threads will be used with the selected CPU affinity (0,1). All the substructure types are selected for detection, but tuning is performed without the use of sampling. The matrix in the CSR format is first split into partitions, the x and y vectors are created from user-defined arrays and finally the loop is executed.

```
1  /* Define CSR data structures */
2  spx_index_t rowptr[] = {0,5,6,10,15,18,22,24,29,33,38};
3  spx_index_t colind[] = {0,1,2,3,8,7,0,1,6,9,0,1,3,5,9,0,1,
4                          9,0,1,5,9,2,3,2,3,4,5,7,2,3,4,5,2,
5                          3,4,5,9};
6  spx_value_t values[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,
7                          15,16,17,18,19,20,21,22,23,24,25,
8                          26,26.1,26.2,27,28,29,29.1,29.2,
9                          30,31,31.1,31.2,32};

11 /* Define vector arrays */
12 spx_value_t x[] = {1,2,3,4,5,6,7,8,9,10};
13 spx_value_t y[] = {0.19,0.28,0.31,0.42,1.32,
14                    2.64,0.75,2.36,0.91,1};

16 spx_index_t nrows, ncols;
17 spx_scalar_t alpha, beta;

19 nrows = 10; ncols = 10;
20 alpha = 0.42; beta = 0.10;

22 /* Initialize library */
23 spx_init();

25 /* Set tuning options */
26 spx_option_set("spx.rt.nr_threads", "2");
27 spx_option_set("spx.rt.cpu_affinity", "0,1");
28 spx_option_set("spx.preproc.xform", "all");

30 /* Partition matrix */
31 spx_partition_t *parts = spx_partition_csr(rowptr, nrows,
       2);

33 /* Create vectors from arrays */
34 spx_vector_t *x_view = spx_vec_create_from_buff(x, ncols,
       parts);
35 spx_vector_t *y_view = spx_vec_create_from_buff(y, nrows,
       parts);

37 /* Declare a tuned matrix handle */
38 spx_matrix_t *A = INVALID_MAT;
```

```
40 /* Run 128 iterations of the SpMV kernel:
41        y <- alpha*A*x + beta*y */
42 int i;
43 for (i = 0; i < 128; i++) {
44     spx_csr_matvec_mult(A, nrows, ncols, rowptr, colind,
45                         values, alpha, x_view, beta, y_view
                                );
46 }

48 /* Cleanup interface objects */
49 spx_mat_destroy(A);
50 spx_vec_destroy(x_view);
51 spx_vec_destroy(y_view);
52 spx_partition_destroy(parts);

54 /* Shutdown library */
55 spx_finalize();
```

Listing 3.2: A usage example with reordering and sampling enabled. The matrix in this example is loaded from a file in the MMF format. The tuning is performed explicitly with sampling enabled, using 48 sampling windows and detecting all substructure types. Reordering is enabled with the `OP_REORDER` option of the `spx_mat_tune()` routine. We must observe that the x and y vectors must be explicitly reordered before executing the kernel and inverse-reordered after the execution.

```
1  /* Initialize library */
2  spx_init();

4  /* Load matrix from file */
5  spx_input_t *input = spx_input_load_mmf(file);

7  /* Set some tuning options */
8  spx_option_set("spx.preproc.xform", "all");
9  spx_option_set("spx.preproc.sampling", "portion");
10 spx_option_set("spx.preproc.sampling.nr_samples", "48");

12 /* Transform to CSX */
13 spx_matrix_t *A = spx_mat_tune(input, OP_REORDER);

15 /* Create randomly filled vectors */
16 spx_index_t ncols = spx_mat_get_ncols(A);
17 spx_index_t nrows = spx_mat_get_nrows(A);
18 spx_partition_t *parts = spx_mat_get_partition(A);
19 spx_vector_t *x = spx_vec_create_random(ncols, parts);
20 spx_vector_t *y = spx_vec_create_random(nrows, parts);

22 /* Reorder vectors */
23 spx_perm_t *p = spx_mat_get_perm(A);
```

```
24  spx_vec_reorder(x, p);
25  spx_vec_reorder(y, p);

27  /* Run the SpMV kernel:
28     y <- alpha*A*x + beta*y */
29  spx_matvec_mult(alpha, A, x, beta, y);

31  /* Inverse-reorder the output vector */
32  spx_vec_inv_reorder(y, p);

34  /* Cleanup interface objects */
35  spx_input_destroy(input);
36  spx_mat_destroy(A);
37  spx_vec_destroy(x);
38  spx_vec_destroy(y);
39  spx_partition_destroy(parts);

41  /* Shutdown library */
42  spx_finalize();
```

3.3 Logging

Logging is a critical technique for troubleshooting and maintaining software systems. The logging framework of SparseX was designed to be typesafe, threadsafe (at line level), flexible and as light-weight as possible. It adopts the look-and-feel of C++ streams and can be enabled or disabled both at compile time and runtime.

The architecture of the framework has three parts: (a) the front-end (b) the core and (c) the back-end.

The front-end is the top of the framework providing the actual logging functions and a number of utilities to determine how the framework will operate.

The core consists of a "logging handler", responsible for enabling or disabling specific logging levels and formatting the data of each level, thus, fully controlling the verbosity of the outputted data. The available logging levels are `Error`, `Warning`, `Info` and `Debug`. The actual redirection of the data to the back-end is performed by another entity, that implements the stream-like behavior available in the front-end.

The back-end defines an output device for the logged data, called a *sink*. Currently, the logging framework implements three sinks, namely `Console`, `File` and `Null`. The first two redirect the data to stderr and a file respectively, while `Null` forms a special sink used for deactivation of the logging process at runtime. The back-end can be easily extended to support additional sinks.

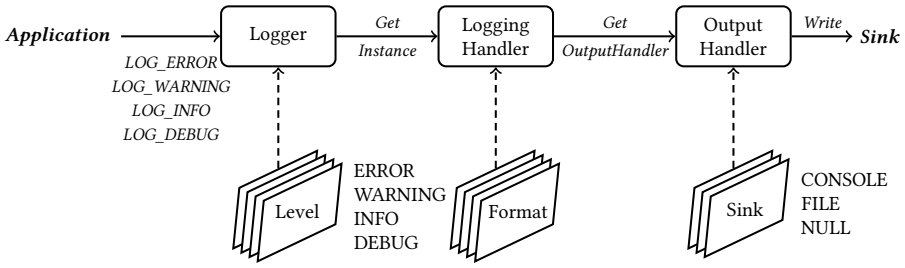


Figure 3.2: Data flow through the logging framework of the SparseX library.

Conceptually, data flows through the logging framework as shown in Figure 3.2.

By default, only the `Error` and `Warning` logging levels are enabled. However, the user may find the `Info` level particularly helpful in understanding the preprocessing phase of CSX while evaluating program performance. For example, `Info` activates the printing of statistics about the detected and encoded substructures during the conversion to the CSX format. Additionally, in case of NUMA architectures, where the performance of the SpMV kernel is sensitive to the correct placement of the involved data on the system's memory nodes, information is given on the success or failure of the corresponding memory allocations. Table 3.4 gives an overview of the available routines for configuring the logging process.

Routine	Description
`spx_log_enable_all_console`	Activates all logging levels and redirects output to stderr.
`spx_log_enable_all_file`	Activates all logging levels and redirects output to a logfile.
`spx_log_disable_all`	Disables logging.
`spx_log_disable_error`	Disables the `Error` logging level.
`spx_log_disable_warning`	Disables the `Warning` logging level.
`spx_log_disable_info`	Disables the `Info` logging level.

Table 3.4: Available routines to control logging.

3.4 Error handling

In general, the SparseX interface distinguishes three types of errors: (a) fatal errors generated by the operating system, (b) logical errors (i.e., created by the user) that are fatal to the program execution and (c) logical errors that

do not lead to program failure The first category may include memory- and file-related errors. The second category may include errors due to invalid arguments supplied to one of the interface's routines. Finally, non-fatal logical errors may occur, for example, when trying to set an option to an invalid value.

The interface handles errors with the use of error handling routines and error codes or invalid handle returns. When an error condition is detected within a SparseX routine, it is treated as following:

- The routine calls the current error handler.
- Regardless of what action the error handler performs, the routine returns an error code or an invalid handle depending on the routine.

The default error handler uses the logging framework described in the previous section to output messages of the following form

[*prefix*]: *message* [*sourcefile*:*lineno*:*function*]

where *prefix* can be either `ERROR` or `WARNING` depending on the error type. When an error of the first category occurs, the default error handler also outputs the error message generated by the operating system and subsequently exits the program with a nonzero exit code. In case of logical errors, on the other hand, returning error codes instead of exiting seemed more appropriate, since most routines make requests on available resources and their failure needs to be recoverable.

If the user wishes to handle errors in a different manner, she may set her own error handling routine by calling `spx_err_set_handler()`, as long as it conforms to the signature set by our interface. For more details see Appendix B.

A couple of macros are used to make the error handling a bit more convenient. These macros are used throughout the interface and can be employed by the application programmer as well. When an error is first detected, one should set it by calling

`SETERROR_0(error_code)` or
`SETERROR_1(error_code, message)`

Both macros call the current error handler, while the second also supplies a custom message.

4

Evaluating the performance of SparseX

In this chapter, we assess the potential benefit of using SparseX in solver libraries or standalone applications by running tests on representative data and a variety of advanced multicore architectures. A performance evaluation of the CSX file feature indicates that the cost of tuning can be amortized in future runs of the same matrix, while comparison to other popular libraries substantiates previous claims on CSX's (and CSX-Sym's) improved performance over alternative sparse matrix storage formats [Kourtis et al., 2011; Karakasis et al., 2013; Gkountouvas et al., 2013].

4.1 Experimental setup

Before proceeding with our quantitative evaluation of the SparseX library, it is essential to present the matrices, the hardware platforms and the experimental methodology we used for our benchmarking.

4.1.1 Matrix suite

In order to evaluate the performance of our library, we have selected 30 matrices from the University of Florida Sparse Matrix Collection [Davis and Hu, 2011], covering a wide variety of application domains. This diverse set of matrices exhibits varying properties relevant to SpMV performance, including matrix dimension, sparsity, average number of nonzeros per row, the existence of dense block substructures and symmetry, allowing us to examine different aspects of the kernel. Appendix A provides images that depict the sparsity structure of each matrix in our suite.

Table 4.1 gives the properties of the selected matrices. Two thirds of the matrices are derived from problems with an underlying 2D/3D geometry, since these are more frequently encountered in the solution of sparse linear systems. Nine matrices have rather short rows, exhibiting a very low arithmetic intensity with a flop:byte ratio ranging below 1.5. The most sparse

matrix is Hamrle3, which has only four non-zeros per row on average, while the denser is TSOPF_RS_b2383 with 424 elements per row. We must note that every matrix in our suite does not fit in the last level cache of each of our test platforms. This choice was made in order to highlight the effect of compression on the resulting memory traffic.

Matrix	Dimension	Nonzeros	Size (MiB)	f:b ratio	Problem domain	2D/3D
xenon2	157,464	3,866,688	44.85	0.156	Materials	Yes
ASIC_680k	682,862	3,871,773	46.91	0.129	Circuit Sim.	No
torso3	259,156	4,429,042	51.67	0.152	Other	Yes
Chebyshev4	68,121	5,377,761	61.80	0.163	Structural	Yes
Hamrle3	1,447,360	5,514,242	68.63	0.116	Circuit Sim.	No
pre2	659,033	5,959,282	70.71	0.141	Circuit Sim.	No
cage13	445,315	7,479,343	87.29	0.152	Graph	No
atmosmodj	1,270,432	8,814,880	105.72	0.135	C.F.D.	Yes
ohne2	181,343	11,063,545	127.30	0.162	Semiconductor	Yes
kkt_power	2,063,494	14,612,663	175.10	0.135	Optimization	No
TSOPF_RS_b2383	38,120	16,171,169	185.21	0.166	Power	No
Ga41As41H72	268,096	18,488,476	212.61	0.163	Chemistry	No
Freescale1	3,428,755	18,920,347	229.61	0.128	Circuit Sim.	No
rajat31	4,690,002	20,316,253	250.39	0.120	Circuit Sim.	No
F1	3,428,755	26,837,113	308.44	0.163	Structural	Yes
parabolic_fem	525,825	3,674,625	44.06	0.135	C.F.D.	Yes
offshore	259,789	4,242,673	49.54	0.151	Electromagnetics	Yes
consph	83,334	6,010,480	69.10	0.163	F.E.M.	Yes
bmw7st_1	141,347	7,339,667	85.54	0.161	Structural	Yes
G3_circuit	1,585,478	7,660,826	93.72	0.124	Circuit Sim.	No
thermal2	1,228,045	8,580,313	102.88	0.135	Thermal	Yes
m_t1	97,578	9,753,570	111.99	0.164	Structural	Yes
bwmcra_1	148,770	10,644,002	122.38	0.163	Structural	Yes
hood	220,542	10,768,436	124.08	0.161	Structural	Yes
crankseg_2	63,838	14,148,858	162.16	0.165	Structural	Yes
nd12k	36,000	14,220,946	162.88	0.166	Other	Yes
af_5_k101	503,625	17,550,675	202.77	0.159	Structural	Yes
inline_1	503,712	36,816,342	423.25	0.163	Structural	Yes
ldoor	952,203	46,522,475	536.04	0.161	Structural	Yes
boneS10	914,898	55,468,422	638.28	0.162	Model Reduction	Yes

Table 4.1: The matrix suite used for experimental evaluation. For each matrix, we provide the matrix dimension (square matrices only), the number of nonzero elements, the size and the arithmetic intensity (flop:byte ratio) in the CSR format, the problem domain and whether it is derived from a problem with an underlying 2D/3D geometry.

	Dunnington	Gainestown	Westmere-EP	Sandy Bridge-EP
Model	Intel Xeon X7460	Intel Xeon X5560	Intel Xeon X5650	Intel Xeon E5-4620
Microarchitecture	Intel Core	Intel Nehalem	Intel Westmere	Intel Sandy Bridge
Clock frequency	2.66 GHz	2.8 GHz	2.66 GHz	2.26 GHz
L1 cache (D/I)	32 KiB/32 KiB	32 KiB/32 KiB	32 KiB/32 KiB	32 KiB/32 KiB
L2 cache	3 MiB *(per 2 cores)*	256 KiB *(per core)*	256 KiB *(per core)*	256 KiB *(per core)*
L3 cache	16 MiB	8 MiB	12 MiB	16 MiB
Cores/Threads	6/6	4/8	6/12	8/16
Sustained memory b/w	8.1 GiB/s	2×15.5 GiB/s	2×15.7 GiB/s	4×13.5 GiB/s
Multiprocessor configurations				
Sockets	4	2	2	4
Cores/Threads	24/24	8/16	12/24	32/64

Table 4.2: Technical characteristics of the hardware platforms used for the experimental evaluations. The sustained memory bandwidth figures are obtained with the STREAM benchmark [McCalpin, 1995] utilizing the full system.

4.1.2 Hardware platforms

The hardware platforms we use for our quantitative analysis comprise of one symmetric shared memory (SMP) and three cache-coherent non-uniform memory access (cc-NUMA) multiprocessor systems. The SMP system is a four-way six-core Intel Xeon X7460 (codename Dunnington) multiprocessors. The NUMA systems are a two-way quad-core Intel Xeon X5560 (codename Gainestown) multiprocessor, a two-way six-core Intel Xeon X5650 (codename Westmere-EP) and a four-way six-core Intel Xeon E5-4620 (codename Sandy Bridge-EP). In the following, we will refer to each platform using its codename. Table 4.2 lists the technical specifications of our test platforms.

In our SMP system, Dunnington, all four sockets use the common bus to communicate with the main memory and with each other, a layout that can become a serious bottleneck for very memory-intensive multithreaded applications, like the SpMV kernel.

The Gainestown, Westmere-EP and Sandy Bridge-EP systems depart from the centralized SMP logic by moving the memory controller inside the socket and splitting, as a result, the physical memory into per-processor banks. Each memory controller can provide a sustained memory bandwidth almost 2× faster than the one offered by our SMP architecture. This means that accesses addressed to a processor's local memory are cheaper in terms of memory latency. Interprocessor communication, as well as remote memory accesses, are routed through a dedicated interconnection network (Intel® Quick-Path Interconnect – QPI [Kurd et al., 2008]) with a bandwidth much smaller than the available memory bandwidth. Therefore, the interconnection link is likely to become a bottleneck if a significant amount of main memory traffic is routed through it due to remote memory accesses

[Goumas et al., 2009]. We expect this layout to alleviate the SpMV kernel in terms of memory intensity, assuming correct data placement is performed among the processors.

Software setup

All systems run a 64-bit version of the Linux OS (kernel version 3.7.10) and the GNU Compiler Collection (gcc, g++ etc.), version 4.6, is used for the compilation of every project, unless stated otherwise. Multithreaded versions of the SparseX code (including the SpMV kernel) are written using explicit, native threading with the Pthreads userspace library (NPTL, version 2.11.3), while NUMA-aware versions are built with the numactl library (version 2.0.8), which is a wrapper userspace library of the low-level memory allocation system call interface of the Linux kernel. Finally, the LLVM/Clang compiler framework, version 3.0, is used for the runtime code generation of CSX.

4.1.3 Measurement policies

Thread placement

In modern multicore processors, a core is not an independent processing element, but rather a part of a larger on-chip system and, hence, shares resources (pipeline, cache hierarchy, bus/memory interfaces) with other cores. For example, in our Gainestown platform, two threads may share all the preprocessor resources from the pipeline up to the memory controller, if they are placed on the same logical core (this technology is known as Simultaneous Multithreading (SMT) [Tullsen et al., 1995] and has been initially commercialized as Hyper-Threading (HT) by Intel in its Netburst microarchitecture [Koufaty and Marr, 2003]), but they may also share nothing, if they are placed on different sockets. It has been documented that the execution time of a thread can vary greatly depending on which threads are running on the other cores of the same chip [Lin et al., 2008]. Therefore, the way threads are assigned to different hardware contexts, denoted as *thread placement*, has a significant impact on overall system performance, depending on the application's workload. It is therefore essential to pin threads to specific cores during measurements following a specific resource sharing control scheme. In CSX, thread pinning is performed with the `sched_setaffinity()` Linux kernel system call, which allows to assign the calling thread to an arbitrary set of logical CPUs. For the assignment of threads to cores, we define two different policies:

- *share-all*: This policy assigns threads to cores so that the maximum sharing of resources is achieved, with the exception of pipeline resources (Hyper-Threading feature). In Dunnington, for example, the cores sharing the L2 cache will be filled first, followed by the cores sharing the L3, and so forth for the rest of the sockets. The advantage of this policy is that it provides more insight of the performance behavior as we scale a system by adding more sockets.
- *share-nothing*: This policy assigns threads to cores so that the least resource sharing is achieved. Taking on the example of Dunnington, the threads will be first spread across the sockets, then across the L2 caches, and finally start to share. The advantage of this policy is that it utilizes the full system's potential right from the configurations with a small number of threads.

Data types

To simulate the typical sparse matrix storage case, we use 32-bit integers for indexing and 64-bit, double precision floating point values for the nonzero elements.

4.2 CSX file evaluation

In this section we perform an evaluation of the CSX file feature provided by the SparseX library. As mentioned in the previous chapter, the user may save a sparse matrix already tuned in the CSX format with the `spx_mat_save()` routine. In a future session, she may retrieve the disk-cached CSX matrix with the `spx_mat_restore()` routine and perform a multiplication as usual. This eliminates the need to load the original matrix and go through the entire preprocessing phase of CSX (actually, only the step that involves the SpMV code generation is repeated, see Section 2.4). If the user also wishes to change some values in the freshly loaded matrix, she can make use of the `spx_set_entry()` routine. The only (obvious) limitation is that the same number of threads will be used. Thus, from the available runtime options (see Table 3.2), only the CPU affinity may be overridden. Listing 4.1, gives an example of how this feature may be used.

Listing 4.1: A usage example that loads CSX from a file.

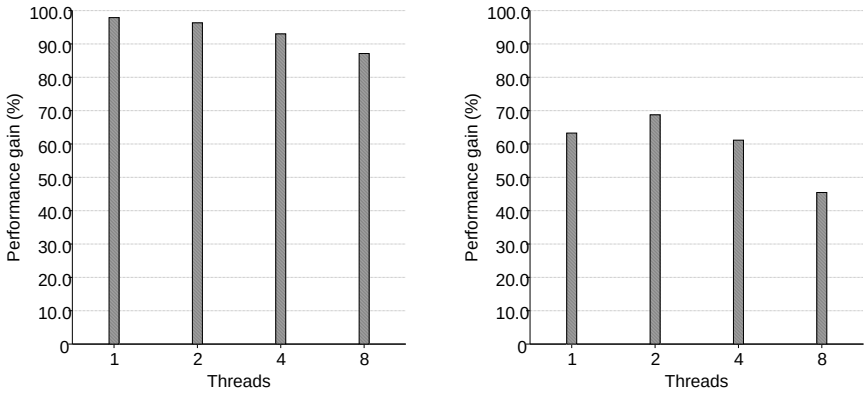
```
1  spx_scalar_t alpha, beta;
2  alpha = 0.42; beta = 0.10;

4  /* Initialize library */
```

```
5  spx_init();

7  /* Retrieve matrix from binary file */
8  spx_matrix_t *A = spx_mat_restore(file);

10 /* Create randomly filled vectors */
11 spx_index_t ncols = spx_mat_get_ncols(A);
12 spx_index_t nrows = spx_mat_get_nrows(A);
13 spx_partition_t *parts = spx_mat_get_partition(A);
14 spx_vector_t *x = spx_vec_create_random(ncols, parts);
15 spx_vector_t *y = spx_vec_create_random(nrows, parts);

17 /* Run the SpMV kernel:
18    y <- alpha*A*x + beta*y */
19 spx_matvec_mult(alpha, A, x, beta, y);

21 /* Cleanup interface objects */
22 spx_mat_destroy(A);
23 spx_vec_destroy(x);
24 spx_vec_destroy(y);
25 spx_partition_destroy(parts);

27 /* Shutdown library */
28 spx_finalize();
```

In order to measure the performance gain in terms of preprocessing cost, the following benchmark scenario was executed:

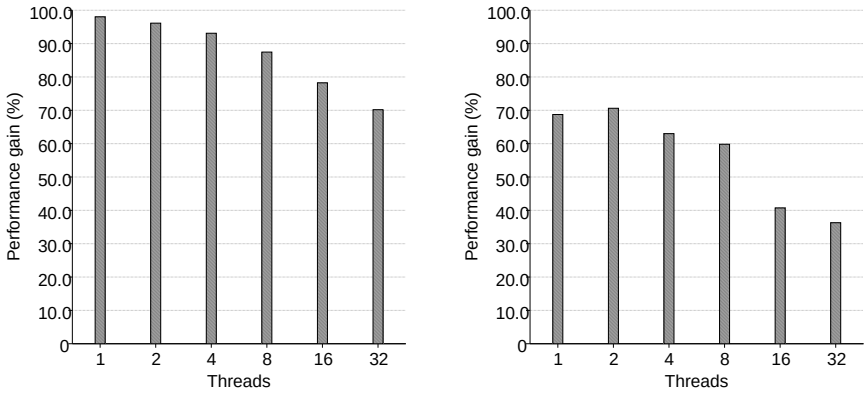
1. A first session tunes a sparse matrix into the CSX format and subsequently stores it in a file. In this session, we time the `spx_mat_tune()` routine. Note be taken that this doesn't include the cost of loading the matrix from either CSR or MMF.
2. Since the file rests in the file-system cache after the first session, we hint the OS, through a call to `posix_fadvise()`, to free the associated cached pages. This better simulates potential use of this feature, where the matrix is freshly loaded from the disk.
3. A second session restores the matrix. In this session, we time the `spx_mat_restore()` routine.

This scenario was run for both the full and sampled versions of CSX's preprocessing phase. Figures 4.1 and 4.2 show the average performance gain achieved in both cases in Gainestown and Sandy Bridge-EP respectively. In both cases, we notice that the benefit margin tightens when the number of threads grows, but this comes to no surprise since the `spx_mat_restore()` routine is a serial operation, (consisting mainly of I/O operations, data allocations and runtime code generation), while the tuning phase is performed in parallel and, thus, benefits from a multithreaded configuration. For com-



(a) Full preprocessing **(b)** Sampled preprocessing

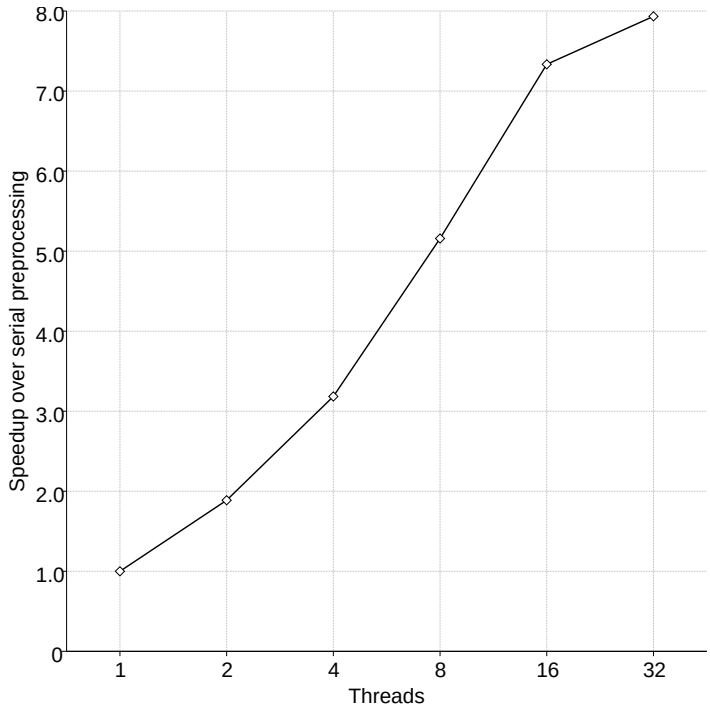
Figure 4.1: Performance gains from using a disk-cached CSX matrix in Gainestown.



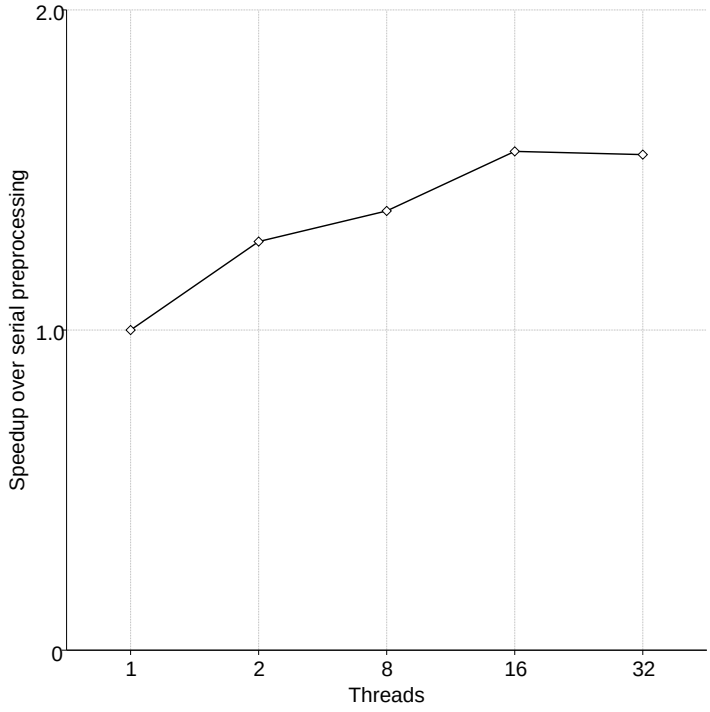
(a) Full preprocessing **(b)** Sampled preprocessing

Figure 4.2: Performance gains from using a disk-cached CSX matrix in Sandy Bridge-EP.

pleteness, the scalability of the preprocessing phase in Sandy Bridge-EP is depicted in Figure 4.3. Nonetheless, a minimum 36% performance benefit is guaranteed against the sampled preprocessing and a 70% against the full preprocessing in Sandy Bridge-EP, while in Gainestown the corresponding values climb up to 46% and 87%, pinpointing the importance of this feature when working on a specific matrix.



(a) Full preprocessing



(b) Optimized preprocessing

Figure 4.3: Scalability of the preprocessing phase of CSX in Sandy Bridge-EP.

4.3 SparseX versus other high performance libraries

In order to establish our library's high performance on multicore architectures, we decided to carry out a thorough comparison to other popular libraries that provide parallel implementations of the SpMV kernel, namely the widely-used commercial Math Kernel Library (MKL) by Intel® [Intel® Coorporation, 2013a] and the open-source parallel implementation of the Optimized Sparse Kernel Interface (pOSKI) [Vuduc et al., 2005; Ankit, 2008], which has been developed by the Berkeley Benchmarking and Optimization (BeBOP) group.

4.3.1 Intel® MKL

The Intel® Math Kernel Library (Intel® MKL) enables the improvement of performance of scientific, engineering, and financial software that solves large computational problems. It provides a set of linear algebra routines, fast Fourier transforms, as well as vectorized math and random number generation functions, all optimized for the latest Intel® processors, including processors with multiple cores, such as those in our experimental platforms. It fully implements and extends the BLAS interface, providing, among others, routines for running the SpMV kernel using a variety of sparse matrix storage formats, namely the COO, CSR, CSC and BCSR formats. When deciding on which version of the kernel to benchmark, the BCSR variation was rejected since it only supports $m \times k$ block sparse matrices, where m and k are explicit, thus limiting its use in our diverse matrix suite. It would also require some kind of auto-tuning phase, in order to detect the optimal block dimensions, followed by a conversion of the input matrix into the BCSR format. Since the COO and CSC formats are generally less efficient, the CSR version of the kernel was finally selected, namely the `mkl_dcsrmv()` routine. The library actually supports a 3-array variation of the CSR format (3 refers to the number of indexing structures), where the usual `rowptr` and `colind` arrays are backed by a third array that contains pointers to the end of each row. The interface also provides a similar routine for symmetric matrices, namely `mkl_cspblas_dcsrsymv()`, that was also tested, but exhibited much lower performance in comparison to the non-symmetric routine and, thus, its experimental results will not be provided. Listing 4.2 provides part of the source code employed to execute the SpMV kernel using the aforementioned CSR variant (non-symmetric version).

Since Intel® MKL doesn't explicitly support NUMA platforms, in order to attain optimal performance with this library, we made use of the *numactl* utility of Linux, that runs processes with a specific NUMA scheduling or mem-

ory placement policy. More precisely, we employed the `--interleave=all` option to set a memory interleave policy that will force the application to allocate memory using round robin on all nodes in the current cpuset. This is also suggested in many articles of the Intel® Developer Zone as a way to improve MKL's performance on NUMA-based systems [Henry, 2012]. Finally, thread affinity in MKL is internally set to the optimal configuration by default (it tries to use what it considers to be the best number of threads, up to maximum number specified by the user [Intel® Coorporation, 2013b]), and, therefore, it was not explicitly controlled through environment variables. Only the number of threads was provided through the `OMP_NUM_THREADS` variable.

Listing 4.2: Sample of the source code used to perform a doubly nested loop of SpMV operations with Intel® MKL.

```
1  // ...

3  MKL_INT *pointerB, *pointerE;
4  char    transa;
5  char    matdescra[6];

7  transa = 'n';
8  matdescra[0] = 'g';
9  matdescra[1] = '-';
10 matdescra[2] = '-';
11 matdescra[3] = 'c';

13 pointerB = (MKL_INT *) malloc(sizeof(MKL_INT) * nrows);
14 pointerE = (MKL_INT *) malloc(sizeof(MKL_INT) * nrows);
15 for (int i = 0; i < nrows; i++) {
16     pointerB[i] = rowptr[i];
17     pointerE[i] = rowptr[i+1];
18 }

20 mkl_set_num_threads(NR_THREADS);

22 // SpMV doubly nested loop
23 for (size_t i = 0; i < OUTER_LOOPS; i++) {
24     for (size_t j = 0; j < LOOPS; j++) {
25         mkl_dcsrmv(&transa, &nrows, &ncols, &ALPHA,
26                    matdescra, values, colind, pointerB,
27                    pointerE, x, &BETA, y);
28     }
29 }

31 // ...
```

4.3.2 BeBOP pOSKI

The parallel Optimized Sparse Kernel Interface (pOSKI) library is a parallel autotuner for SpMV, built on top of the highly-praised OSKI package. Its autotuning capabilities rely on extensive offline benchmarking of different kernel implementations and a number of heuristics that dynamically select the best storage format and implementation for a specific matrix at runtime. Supported sparse matrix storage formats include CSR, BCSR and BCSR-variants. During the installation process, pOSKI identifies the fastest implementations based on empirical search and creates optimized routines for the target machine's hardware. These static kernels become the defaults that are called when runtime tuning is not used. To use the most efficient kernel, however, the matrix storage format may need to be reorganized. This can be achieved at runtime if the user explicitly asks for the matrix to be tuned for a specific kernel by selecting either the moderate or aggressive tuning option.

On all of our test platforms, the library was built with explicit compiler flags, since the auto-detect code of the library is currently out-of-date. More specifically, we used the `-m64` flag, to generate code for the x86_64 architecture, the `-march=native` flag, which enables all instruction subsets supported by the local machine, and the `-O3` optimization flag. On our NUMA platforms, we also set the `--with-numa` installation option to enable NUMA-awareness. Finally, in order to better evaluate pOSKI's performance, both offline and runtime tuning capabilities were enforced. Offline tuning was enabled during the installation process with the `--with-tune` option, while full runtime tuning was enforced through the `ALWAYS_TUNE_AGGRESSIVELY` option of the tuning routine. A sample of the source code used when benchmarking the pOSKI library is given in Listing 4.3.

Listing 4.3: Sample of the source code used to perform a doubly nested loop of SpMV operations with pOSKI.

```
1  // ...

3  // Matrix loading
4  poski_threadarg_t *poski_thread = poski_InitThreads();
5  poski_ThreadHints(poski_thread, NULL, POSKI_THREADPOOL,
6                    NR_THREADS);
7  poski_partitionarg_t *partitionMat =
       poski_partitionMatHints(OneD, NR_THREADS,
       KERNEL_MatMult, OP_NORMAL);
8  poski_mat_t A_tunable = poski_CreateMatCSR(...);

10 // Vector loading
11 poski_partitionvec_t *partitionVecX =
       poski_PartitionVecHints(...);
```

```
12  poski_vec_t x_view = poski_CreateVec(..., partitionVecX);
13  poski_partitionvec_t *partitionVecY =
        poski_PartitionVecHints(...);
14  poski_vec_t y_view = poski_CreateVec(..., partitionVecY);

16  // Matrix tuning
17  poski_TuneHint_MatMult(A_tunable, OP_NORMAL, ALPHA, x_view,
         BETA, y_view, ALWAYS_TUNE_AGGRESSIVELY);
18  poski_TuneMat(A_tunable);

20  // SpMV doubly nested loop
21  for (size_t i = 0; i < OUTER_LOOPS; i++) {
22      for (size_t j = 0; j < LOOPS; j++) {
23          poski_MatMult(A_tunable, OP_NORMAL, ALPHA, x_view,
24                        BETA, y_view);
25      }
26  }

28  // ...
```

4.3.3 SparseX vs MKL vs pOSKI performance analysis

If a library wishes to be competitive in terms of SpMV performance, it should limit resource sharing among running threads. Since this is the default policy for SparseX, we will only look upon the "share-nothing" results for comparison purposes. Measurements of the "share-all" core-filling policy are only provided to gain better insight into the performance of the SpMV kernel on our experimental platforms.

For all libraries, we measure the total time it takes to calculate 128 consecutive iterations of the SpMV kernel ($y \leftarrow alpha \cdot A \cdot x + beta \cdot y$) with randomly created x and y vectors, a randomly generated *alpha* and $beta = 1$. These settings better simulate use of the kernel in the CG iterative method (see Section 1.2). Each reported result is the median value from 5 runs with a warm cache.

SMP architectures

In Dunnington, our four-way SMP architecture, the SpMV kernel is expected to fully expose its memory-intensive nature by stressing the common bus used by all sockets to communicate with the main memory. Indeed, in Figure 4.4 we observe that under a "share-all" core filling policy, in all multithreaded configurations that use a single socket (i.e., 2- and 6-threaded), memory contention prohibits performance scaling. However, whenever an additional socket comes into play (e.g., in the 12- and 24-threaded configura-

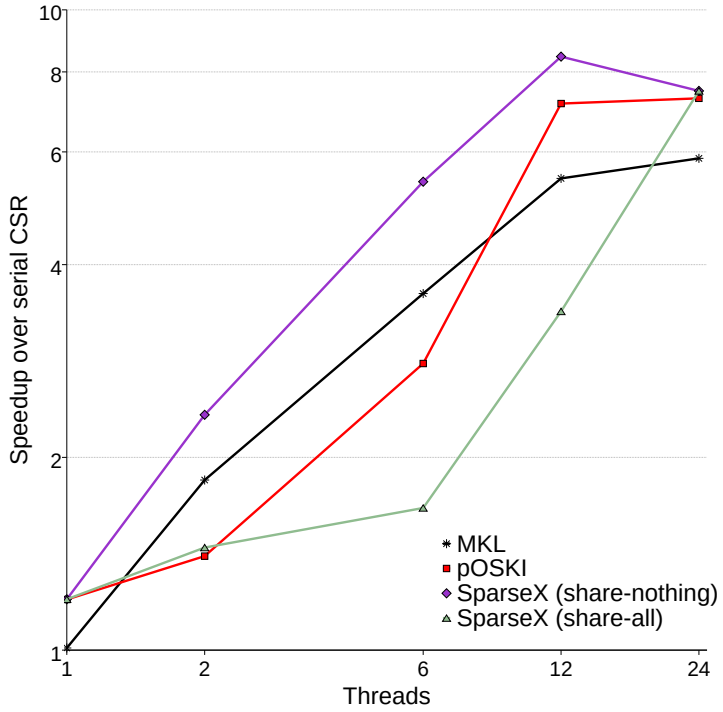


Figure 4.4: Speedup over the serial naive CSR implementation of SpMV in Dunnington. For SparseX, the average speedup is depicted using both core-filling policies.

tion), a major performance improvement is achieved, confirming the kernel's "craving" for memory bandwidth. The same conclusion can be drawn by simply comparing the two core-filling policies in SparseX. When moving, for example, from 2 to 6 threads, use of a second socket in the "share-nothing" policy alleviates the common bus, leading to a 2.4× speedup (with reference to the 2-threaded configuration) over a mere 1.18× in the "share-all" case.

As expected from previous studies of the CSX format [Karakasis et al., 2013; Gkountouvas et al., 2013], the large compression capabilities of CSX give our library a unique advantage in SMP architectures, where the memory bottleneck is more prominent. As shown in Figures 4.5 and 4.6, SparseX outperforms MKL and pOSKI in the majority of matrices. It reaches a 17.5% average performance improvement over pOSKI and 53.9% over MKL for the 12-threaded configuration. It is important to note here that the slowdown in the 24-threaded configuration is due to the non-optimal thread management currently implemented in SparseX. Specifically, in every SpMV iteration our main thread internally creates the corresponding number of threads using `pthread_create()` and waits for each one of them with a call to

`pthread_join()`. This means that we spawn and destroy threads in every iteration. This, of course, is an expensive process in terms of time, which turns out to be a performance hindering factor as the number of threads grows, particularly in small matrices with low execution times (e.g., *xenon2*). In order to overcome this problem, we are currently redesigning thread management in SparseX to make use of a *thread pool*, which eliminates the need to continuously create and destroy threads. This is actually, the threading model employed by pOSKI. For the aforementioned reason, we will examine the 12-threaded configuration in Dunnington, which is more representative of CSX's actual performance capabilities.

Examining Figure 4.5, we notice that SparseX achieves exceptional performance improvements in matrices dominated by diagonal patterns (e.g., *cage13, atmosmodj, rajat31, ktt_power*), which cannot be efficiently exploited by the storage formats employed by our rival libraries. pOSKI becomes competitive mainly in block-dominated matrices (*xenon2, F1, consph, bmw7st_1, m_t1, bmwcra_1, crankseg_2, af_5_k101, inline_1, ldoor, bones10*), which favor the BCSR format, but manages to surpass SparseX's performance only in *xenon2*. This is the result of the improved computational characteristics of the BCSR implementation of pOSKI, which applies vectorization at the block-level. This optimization pays off in case of *xenon2*, due to the fact that this matrix fits in the aggregate cache of Dunnington and the corresponding source vectors fit in the L2 cache. Finally, in most low-performing matrices, mainly characterized by an irregular nonzero element structure and very short rows (e.g., *Hamrle3, Freescale1, parabolic_fem, G3_circuit, thermal2*), all libraries stay close. In these matrices, SpMV's performance is hindered by a number of factors not related to the memory bandwidth saturation, such as irregular accesses in the input vector, increased loop overheads and load imbalance.

NUMA architectures

In NUMA architectures, where the available memory bandwidth is considerably higher, the performance landscape changes for the majority of matrices, but the fact remains that our library stays ahead of both MKL and pOSKI in NUMA platforms as well. A first glance at Figures 4.7 and 4.8 indicates that pOSKI's NUMA-aware optimizations do not pay off at all on these platforms. Performance ceases to improve for more than 4 threads in Gainestown and Sandy Bridge-EP and 6 threads in Westmere-EP, while exhibiting a minor slowdown in some cases. MKL, on the other hand, is more performant, even though it isn't explicitly tuned for NUMA platforms. It even manages to slightly improve its performance in the 24- and 64-threaded (HT-enabled)

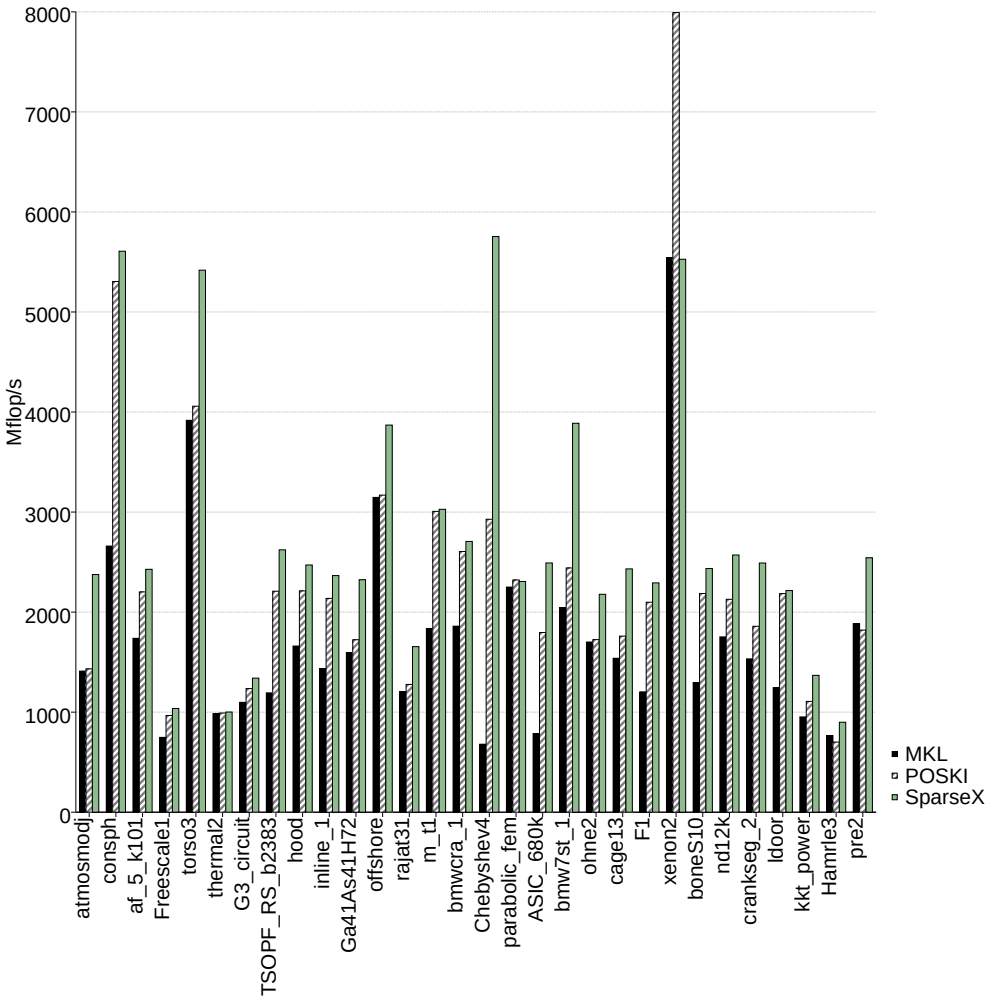


Figure 4.5: Per-matrix performance in Dunnington using 12 threads for SparseX, MKLand pOSKI. SparseX outperforms both MKL and pOSKI in the majority of the selected matrices. This configuration is more representative of CSX's performance capabilities (see text).

configurations in Westmere-EP and Sandy Bridge-EP, which comes to a surprise, since SMT technology is not suited for the SpMV kernel. SMT, in general, is usually effective when each thread is performing different types of operations and there are under-utilized resources on the processor, which is definitely not the case in SpMV. Therefore, even though it seems that MKL benefits from HT, this is not actually true. If the requested number of threads exceeds the number of physical cores, MKL will dynamically scale down the number of threads to the number of physical cores. Thus, in Sandy Bridge-EP,

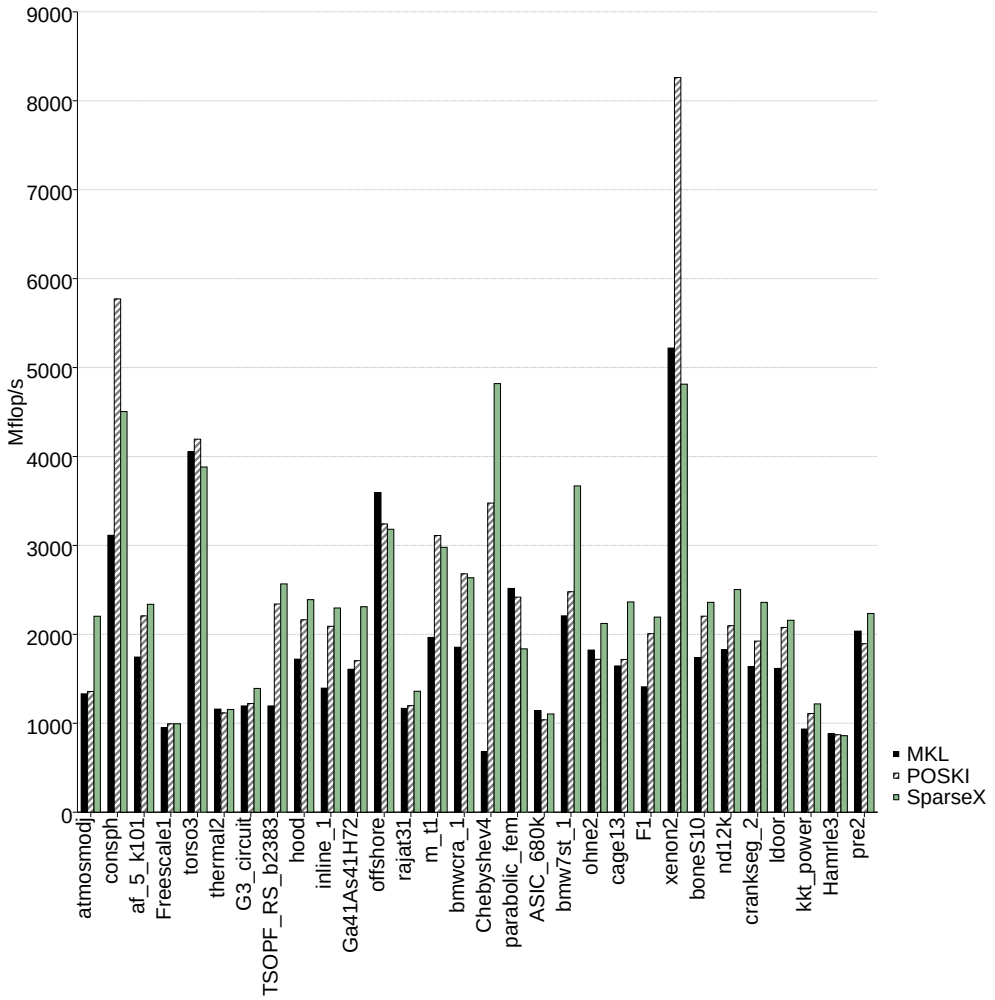
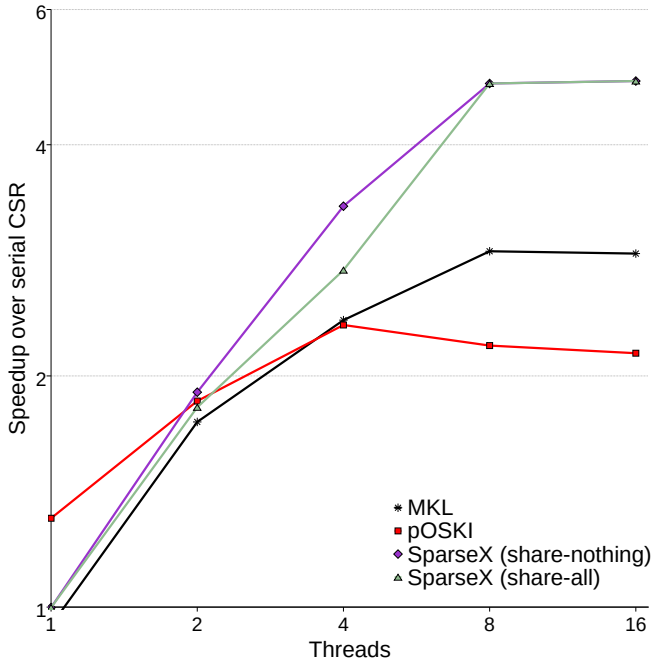


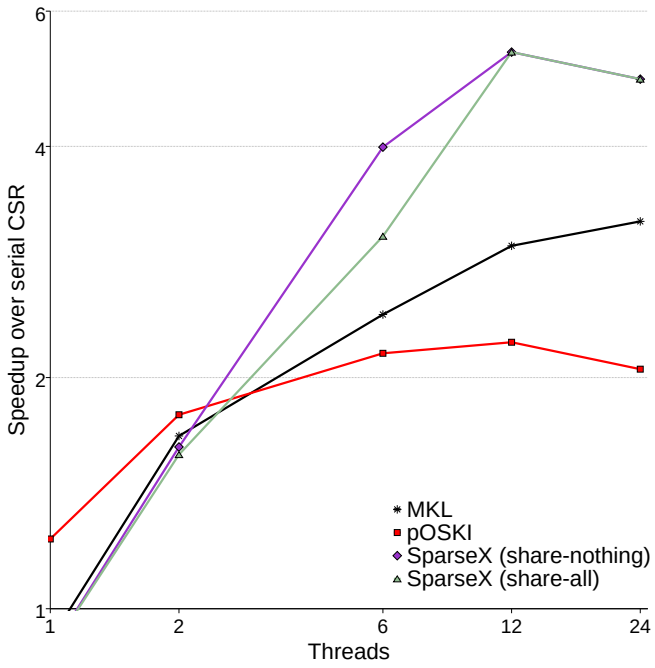
Figure 4.6: Per-matrix performance in Dunnington using 24 threads for SparseX, MKL and pOSKI. SparseX outperforms both MKL and pOSKI in 19 out of the 30 selected matrices, while exhibiting poor performance only in 4 matrices, namely *consph, offshore, parabolic_fem* and *xenon2*.

for example, MKL's 64-threaded performance should actually be compared to the 32-threaded performance of SparseX and pOSKI. Nonetheless, out library holds a major average performance lead of 65% over MKL and 114% over pOSKI in the 8-threaded configuration in Gainestown, 76% and 140% respectively in the 12-threaded configuration in Westmere-EP and 30% and 154% in the 32-threaded configuration in Sandy Bridge-EP.

Observing the per-matrix performance results in Figure 4.9, in comparison to those of 4.5, one should notice an important change in the set of high-

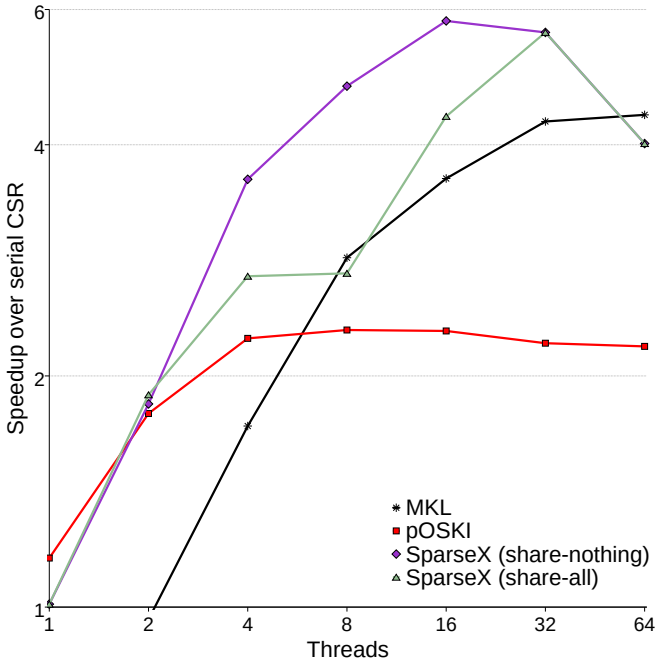


(a) Gainestown



(b) Westmere-EP

Figure 4.7: SpMV kernel speedup in Gainestown and Westmere-EP. For SparseX, the average speedup is depicted using both core-filling policies in the CSX format. The 16- and 24-threaded configurations in Gainestown and Westmere-EP respectively make use of Hyper-Threading, which does not benefit the SpMV kernel (see text).



(a) Sandy Bridge-EP

Figure 4.8: SpMV kernel speedup Sandy Bridge-EP. For SparseX, the average speedup is depicted using both core-filling policies in the CSX format. Slowdown in the 32- and 64-threaded configuration is mainly due to the non-optimal thread management model employed by SparseX (see text for more details), while the 64-threaded configuration also makes use of Hyper-Threading, which does not benefit the SpMV kernel (again, see text). MKL does not actually make use of the Hyper-Threading technology.

performing matrices. All large matrices of our test suite, including *boneS10, ldoor, af_5_k101, F1, rajat31, Freescale1* and *TSOPF_RS_b2383*, have significantly improved there performance due to the ample memory bandwidth (almost three times faster in case of Sandy Bridge-EP) available in our NUMA platforms.

4.3.4 SparseX on symmetric matrices

Symmetric matrices comprise a large subcategory of sparse matrices that arise in real-life applications. These matrices introduce a great opportunity when it comes to reducing the total matrix size, since only the lower (or upper) half of the matrix and its main diagonal suffice to describe the whole matrix. Thus, when the matrix is symmetric, the user can benefit from the

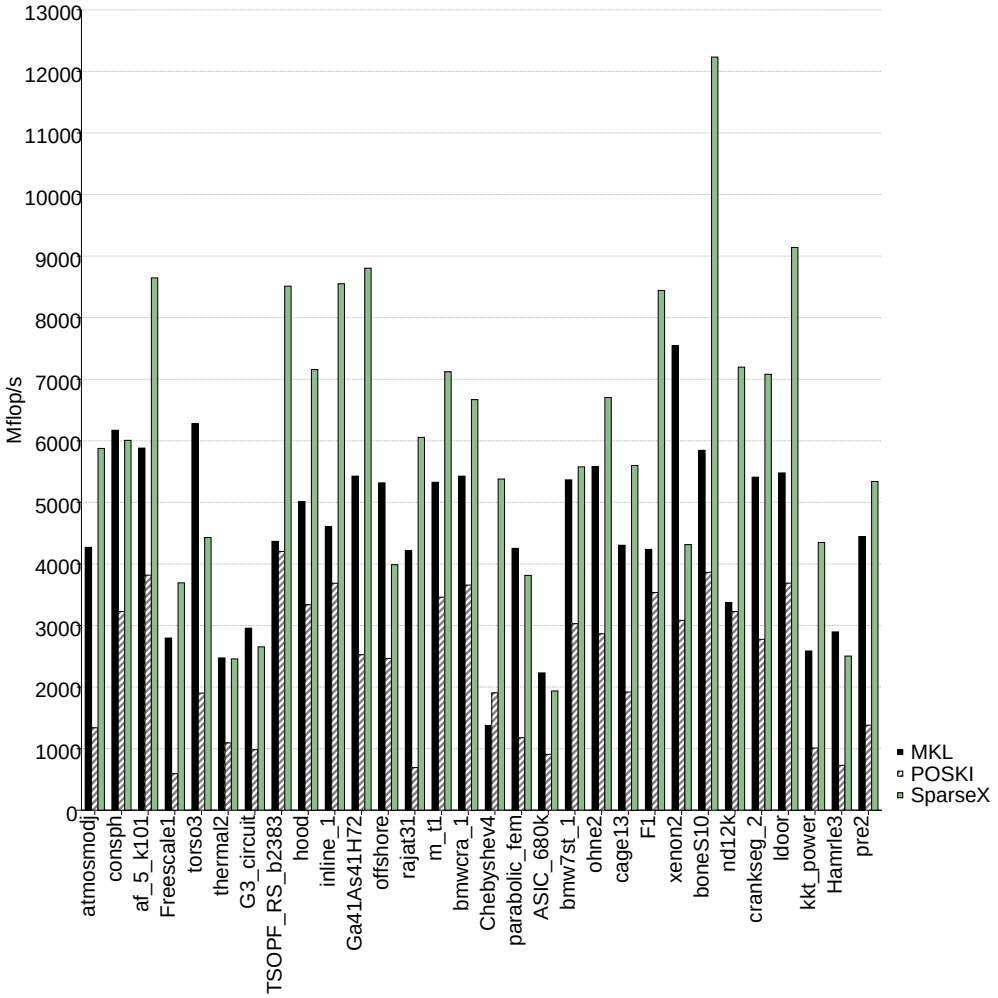


Figure 4.9: The SpMV performance in Sandy Bridge-EP for every sparse matrix in our suite using 16 threads in a NUMA-aware configuration. For SparseX performance is depicted using a "share-nothing" core-filling policy. SparseX manages to surpass both MKL and pOSKI in the majority of matrices, with astonishing performance gains in most high-performing matrices.

symmetric variant of the CSX format by setting the appropriate tuning option (`spx.matrix.symmetric`). This format reduces the overall matrix size almost to half in comparison to the non-symmetric version of CSX, thus attaining even higher performance. As mentioned earlier on, the symmetric variant of MKL performed very poorly and is, therefore, not presented in our experimental results. As for pOSKI, despite the fact that symmetry could be

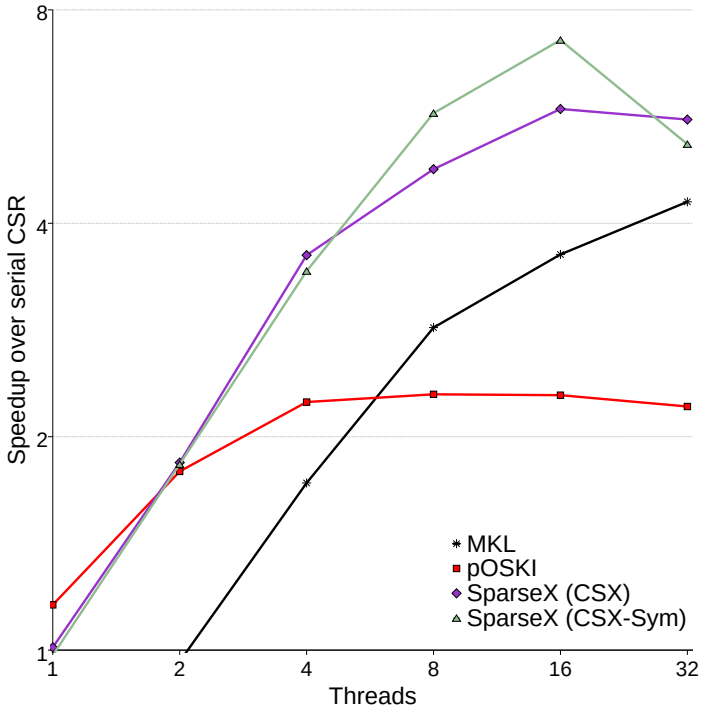


Figure 4.10: SpMV kernel speedup in Sandy Bridge-EP on the subset of symmetric matrices of our test suite. For SparseX, the average speedup is depicted using both formats, i.e., CSX and CSX-Sym.

provided as an input matrix property in OSKI, this option is not available in the parallel autotuner.

Figure 4.10 shows the average speedup over the symmetric matrices of our suite. SparseX achieves a 99% performance improvement over MKL and a 213% over POSKI in the 16-threaded configuration. Again, in 32 threads, the non-optimal thread management of our library, in conjunction with the additional synchronization point required before the reduction phase of the symmetric SpMV kernel, burdens total execution time in many matrices. Nonetheless, SparseX still manages to stay ahead of MKL and pOSKI.

For high-bandwidth matrices, reordering may yield considerable performance improvements, as depicted in Figure 4.11 and explained in Section 3.2.1. It comes with a non-negligible cost, if performed entirely at runtime, as it involves finding a suitable permutation, applying it to the input vectors of the SpMV kernel and, finally, applying the inverse permutation to the output vector in order to receive the correct result [Liu and Sherman, 1976]. However, this cost can be amortized if the user expects a large number of SpMV operations to be performed, which is usually the case in iter-

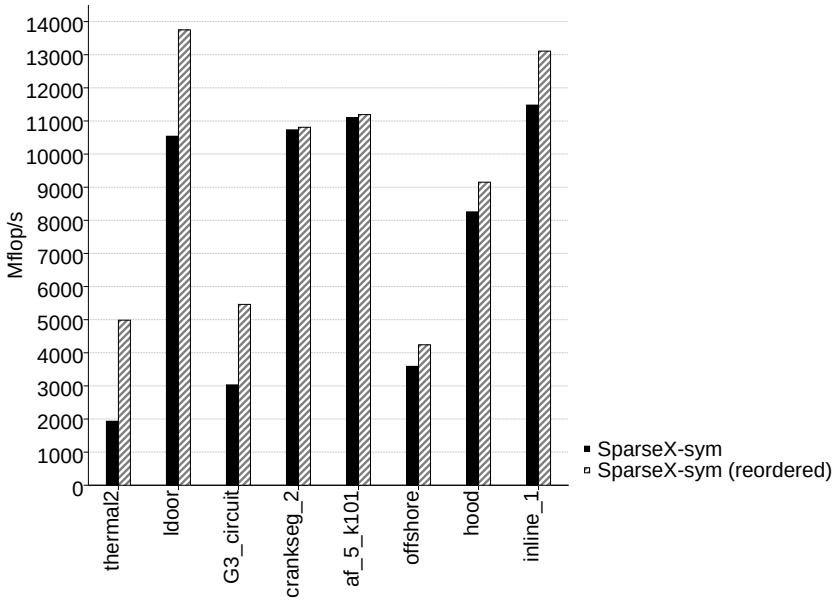


Figure 4.11: The effect of reordering on the SpMV performance using the symmetric variant of CSX in a 16-threaded configuration in Sandy Bridge-EP. Experimental results are provided only on the subset of symmetric matrices in our test suite that benefit from applying the Reverse Cuthill McKee bandwidth-reduction algorithm, which is provided by our API.

ative solvers. Usually though, the input matrix is already reordered, thus, reordering at runtime only involves applying the permutations, making it much more lightweight.

4.3.5 Performance issues

Thread management

Even though SparseX outperforms both MKL and pOSKI in all multithreaded configurations, it exhibits a noticeable performance degradation on Dunnington and Sandy Bridge-EP, when using 24 and 32 cores respectively under a share-nothing core-filling policy (approximately 10% on average of all matrices on Dunnington and 3.4% on Sandy Bridge-EP). Our measurements indicate that this is due to the current thread management model employed by SparseX, as mentioned earlier in our performance analysis. We believe that creating and destroying threads in every SpMV iteration incurs a significant overhead for a large number of threads, which tends to dominate total execution time in small high-performing matrices. As a proof of point, we performed a benchmark that simulates use of a *thread pool*, which we consider to

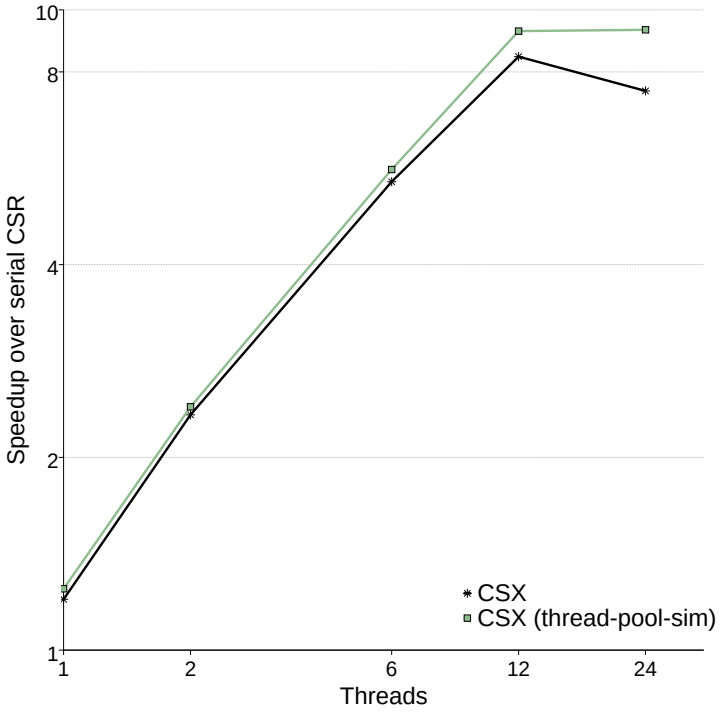


Figure 4.12: SpMV kernel speedup in Dunnington as a result of the thread pool simulation benchmark.

be a viable solution to this problem. In this benchmark, threads are created once at library initialization (instead of every SpMV iteration), while synchronization is performed at the end of each SpMV iteration, with the use of explicit barriers (instead of a thread join). Figure 4.12 shows the results of this benchmark on Dunnington. For more than 6 threads, CSX slightly improves its performance, while it no longer encounters a performance slowdown at the 24-threaded configuration, where the system is completely saturated, as expected.

Load imbalance

In matrices with a very irregular nonzero element structure, our static matrix partitioning scheme (see Section 2.4) cannot guarantee a fair distribution of the work among the threads. This is due to the fact that our scheme does not take into account the sparsity pattern within a thread partition. For example, two partitions may have the same number of nonzero elements, but different access patterns in the input vector; SpMV will be rather slow in the partition with the most irregular pattern, hindering the entire multithreaded

execution. The most typical example of this situation is the *Hamrle3* matrix (see Appendix A). A closer inspection at *Hamrle3*'s structure reveals that the upper half of the matrix is quite regular, while the rest is very irregular with large and variable distances between the nonzero elements of a row. In another case, some parts of the matrix may have extremely short rows, while others may have rows with a row length close to the matrix dimension. Threads assigned to the parts with very short rows are slower because they are burdened by more loop overheads. A typical matrix of such a behavior is the *ASIC_680k*. It is worth mentioning that in a 2-threaded configuration *Hamrle3* exhibits a 127% load imbalance, which climbs up to 368% using 32 threads, while *ASIC_680k* has a 20% and 388% load imbalance respectively. As the number of participating threads increases, it is more likely to have some partitions with very different row lengths or memory access patterns from others, amplifying the effect of load imbalance on performance.

5

Conclusions

In this thesis, we have thoroughly presented SparseX, a high performance open-source library for performing efficient Sparse Matrix-Vector Multiplications on modern multicore architectures. We provided some implementations details on the core library and gave a detailed overview of the user API, accompanied by a number of examples that exhibit its ease of use. Next, we evaluated the disk-cached CSX matrix feature, which significantly reduces the preprocessing cost of CSX when the same (or a structurally equivalent) matrix needs to be used in multiple sessions. Finally, we performed a comparison of SparseX to other popular libraries that provide the same functionality on a wide variety of hardware platforms, establishing our library's potential for improving the sparse linear algebra software stack. In the following section we provide some future research prospects and directions.

5.1 Future work

We are currently working on modifying thread management in SparseX. As a first step, we are planning to evaluate the use of a thread pool, which we believe to be more appropriate for executing a large number of SpMV iterations, especially as the number of threads grows. Our current model spawns and destroys threads in every SpMV iteration. This process can become expensive for a large number of threads and dominate total execution time in small high-performing matrices. We are also considering implementing a custom static memory barrier, that will make use of spinlocks, as a viable approach to reducing synchronization costs. With spinlocks, we can avoid wasting time on constantly putting threads, that have completed their computations, to sleep and waking them up again. Instead, threads will remain in a busy waiting state for a short time period, leading, eventually, to a higher processing throughput.

Another issue, that becomes significant to SpMV performance as the number of cores increases, is the matrix partitioning method employed. Al-

though our static partitioning scheme is widely used in many SpMV implementations on multicore processors, and leads to good performance for most matrices, for a large number of cores (e.g., in a many-core system) it is more likely to create partitions with very different row lengths and memory access patterns, resulting in significant load imbalance among the threads. Thus, if we ever consider porting CSX to many-core architectures, implementation of an adaptive load balancing technique will have to be considered.

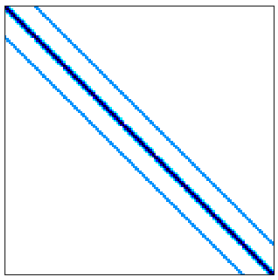
By alleviating the memory pressure, CSX leaves more space for optimizations targeting the computational part of the kernel, such as vectorization. Even though the effect of vectorization diminishes as the problem becomes more memory bound (i.e., completely saturate the system), it can achieve significant speedups when the working set of the matrix fits in the aggregate cache of the system [Karakasis et al., 2009]. We also expect vectorization to boost performance in NUMA platforms, which mitigate the memory bottleneck and further expose computations. To this end, we are looking into Intel®'s Advanced Vector Extensions (AVX), that provide 256-bit integer SIMD extensions that claim to accelerate computation across integer and floating-point domains using 256-bit vector registers.

Finally, in order for SparseX to be of essential practical value to the HPC community, it is important to intergate it into higher-level sparse solver libraries, such as PETSc or the Trilinos project [Balay et al., 2013; Heroux et al., 2005]. Integration with such systems has the unique advantage of a large potential user base, that would contribute, in our case, to the better dissemination of the CSX format.

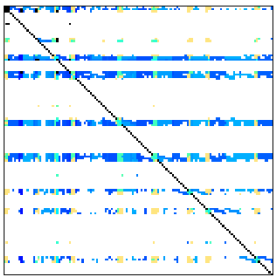
A

Matrix suite

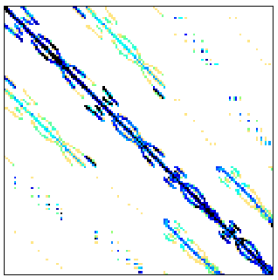
Our experimental matrix suite includes 30 matrices from the University of Florida Sparse Matrix Collection [Davis and Hu, 2011]. This collection contains thousands of sparse matrices that arise in real applications from a variety of scientific domains. It has become the standard source for matrices in the numerical linear algebra community for the performance evaluation of sparse matrix algorithms. We have selected matrices with an underlying 2D or 3D geometry coming from computational fluid dynamics, electromagnetics, thermodynamics, materials, structural mechanics and model reduction, as well as matrices without such a geometry from electrical circuit simulation, chemical process simulation, power networks and graphs.



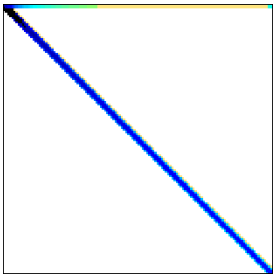
(1) *xenon2*



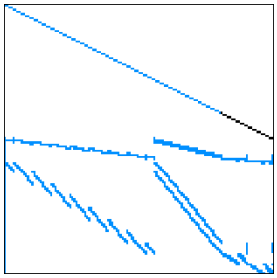
(2) *ASIC_680k*



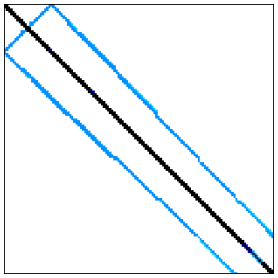
(3) *torso3*



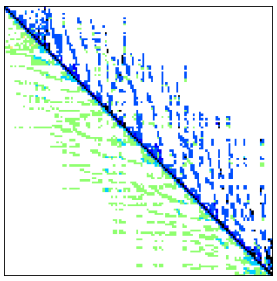
(4) *Chebyshev4*



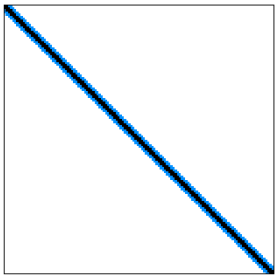
(5) *Hamrle3*



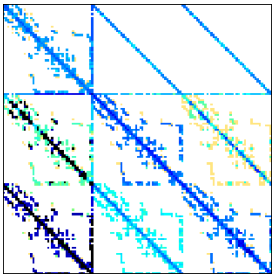
(6) *pre2*



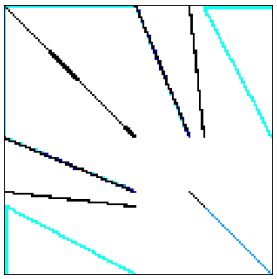
(7) *cage13*



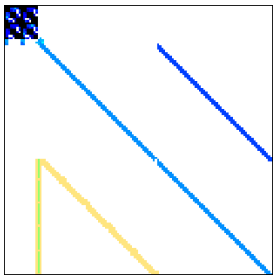
(8) *atmosmodj*



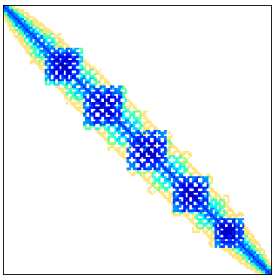
(9) *ohne2*



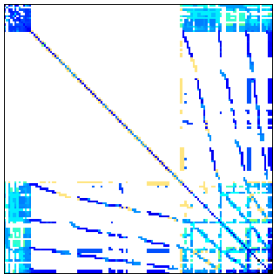
(10) *kkt_power*



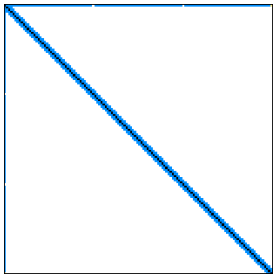
(11) *TSOPF_RS_b2383*



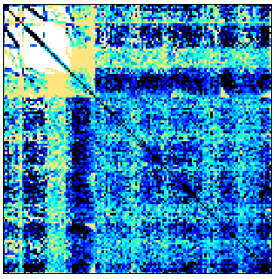
(12) *Ga41As41H72*



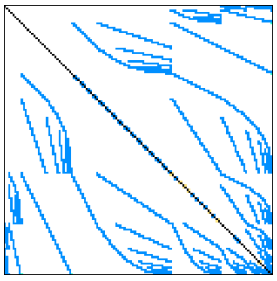
(13) *Freescale1*



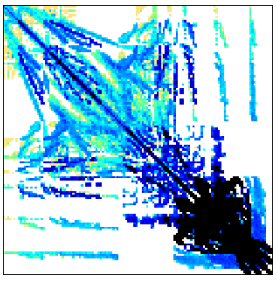
(14) *rajat31*



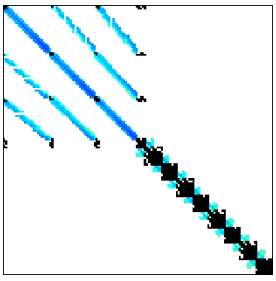
(15) *F1*



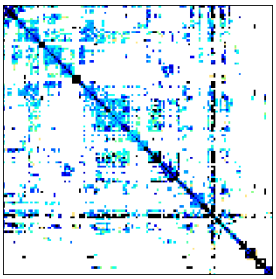
(16) *parabolic_fem*



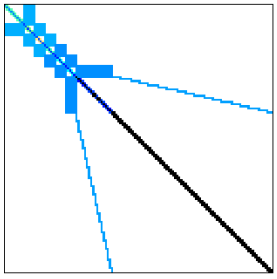
(17) *offshore*



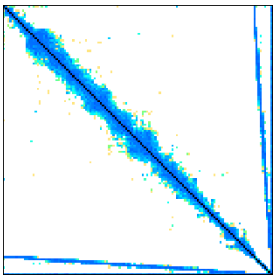
(18) *consph*



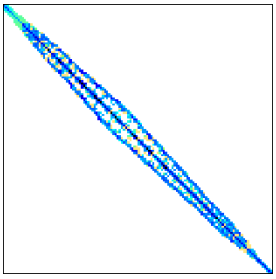
(19) *bmw7st_1*



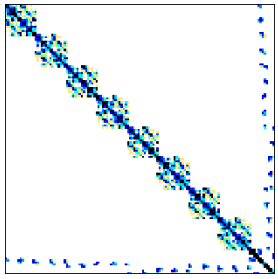
(20) *G3_circuit*



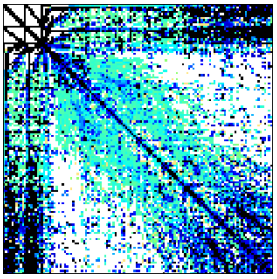
(21) *thermal2*



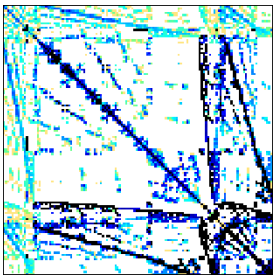
(22) *m_t1*



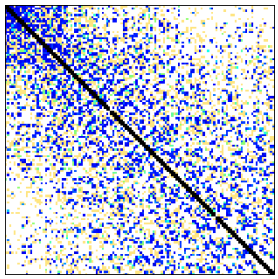
(23) *bmwcra_1*



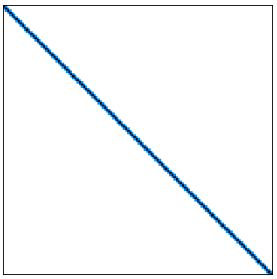
(24) *hood*



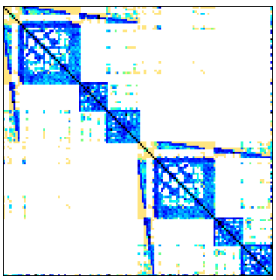
(25) *crankseg_2*



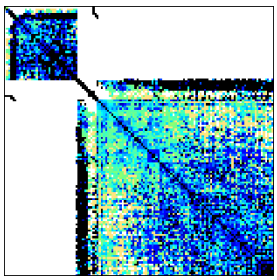
(26) *nd12k*



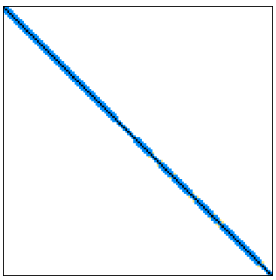
(27) *af_5_k101*



(28) *inline_1*



(29) *ldoor*



(30) *boneS10*

B

C bindings reference

Generated by Doxygen 1.8.6

A.1 Available routines

- void spx_init ()
- void spx_finalize ()
- spx_input_t * spx_input_load_csr (spx_index_t *rowptr, spx_index_t *colind, spx_value_t *values, spx_index_t nr_rows, spx_index_t nr_cols,...)
- spx_input_t * spx_input_load_mmf (const char *filename)
- spx_error_t spx_input_destroy (spx_input_t *input)
- spx_matrix_t * spx_mat_tune (spx_input_t *input,...)
- spx_error_t spx_mat_get_entry (const spx_matrix_t *A, spx_index_t row, spx_index_t column, spx_value_t *value,...)
- spx_error_t spx_mat_set_entry (spx_matrix_t *A, spx_index_t row, spx_index_t column, spx_value_t value,...)
- spx_error_t spx_mat_save (const spx_matrix_t *A, const char *filename)
- spx_matrix_t * spx_mat_restore (const char *filename)
- spx_index_t spx_mat_get_nrows (const spx_matrix_t *A)
- spx_index_t spx_mat_get_ncols (const spx_matrix_t *A)
- spx_index_t spx_mat_get_nnz (const spx_matrix_t *A)
- spx_partition_t * spx_mat_get_partition (spx_matrix_t *A)
- spx_perm_t * spx_mat_get_perm (const spx_matrix_t *A)
- spx_error_t spx_matvec_mult (spx_scalar_t alpha, const spx_matrix_t *A, spx_vector_t *x, spx_vector_t *y)
- spx_error_t spx_matvec_kernel (spx_scalar_t alpha, const spx_matrix_t *A, spx_vector_t *x, spx_scalar_t beta, spx_vector_t *y)
- spx_error_t spx_matvec_kernel_csr (spx_matrix_t *A, spx_index_t nr_rows, spx_index_t nr_cols, spx_index_t *rowptr, spx_index_t *colind, spx_value_t *values, spx_scalar_t alpha, spx_vector_t *x, spx_scalar_t beta, spx_vector_t *y)
- spx_error_t spx_mat_destroy (spx_matrix_t *A)
- spx_partition_t * spx_partition_csr (spx_index_t *rowptr, spx_index_t nr_rows, unsigned int nr_threads)
- spx_error_t spx_partition_destroy (spx_partition_t *p)
- void spx_option_set (const char *option, const char *string)
- void spx_options_set_from_env ()
- spx_vector_t * spx_vec_create (unsigned long size, spx_partition_t *p)
- spx_vector_t * spx_vec_create_from_buff (spx_value_t *buff, unsigned long size, spx_partition_t *p, spx_copymode_t mode)
- spx_vector_t * spx_vec_create_random (unsigned long size, spx_partition_t *p)
- void spx_vec_init (spx_vector_t *v, spx_value_t val)

- void spx_vec_init_part (spx_vector_t *v, spx_value_t val, spx_index_t start, spx_index_t end)
- void spx_vec_init_rand_range (spx_vector_t *v, spx_value_t max, spx_value_t min)
- spx_error_t spx_vec_set_entry (spx_vector_t *v, spx_index_t idx, spx_value_t val)
- void spx_vec_scale (spx_vector_t *v1, spx_vector_t *v2, spx_scalar_t num)
- void spx_vec_scale_add (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_scalar_t num)
- void spx_vec_scale_add_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_scalar_t num, spx_index_t start, spx_index_t end)
- void spx_vec_add (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3)
- void spx_vec_add_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_index_t start, spx_index_t end)
- void spx_vec_sub (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3)
- void spx_vec_sub_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_index_t start, spx_index_t end)
- spx_value_t spx_vec_mul (const spx_vector_t *v1, const spx_vector_t *v2)
- spx_value_t spx_vec_mul_part (const spx_vector_t *v1, const spx_vector_t *v2, spx_index_t start, spx_index_t end)
- spx_error_t spx_vec_reorder (spx_vector_t *v, spx_perm_t *p)
- spx_error_t spx_vec_inv_reorder (spx_vector_t *v, spx_perm_t *p)
- void spx_vec_copy (const spx_vector_t *v1, spx_vector_t *v2)
- int spx_vec_compare (const spx_vector_t *v1, const spx_vector_t *v2)
- void spx_vec_print (const spx_vector_t *v)
- void spx_vec_destroy (spx_vector_t *v)
- void spx_timer_clear (spx_timer_t *t)
- void spx_timer_start (spx_timer_t *t)
- void spx_timer_pause (spx_timer_t *t)
- double spx_timer_get_secs (spx_timer_t *t)
- void spx_log_disable_all ()
- void spx_log_error_console ()
- void spx_log_warning_console ()
- void spx_log_info_console ()
- void spx_log_verbose_console ()
- void spx_log_debug_console ()
- void spx_log_error_file ()
- void spx_log_warning_file ()
- void spx_log_info_file ()
- void spx_log_verbose_file ()

- void spx_log_debug_file ()
- void spx_log_all_console ()
- void spx_log_all_file (const char *file)
- void spx_log_set_file (const char *file)
- spx_errhandler_t spx_err_get_handler ()
- void spx_err_set_handler (spx_errhandler_t new_handler)

A.2 Routine documentation

A.2.1 void spx_init ( )

Library initialization routine.

A.2.2 void spx_finalize ( )

Library shutdown routine.

A.2.3 spx_input_t* spx_input_load_csr (spx_index_t * *rowptr,* spx_index_t * *colind,* spx_value_t * *values,* spx_index_t *nr_rows,* spx_index_t *nr_cols,* ...)

Creates and returns a valid tunable matrix object from a Compressed Sparse Row (CSR) representation.

Parameters

in	*rowptr*	array *rowptr* of the CSR format.
in	*colind*	array *colind* of the CSR format.
in	*values*	array *values* of the CSR format.
in	*nr_rows*	number of rows of the matrix.
in	*nr_cols*	number of columns of the matrix.
in	*optional*	argument that specifies the indexing (either INDEXING_ZERO_BASED or INDEXING_ONE_BASED).

Returns

a handle to the input matrix or INVALID_INPUT in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments are invalid.
2. SPX_ERR_INPUT_MAT: the input data arrays do not correspond to a valid CSR representation.

A.2.4 spx_input_t* spx_input_load_mmf (const char * *filename*)

Creates and returns a valid tunable matrix object from a file in the Matrix Market File Format (MMF).

Parameters

in	*filename*	name of the file where the matrix is stored.

Returns

a handle to the input matrix or INVALID_INPUT in case an error occurs.

Possible error conditions

1. SPX_ERR_FILE: the file does not exist or an error occured while trying to read it (possibly not in valid MMF format).

A.2.5 spx_error_t spx_input_destroy (spx_input_t * *input*)

Releases any memory internally used by the sparse matrix input handle *input*.

Parameters

in	*input*	the input matrix handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the input matrix handle is invalid.

A.2.6 spx_matrix_t* spx_mat_tune (spx_input_t * *input, ...*)

Converts the input matrix into the CSX format by applying all the options previously set with the spx_option_set() routine. In case no options have been explicitly set, the default values are used (see Table 3.2 of the User's Guide).

Parameters

in	*input*	the input matrix handle.
in	*optional*	optional flag that indicates whether the matrix should be reordered by applying the Reverse Cuthill McKee algorithm (OP_REORDER).

Returns

a handle to the tuned matrix or INVALID_MAT in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the input matrix handle is invalid.
2. SPX_ERR_TUNED_MAT: conversion to the CSX format failed.

A.2.7 spx_error_t spx_mat_get_entry (const spx_matrix_t * *A,* spx_index_t *row,* spx_index_t *column,* spx_value_t * *value,* ...)

Returns the value of the corresponding nonzero element in (*row*, *column*), where *row* and *column* can be either zero- or one-based indexes. Default indexing is zero-based, but it can be overidden through the optional flag. If the element exists, its value is returned in *value*.

Parameters

in	*A*	the tuned matrix handle.
in	*row*	the a row of the element to be retrieved.
in	*column*	the column of the element to be retrieved.
out	*value*	the value of the element in (*row*, *column*).
in	*optional*	argument that specifies the indexing (either INDEXI↩ NG_ZERO_BASED or INDEXING_ONE_BASED).

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_OUT_OF_BOUNDS: the element is out of range.
3. SPX_ERR_ENTRY_NOT_FOUND: the element doesn't exist (i.e. is not nonzero).

A.2.8 spx_error_t spx_mat_set_entry (spx_matrix_t * *A,* spx_index_t *row,* spx_index_t *column,* spx_value_t *value,* ...)

Sets the value of the corresponding element in (*row*, *column*), where *row* and *column* can be either zero- or one-based indexes. Default indexing is zero-based, but it can be overidden through the optional flag.

Parameters

in	*A*	the tuned matrix handle.
in	*row*	the row of the element to be set.
in	*column*	the column of the element to be set.
in	*value*	the new value of the element in (*row*, *column*).
in	*optional*	argument that specifies the indexing (either INDEXI↩ NG_ZERO_BASED or INDEXING_ONE_BASED).

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_OUT_OF_BOUNDS: the element is out of range.
3. SPX_ERR_ENTRY_NOT_FOUND: the element doesn't exist (i.e. is not nonzero).

A.2.9 spx_error_t spx_mat_save (const spx_matrix_t * *A*, const char * *filename*)

Stores the matrix in the CSX format into a binary file.

Parameters

in	*A*	the tuned matrix handle.
in	*filename*	the name of the binary file where the matrix will be dumped.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

Possible warnings

1. SPX_WARN_CSXFILE: a filename hasn't been supplied so the default *csx_file.bin* will be used.

A.2.10 spx_matrix_t* spx_mat_restore (const char * *filename*)

Reconstructs the matrix in the CSX format from a binary file.

Parameters

in	*filename*	the name of the file where the matrix is stored.

Returns

a handle to the tuned matrix or INVALID_MAT in case an error occurs.

Possible error conditions

1. SPX_ERR_FILE: invalid filename argument or file read error.
2. SPX_ERR_TUNED_MAT: reproducing the matrix failed.

A.2.11 spx_index_t spx_mat_get_nrows (const spx_matrix_t * *A*)

Returns the number of rows of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of rows.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.12 spx_index_t spx_mat_get_ncols (const spx_matrix_t * *A*)

Returns the number of columns of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of columns.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.13 spx_index_t spx_mat_get_nnz (const spx_matrix_t * *A*)

Returns the number of nonzero elements of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of nonzeros.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.14 spx_partition_t* spx_mat_get_partition (spx_matrix_t * *A*)

Returns a partitioning object for the supplied matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

a valid partitioning object or INVALID_PART in case an error occurs.

A.2.15 spx_perm_t* spx_mat_get_perm (const spx_matrix_t * *A*)

Returns the permutation computed for the supplied matrix by applying the Reverse Cuthill-McKee algorithm.

Parameters

in	*A*	the tuned matrix handle.

Returns

a handle to the permutation object or INVALID_PERM in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid or the matrix has not been reordered.

A.2.16 spx_error_t spx_matvec_mult (spx_value_t *alpha,* const spx_matrix_t * *A,* spx_vector_t * *x,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x \tag{A.1}$$

where *alpha* is a scalar, x and y are vectors and A is a sparse matrix in the CSX format.

Parameters

in	*A*	the tuned matrix handle.
in	*alpha*	a scalar.
in	*x*	the input vector.
in,out	*y*	the output vector.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.17 spx_error_t spx_matvec_kernel (spx_value_t *alpha,* const spx_matrix_t * *A,* spx_vector_t * *x,* spx_value_t *beta,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x + beta \cdot y \tag{A.2}$$

where *alpha* and *beta* are scalars, x and y are vectors and A is a sparse matrix in the CSX format.

Parameters

in	A	the tuned matrix handle.
in	*alpha*	a scalar.
in	x	the input vector.
in	*beta*	a scalar.
in,out	y	the output vector.

Returns

an error code.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.18 spx_error_t spx_matvec_kernel_csr (spx_matrix_t * *A,* spx_index_t *nr_rows,* spx_index_t *nr_cols,* spx_index_t * *rowptr,* spx_index_t * *colind,* spx_value_t * *values,* spx_value_t *alpha,* spx_vector_t * *x,* spx_value_t *beta,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x + beta \cdot y \tag{A.3}$$

where *alpha* and *beta* are scalars, x and y are vectors and A is a sparse matrix. The matrix is originally given in the CSR format and converted internally into the CSX format. This higher-level routine hides the preprocessing phase of CSX. It can be efficiently used in a loop, since only the first call will convert the matrix into the CSX format and every subsequent call will use the previously tuned matrix handle.

Parameters

in	*A*	either an invalid matrix handle or a tuned matrix handle. If A is equal to an INVALID_MAT then the matrix in the CSR format is first converted to CSX. Otherwise, the valid (previously) tuned matrix handle is used to perform the multiplication.
in	*nr_rows*	number of rows of the matrix *A*.
in	*nr_cols*	number of columns of the matrix *A*.
in	*rowptr*	array *rowptr* of the CSR format.
in	*colind*	array *colind* of the CSR format.
in	*values*	array *values* of the CSR format.
in	*alpha*	a scalar.
in	*x*	the input vector.
in	*beta*	a scalar.
in,out	*y*	the output vector.

Returns

an error code.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_INPUT_MAT: the input data arrays do not correspond to a valid CSR representation.
3. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.19 spx_error_t spx_mat_destroy (spx_matrix_t * *A*)

Releases any memory internally used by the tuned matrix handle *A*.

Parameters

in	*A*	the tuned matrix handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.
2. SPX_ERR_MEM_FREE: deallocation failed.

A.2.20 spx_partition_t* spx_partition_csr (spx_index_t * *rowptr,* spx_index_t *nr_rows,* size_t *nr_threads*)

Creates a partitioning object for the matrix in the Compressed Sparse Row (CSR) format. This routine should be used in conjunction with the spx_matvec_kernel_csr() multiplication routine.

Parameters

in	*rowptr*	array *rowptr* of the CSR format.
in	*nr_rows*	number of rows of the matrix.
in	*nr_threads*	number of partitions of the matrix.

Returns

a partitioning object or INVALID_PART in case an error occurs.

A.2.21 spx_error_t spx_partition_destroy (spx_partition_t * *p*)

Releases any memory internally used by the partitioning handle *p*.

Parameters

in	*p*	the partitioning handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the partitioning handle is invalid.
2. SPX_ERR_MEM_FREE: deallocation failed.

A.2.22 spx_error_t spx_option_set (const char * *option,* const char * *string*)

Sets the *option* according to the description in *string* for the tuning process to follow. For available tuning options and how to set them refer to Table 3.2 of the User's Guide.

Parameters

in	*option*	the option to be set.

in	*string*	a description of how to set the option.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_MEM_FREE: deallocation failed.

A.2.23 void spx_options_set_from_env ( )

Sets the tuning options according to the environmental variables set on the command line. The available environmental variables include:

- NR_THREADS, for setting the number of threads
- MT_CONF, for setting the thread affinity
- XFORM_CONF, for selecting the substructure types that will be detected
- SAMPLES, for defining the number of samples
- SAMPLING_PORTION, for defining the portion of the matrix that will be sampled
- WINDOW_SIZE, for defining the size of the windows that will be used during th sampling process

Refer to Table 3.2 of the User's Guide for instructions on how to set each of the aforementioned variables.

A.2.24 spx_vector_t* spx_vec_create (size_t *size*, spx_partition_t * *p*)

Creates and returns a vector object, whose values must be explicitly initialized.

Parameters

in	*size*	the size of the vector to be created.
in	*p*	a partitioning handle.

Returns

a valid vector object or INVALID_VEC in case of an error.

A.2.25 spx_vector_t* spx_vec_create_from_buff (spx_value_t * *buff*, size_t *size*, spx_partition_t * *p*, spx_copymode_t *mode*)

Creates and returns a vector object, whose values are mapped to a user-defined array. If OP_SHARE is set, then the input buffer will be shared with the user and modifications will directly apply to it. If OP_COPY is selected, a copy of the input vector will be created and no modification of the original buffer will occur.

Parameters

in	*buff*	the user-supplied buffer.
in	*size*	the size of the buffer.
in	*p*	a partitioning handle.
in	*mode*	the copy mode (either OP_SHARE or OP_COPY).

Returns

a valid vector object or INVALID_VEC in case of an error.

A.2.26 spx_vector_t* spx_vec_create_random (unsigned long *size*, spx_partition_t * *p*)

Creates and returns a vector object, whose values are randomly filled.

Parameters

in	*size*	the size of the vector to be created.
in	*p*	a partitioning handle.

Returns

a valid vector object or INVALID_VEC in case of an error.

A.2.27 void spx_vec_init (spx_vector_t * *v*, spx_value_t *val*)

Initializes the vector object *v* with *val*.

Parameters

in	*v*	a valid vector object.
in	*val*	the value to fill the vector with.

A.2.28 void spx_vec_init_part (spx_vector_t * *v*, spx_value_t *val*, spx_index_t *start*, spx_index_t *end*)

Initializes the [*start*, *end*) range of the vector object *v* with *val*.

Parameters

in	*v*	a valid vector object.
in	*val*	the value to fill the vector with.

in	*start*	starting index.
in	*end*	ending index.

A.2.29 void spx_vec_init_rand_range (spx_vector_t * *v*, spx_value_t *max*, spx_value_t *min*)

Initializes the vector object *v* with random values in the range [*min*, *max*].

Parameters

in	*v*	a valid vector object.
in	*max*	maximum value of initializing range.
in	*min*	minimum value of initializing range.

A.2.30 spx_error_t spx_vec_set_entry (spx_vector_t * *v*, spx_index_t *idx*, spx_value_t *val*)

Sets the element at index *idx* of vector *v* to be equal to *val*. *idx* is assumed to be one-based.

Parameters

in	*v*	a valid vector object.
in	*idx*	an index inside the vector.
in	*val*	the value to be set.

A.2.31 void spx_vec_scale (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_scalar_t *num*)

Scales the input vector *v1* by a constant value *num* and places the result in vector *v2*, i.e. $v2 = num * v1$.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*num*	the constant by which to scale *v1*.

A.2.32 void spx_vec_scale_add (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_scalar_t *num*)

Scales the input vector *v2* by a constant value *num*, adds the result to vector *v1* and places the result in vector *v3*, i.e. $v3 = v1 + num * v2$.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*num*	the scalar by which to scale *v1*.

A.2.33 void spx_vec_scale_add_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_scalar_t *num*, spx_index_t *start*, spx_index_t *end*)

v3[*start*...*end*) = *v1*[*start*...*end*) + *num* * *v2*[*start*...*end*)

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*num*	the scalar by which to scale *v1*.
in	*start*	starting index.
in	*end*	ending index.

A.2.34 void spx_vec_add (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*)

Adds the input vectors *v1* and *v2* and places the result in *v3*, i.e. *v3* = *v1* + *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.

A.2.35 void spx_vec_sub (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*)

Subtracts the input vector *v2* from *v1* and places the result in *v3*, i.e. *v3* = *v1* - *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

in	*v3*	a valid vector object.

A.2.36 spx_value_t spx_vec_mul (const spx_vector_t * *v1*, const spx_vector_t * *v2*)

Returns the product of the input vectors *v1* and *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

Returns

the product of the input vectors.

A.2.37 void spx_vec_add_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_index_t *start*, spx_index_t *end*)

Adds the range [start...end) of the input vectors *v1* and *v2* and places the result in *v3*, i.e. *v3*[start...end) = *v1*[start...end) + *v2*[start...end).

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*start*	starting index.
in	*end*	ending index.

A.2.38 void spx_vec_sub_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_index_t *start*, spx_index_t *end*)

Subtracts the input vector *v2* from *v1* in the range [start...end) and places the result in *v3*, i.e. *v3*[start...end-1] = *v1*[start...end-1] - *v2*[start...end-1].

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*start*	starting index.

in	*end*	ending index.

A.2.39 spx_value_t spx_vec_mul_part (const spx_vector_t * *v1*, const spx_vector_t * *v2*, spx_index_t *start*, spx_index_t *end*)

Returns the product of the range [*start*, *end*) of the input vectors *v1* and *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*start*	starting index.
in	*end*	ending index.

Returns

the product of the input vectors.

A.2.40 spx_error_t spx_vec_reorder (spx_vector_t * *v*, spx_perm_t * *p*)

Reorders the input vector v according to the permutation p, leaving the original vector intact.

Parameters

in	*v*	a valid vector object.
in	*p*	a permutation.

Returns

the permuted input vector or INVALID_VEC in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.

A.2.41 spx_error_t spx_vec_inv_reorder (spx_vector_t * *v*, spx_perm_t * *p*)

Inverse-reorders the permuted input vector *v*, according to the permutation *p*.

Parameters

in,out	*v*	the vector object to be inverse-reordered.
in	*p*	a permutation.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.

A.2.42 void spx_vec_copy (const spx_vector_t * *v1*, spx_vector_t * *v2*)

Copies the values of v1 to v2.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

A.2.43 int spx_vec_compare (const spx_vector_t * *v1*, const spx_vector_t * *v2*)

Compares the values of v1 and v2. If they are equal it returns 0, else -1.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

A.2.44 void spx_vec_print (const spx_vector_t * *v*)

Prints the input vector v.

Parameters

in	*v*	a valid vector object.

A.2.45 spx_error_t spx_vec_destroy (spx_vector_t * *v*)

Destroys the input vector v.

Parameters

in	*v*	a valid vector object.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

A.2.46 void spx_timer_clear (spx_timer_t * *t*)

Initialises a timer object.

Parameters

in	*t*	timer object to be initialized.

A.2.47 void spx_timer_start (spx_timer_t * *t*)

Starts a timer object.

Parameters

in	*t*	timer object to be launched.

A.2.48 void spx_timer_pause (spx_timer_t * *t*)

Pauses a timer object.

Parameters

in	*t*	timer object to be paused.

A.2.49 double spx_timer_get_secs (spx_timer_t * *t*)

Returns the elapsed time in seconds.

Parameters

in	*t*	a timer object.

Returns

the elapsed seconds.

A.2.50 void spx_log_disable_all ( )

Disables logging in SparseX.

A.2.51 void spx_log_error_console ( )

Activates logging of the Error level on stderr.

A.2.52 void spx_log_warning_console ( )

Activates logging of the Warning level on stderr.

A.2.53 void spx_log_info_console ( )

Activates logging of the Info level on stderr.

A.2.54 void spx_log_verbose_console ( )

Activates logging of the Verbose level on stderr.

A.2.55 void spx_log_debug_console ( )

Activates logging of the Debug level on stderr.

A.2.56 void spx_log_error_file ( )

Activates logging of the Error level on a file. The file name should be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.57 void spx_log_warning_file ( )

Activates logging of the Warning level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.58 void spx_log_info_file ( )

Activates logging of the Info level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.59 void spx_log_verbose_file ( )

Activates logging of the Verbose level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.60 void spx_log_debug_file ( )

Activates logging of the Debug level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.61 void spx_log_all_console ( )

Activates all logging levels and redirects output to stderr.

A.2.62 void spx_log_all_file (const char * *file*)

Activates all logging levels and redirects output to a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.↩ log" file will be created. If the file already exists it will be overwritten.

Parameters

in	*file*	a filename.

A.2.63 void spx_log_set_file (const char * *file*)

Sets the file that will be used when logging is redirected to a file. If the file already exists it will be overwritten.

Parameters

in	*file*	a filename.

A.2.64 spx_errhandler_t spx_err_get_handler ( )

Returns a pointer to the current error handler (either default or user-defined).

Returns

the current error handling routine.

A.2.65 void spx_err_set_handler (spx_errhandler_t *handler*)

This function allows the user to change the default error handling policy with a new one, which must conform to the signature provided above.

Parameters

in	*handler*	user-defined routine.

Bibliography

R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 32–41, Minneapolis, MN, USA, 1992. IEEE Computer Society.

J. Ankit. pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures. 2008.

S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013. URL http://www.mcs.anl.gov/petsc.

R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1987. ISBN 0-89871-328-5.

V. H. F. Batista, G. O. Ainsworth Jr., and F. L. B. Ribeiro. Parallel structurally-symmetric sparse matrix-vector products on multi-core processors. *Computing Research Repository (CoRR)*, abs/1003.0952, 2010.

R. Boisvert, R Pozo, and Karin Remington. The matrix market exchange formats: Initial design. Technical Report NISTIR-5935, National Institute of Standards and Technology, December 1996.

A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE International Parallel & Distributed Processing Symposium*, pages 721–733, Anchorage, AK, USA, 2011. IEEE Computer Society.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*. ACM, 1969.

T. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38:1–25, 2011. URL http://www.cise.ufl.edu/research/sparse/matrices.

T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006. ISBN 0-89871-613-6.

I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, 5(1):18–35, 1979.

I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1989. ISBN 0-19853-421-3.

S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package i: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18(8):1145–1151, 1982.

S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, 2000.

T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris. Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(10):1930–1940, 2013. IEEE.

G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*, 50(1):36–77, 2009.

M. Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, chapter 31. ACM, 2005.

G. Henry. Intel® MKL NUMA notes. *Intel® Developer Zone*, 2012. URL http://www.software.intel.com/en-us/articles/intel-mkl-numa-notes.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49 (6):409–436, 1952.

E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Proceedings of the International Conference on Computational Sciences – Part I*, pages 127–136. Springer-Verlag, 2001.

Intel® Coorporation. *Intel® Math Kernel Library*. Intel® Coorporation, 2013a. URL http://software.intel.com/en-us/intel-mkl.

Intel® Coorporation. *Intel® MKL for Linux OS: User's Guide*, 2013b. URL http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

V. Karakasis. *Optimizing the Sparse Matrix-Vector Multiplication kernel for Modern Multicore Computer Architectures.* PhD thesis, National Technical University of Athens, 2012.

V. Karakasis, G. Goumas, and N. Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-09)*, IPDPS'09, Rome, Italy, 2009. IEEE Computer Society.

V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(10):1930–1940, 2013. IEEE.

D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. volume 23, pages 56–65, 2003.

K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th conference on Computing Frontiers*, Ischia, Italy, 2008b. ACM.

K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. CSX: An extended compression format for SpMV on shared memory systems. In *16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, San Antonio, TX, USA, 2011. ACM.

N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar. Next generation intel®micro-architecture (nehalem) clocking architecture. In *IEEE Symposium on VLSI Circuits*, Honolulu, HI, USA, 2008. IEEE.

C. Lattner. LLVM and Clang: Advancing Compiler Technology. In *Free and Open Source Developers' European Meeting*, Brussels, Belgium, February 2011. IEEE Computer Society. URL http://clang.llvm.org/.

C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, San Jose, CA, USA, 2004. IEEE Computer Society. URL http://www.llvm.org/.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Korgh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 367–378, 2008.

W. H. Liu and A. H. Sherman. Comparative Analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM Numerical Analysis*, pages 198–213, 1976.

J. D. McCalpin. *STREAM: Sustainable memory bandwidth in high performance computing*, 1995. URL http://www.cs.virginia.edu/stream/.

A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Super-computing'99*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

U. W. Pooch and A. Nieder. A survey of indexing techniques for sparse matrices. *ACM Computing Surveys*, 5(2):109–133, 1973.

Y. Saad. *Numerical methods for large eigenvalue problems.* Manchester University Press ND, 1992. ISBN 0-89871-534-2.

Y. Saad. Sparskit: A basic tool kit for sparse matrix computations, 1994.

Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, 2nd edition, 2003. ISBN 0-89871-534-2.

Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientifc and Statistical Computing*, 7(3):856–869, 1986.

R. P. Tewarson. *Sparse Matrices.* Academic Press, 1973. ISBN 0-124-11098-3.

W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *IEEE Proceedings*, 55 (11):1801–1809, 1967.

D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, S. Margherita Ligure, Italy, 1995. ACM.

R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. volume 16, 2005.

R. W. Vuduc and H. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High performance computing and communications.*, volume 3726 of *Lecture notes in computer science*, pages 807–816. Springer, Berlin/Heidelberg, 2005.

J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *4th International conference on high performance computing (HiPC '97)*, 1997.

S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM - A Direct Path to Dependable Software*, 52(4):65–76, 2009.