National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Optimizing the Sparse Matrix-Vector Multiplication Kernel for Modern Multicore Computer Architectures

Ph.D. Thesis

Vasileios K. Karakasis

Electrical and Computer Engineer, Dipl.-Ing.

Athens, Greece
December, 2012

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Optimizing the Sparse Matrix-Vector Multiplication Kernel for Modern Multicore Computer Architectures

Ph.D. Thesis

Vasileios K. Karakasis

Electrical and Computer Engineer, Dipl.-Ing.

Athens, Greece
December, 2012

Vasileios K. Karakasis
Ph.D., National Technical University of Athens, Greece.

Στους γονείς μου,
Κωνσταντίνο και Ειρήνη,
και στα αδέλφια μου,
Μάριο, Όλγα, Αλέξανδρο

Abstract

This thesis focuses on the optimization of the Sparse Matrix-Vector Multiplication kernel (SpMV) for modern multicore architectures. We perform an in-depth performance analysis of the kernel and identified its major performance bottlenecks. This allows us to propose an advanced storage format for sparse matrices, the Compressed Sparse eXtended (CSX) format, which targets specifically the minimization of the memory footprint of the sparse matrix. This format provides significant improvements in the performance of the SpMV kernel in a variety of matrices and multicore architectures, maintaining a considerable performance stability. Finally, we investigate the performance of the SpMV kernel from an energy-efficiency perspective, in order to identify the execution configurations that lead to optimal performance-energy tradeoffs.

Keywords: high performance computing; scientific applications; sparse matrix-vector multiplication; multicore; data compression; energy-efficiency; SpMV; CSX; HPC

Contents

List of Figures

List of Tables

List of Algorithms

Αντί Προλόγου

Πολλές οι σκέψεις και τα συναισθήματα που γεμίζουν το μυαλό μου φτάνοντας στο τέλος του «αγώνα» που λέγεται διδακτορικό. Ανακούφιση και ικανοποίηση για την τελείωσή του, προσδοκία για το μέλλον, νοσταλγία για τις όμορφες και δύσκολες στιγμές του. Κάθε τι που με τον ένα ή τον άλλο τρόπο σημαδεύει την πορεία σου είναι συνήθως άρρηκτα συνδεδεμένο με τον ανθρώπινο παράγοντα, άμεσα ή έμμεσα, διακριτικά ή εύγλωττα. Το διδακτορικό δεν θα μπορούσε να είναι εξαίρεση, όχι μόνο λόγω του πολυετούς του, αλλά και λόγω του δυναμικού και του φάσματος των συναισθημάτων που συχνά-πυκνά προκαλεί κατά την διάρκειά του. Δεν πρόκειται, λοιπόν, επ' ουδενί για ένα «one-man-show», αλλά αποτελεί την συνισταμένη της συνεισφοράς, υλικής, πνευματικής, ψυχολογικής, εμφανούς ή αφανούς, ενός συνόλου εξαιρετικών ανθρώπων.

Κατ' αρχήν, θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, αναπληρωτή καθηγητή Νεκτάριο Κοζύρη, για την εμπιστοσύνη που μου έδειξε στο δύσκολο έργο του διδακτορικού, εν πρώτοις αποδεχόμενός με στο διδακτορικό πρόγραμμα, και εν συνεχεία, φροντίζοντας για την απρόσκοπτη και επί της ουσίας εκπόνηση της διατριβής μου, μέσω της συνεχούς επαφής ολόκληρης της ομάδας μας με την τελευταία τεχνολογία και της ενθάρρυνσης της ενεργού συμμετοχής μας στην ευρύτερη ερευνητική κοινότητα. Οφείλω, επίσης, την ευχαριστία μου και στους συνεπιβλέποντες, καθηγητή Ανδρέα-Γεώργιο Σταφυλοπάτη, για την στενή, αρμονική και εποικοδομητική συνεργασία μας κατά τα πρώτα χρόνια του διδακτορικού, και καθηγητή Παναγιώτη Τσανάκα.

Η διδακτορική αυτή διατριβή, όμως, ίσως να μην υπήρχε χωρίς την αμέριστη συμπαράσταση του Λέκτορα Γιώργου Γκούμα. Η συμβολή του Γιώργου ήταν καθοριστική, τόσο στο επίπεδο της οργάνωσης και της ερευνητικής ισχυροποίησης της ομάδας μας, όσο και συγκεκριμένα στην δική μου διατριβή. Από τα πρώτα μου βήματα στον απαιτητικό χώρο του High Performance Computing, ο Γιώργος βρισκόταν συνεχώς δίπλα μου, στις ατυχίες, που υπήρξαν αρκετές, και στις επιτυχίες, που υπήρξαν σημαντικές και ελπιδοφόρες για το μέλλον.

Θα ήταν μεγάλη παράλειψη στο σύντομο αυτό σημείωμα να μην αναφερθώ στα υπόλοιπα μέλη του εργαστηρίου, παλαιότερα και νεώτερα, που με το υψηλό τους επιστημονικό και πνευματικό επίπεδο δημιουργούν ένα σημαντικό ρεύμα ανόδου που συμπαρασύρει και εμπνέει κάθε νεοεισερχόμενο στην ομάδα. Δύσκολο να ξεχωρίσεις συγκεκριμένα πρόσωπα χωρίς ενδεχομένως να «αδικήσεις» κάποια άλλα, καθότι το ευχάριστο κλίμα εργασίας και συνεργασίας δημιουργήθηκε και συντηρείται συλλογικά. Από τον Βαγγέλη Κούκη, που απετελέσε ένα σημαντικό έναυσμα ώστε να ενταχθώ στην ομάδα του CSLab[1], γνωρίζοντάς τον ως προπτυχιακός φοιτητής, τον Κορνήλιο Κούρτη, με τον οποίο συνεργαστήκαμε και με ενέπνευσε σε μεγάλο βαθμό ο τρόπος εργασίας του και αντίληψης των πραγμάτων, τον Γιώργο Τσουκαλά, τον Νίκο Αναστόπουλο, τον Κωστή Νίκα, τον Τάσο Νάνο, την Γεωργία Κουβέλη, τον Τάσο Κατσιγιάννη, τον Στέφανο Γεράγγελο, με τους οποίους περάσαμε όμορφες στιγμές τεχνικών ή μη αναλύσεων εντός και εκτός συνόρων, τον Θοδωρή Γκούντουβα, ο οποίος συνέβαλε με την διπλωματική του εργασία στην βελτίωση και επέκταση καίριων σημείων της διατριβής μου, αλλά και τα υπόλοιπα μέλη των «Παραλλήλων» και «Κατανεμημένων», όλοι μαζί συνέβαλαν και συμβάλλουν στην δημιουργία του ευχάριστου κλίματος που δρα ως κινητήριος δύναμη για κάθε επιτυχία.

Οφείλω, όμως, να ευχαριστήσω θερμά και τους «εκτός του κάδρου» συμβάλλοντες, τους αεί φίλους μου από τα λυκειακά και προπτυχιακά χρόνια, Νίκο, Κώστα και Σωκράτη, που απετέλεσαν συχνά την διαφυγή και διέξοδο από την τύρβη της καθημερινότητας, και την κοπέλα μου, Ευανθία, που τα τελευταία χρόνια βρισκόταν συνεχώς δίπλα μου, μεταδίδοντάς μου την δύναμη και την αγάπη της, για να συνεχίσω στον δύσκολο δρόμο του διδακτορικού.

Τέλος, αλλά μάλλον ως ακρογωνιαίοι παρά ακροτελεύτιοι, οι γονείς μου, Κωνσταντίνος και Ειρήνη, και τα αδέλφια μου, Μάριος, Όλγα και Αλέξανδρος, αξίζουν την αμέριστη ευγνωμοσύνη και ευχαριστία μου για όλα όσα μου έχουν προσφέρει όλα αυτά τα χρόνια, υλικά, ηθικά, πνευματικά. Τους αφιερώνω αυτό το πόνημα.

Με βαθειά εκτίμηση,
Βασίλειος Κ. Καρακάσης

[1] Η διατριβή αυτή εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων του Τομέα Τεχνολογίας Πληροφορικής και Υπολογιστών της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσοβίου Πολυτεχνείου.

1

Introduction

Research in sparse matrices has been active since the early days of computing systems. Their use is intertwined strongly with the solution of large sparse linear systems, whose mathematical and computational properties, as well as their performance optimization opportunities, are still of particular interest in the applied mathematics and computer science communities. The advent of modern multicore architectures has posed new challenges to the performance optimization of sparse matrix kernels, which are now even more important for the efficient execution of large sparse linear systems. In this chapter, we present briefly the different methods for solving large sparse linear systems and designate the importance of the sparse matrix-vector multiplication kernel, which has been the key motivation for this work. We also discuss the main challenges posed by the modern multicore architectures in the execution of this kernel and give an overview of our approach for its optimization.

1.1 Sparse linear systems

Sparse matrices are finite matrices dominated by zero elements. These matrices arise often with the discretization of partial differential equations (PDE) in finite element methods (FEM) and are usually involved in the solution of large linear systems of the form

$$Ax = b, \tag{1.1}$$

where A is an $n \times n$ *coefficient matrix*, b is the *right-hand side vector*, and x is the *vector of unknowns*. There are two large categories of methods for solving linear systems: *direct* and *iterative solution methods*.

The direct solution methods compute the exact solution of a linear system by *factorizing* the matrix A, i.e., expressing it as a product of two or more matrices, and then relying on forward and backward substitution for computing the unknown vector x. Well-known direct solution methods include the Gaussian elimination, the LU decomposition and the Cholesky factorization [Duff et al., 1989; Barrett et al., 1987; Davis, 2006]. The cost of these methods is proportional to the matrix-matrix multiplication cost [Strassen, 1969], which can be prohibitive for large dense linear systems. For sparse matrices, though, this cost is at the order of the non-zero elements of the matrix, but in this case the implementation of direct methods is not as straightforward as for dense matrices. During the factorization process of the original matrix, new non-zero elements, called *fill-in elements*, may be introduced in places which were initially occupied by zeros. This complicates both the data structures needed to store the sparse matrix and the factorization process. The storage of sparse matrices must efficiently support the insertion of the fill-in elements, whereas the factorization process must be preceded by a fill-in minimization step, in order to reduce the overall insertion overhead [Saad, 2003]. This increased cost of sparse direct methods favors the use of approximate iterative methods for the solution of large sparse linear systems.

The iterative solution methods do not compute the exact solution of a linear system, but instead, they try to provide a close approximation of the solution. The iterative methods for linear systems can be divided into two large categories [Barrett et al., 1987; Saad, 2003]:

(a) the *stationary methods* and

(b) the *projection methods*.

The stationary methods start with an initial approximation x_0 of the system's solution, which then refine iteratively by trying to annihilate one or more components of the residual vector $b - Ax$ at a time. In its general form, a stationary method can be written as following [Barrett et al., 1987]:

$$x^{(k+1)} = Bx^{(k)} + c \tag{1.2}$$

At each iteration k, the next approximation $x^{(k+1)}$ of the solution is computed by multiplying the current approximation with a sparse coefficient matrix B, derived from the original matrix A. The most common stationary methods are the Jacobi, Gauss-Seidel and the Successive Overrelaxation (SOR) methods. The convergence of these methods relies on strict requirements on the structure of the original matrix and, therefore, it is not guaranteed for all matrices [James and Riha, 1975; Ferziger and Peric, 2001; Saad, 2003].

The projection methods try to extract an approximate solution $\tilde{x}$ of the system from a subspace $\mathcal{K}$ of $\mathbb{R}^n$ by constraining the residual vector $r = b - A\tilde{x}$ to be orthogonal to a subspace $\mathcal{L}$ of $\mathbb{R}^n$. Starting from an initial guess x_0, these methods proceed at every iteration by selecting a pair of subspaces $\mathcal{K}$ and $\mathcal{L}$ and advance x in $\mathcal{K}$, so that the new residual is orthogonal to $\mathcal{L}$. In practice, these methods can be viewed as a generalization of the steepest descent method, where at each iteration the selected solution is the one that minimizes the distance from the right hand side vector b. Indeed, the steepest descent method can be obtained by setting $\mathcal{K} = \mathcal{L} = r$ [Saad, 2003].

The most successful and widely used projection methods are the Krylov subspace methods [Saad, 1981]. These methods rely on Krylov subspaces for drawing the approximate solution at each iteration. A Krylov subspace is defined as

$$\mathcal{K}_m(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^{m-1}v\}, \tag{1.3}$$

where the span operator denotes the set of all linear combinations of its vector arguments. At each iteration m, a Krylov method draws a solution from the subspace $\mathcal{K}_m(A, r_0)$, where A is the system's coefficient matrix and r_0 is the residual of the initial guess x_0. A Krylov method can be viewed as a polynomial approximation of A^{-1}. Indeed, if the method has converged after m iterations to a solution $\tilde{x}$ and $x_0 = 0$, then the following equations must hold:

$$x \approx \tilde{x} \tag{1.4}$$

$$A^{-1}b \approx \left(\sum_{i=1}^{m-1} \alpha_i A^i\right) b \tag{1.5}$$

The disadvantage of iterative solution methods compared to direct ones is the lack of robustness. The convergence of stationary methods cannot be guaranteed for all matrices, while the convergence ratio of the projection methods may be too slow to be practical. For this reason, it is desirable to transform the initial problem to an equivalent one with better convergence characteristics. This process is known as *preconditioning* of the linear system.

Preconditioned Krylov methods are currently the most widely used methods for solving large sparse linear systems. Additionally, these methods involve a small and well-defined set of computational kernels, which favor high-performance parallel implementations. Among the most prominent Krylov methods are the Generalized Minimum Residual (GMRES) method and the Conjugate Gradient (CG) method with its variations, Bi-CG and Bi-CG Stabilized [Saad and Schultz, 1986; Hestenes and Stiefel, 1952; van der Vorst, 1992].

1.2 The computational aspect of iterative methods

Krylov iterative methods for the solution of sparse linear systems involve the execution of the following four computational operations [Saad, 2003; Hoemmen, 2010]:

(a) *Vector updates.* A vector update is a combination of scalar multiplication and vector addition. Given two vectors, x and y, and a scalar α, the vector update operation is of the form

$$x = x + \alpha y.$$

This operation is also known as AXPY, named after the corresponding BLAS subroutines [Lawson et al., 1979].

(b) *Dot products.* The dot product is the inner product of two vectors. Thus, given two x and y vectors with n elements, the dot product is calculated as

$$t = x^T y = \sum_{i=1}^{n} x_i y_i.$$

(c) *Matrix-by-vector products.* This is the product of a sparse matrix A and a vector x:

$$y = Ax.$$

The sparse matrix A is the coefficient matrix of the system or the transformed coefficient matrix in case of a preconditioned Krylov method.

(d) *Preconditioner operations.* These are operations related to the preconditioning process and usually involve the direct solution of an 'easy' linear system.

Each of the above well-defined operations forms a *computational kernel*. The most time-consuming kernel in a Krylov method is the matrix-by-vector product, which we will henceforth call *Sparse Matrix-Vector Multiplication* kernel, or simply SpMV. Figure 1.1 shows an execution time breakdown of a typical serial implementation of the CG method in a modern multicore architecture for different coefficient matrices. The majority of the execution time, surpassing 90% in some cases, is spent executing the SpMV kernel. This kernel, therefore, becomes of crucial importance for the acceleration of iterative solution methods and has been recently characterized as one of the computational problems whose optimization will play a significant role in scientific computing in the next decade [Colella, 2004; Asanovic et al., 2006].

However, the importance of the preconditioner operations and the dot products must not be underestimated. An expensive preconditioner, e.g., the incomplete LU factorization (ILU), may consume the majority of the execution time of an iterative method. Nonetheless, since preconditioners are actually direct solution methods, they are not directly related to the iterative process itself, as is the SpMV kernel and the dot products, but they are rather part of the intriguing problem of direct solution methods for sparse matrices [Asanovic et al., 2006]. Additionally, conversely to the SpMV kernel and the vector operations, there is not such a strict requirement in high precision arithmetic for

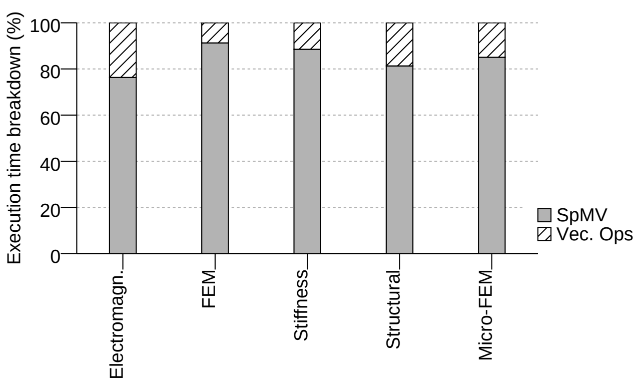


Figure 1.1: Execution time breakdown for the non-preconditioned CG iterative method for different problem categories.

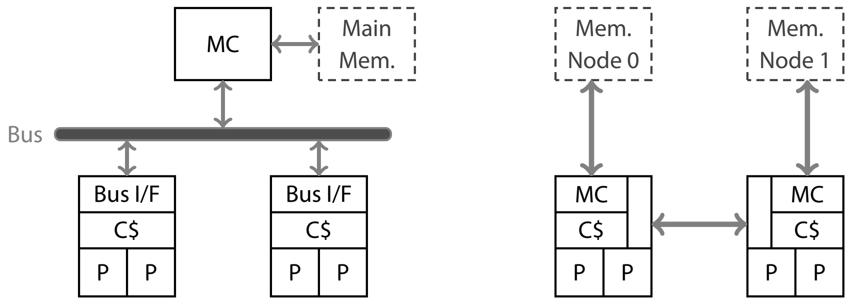
the preconditioners, therefore allowing high-performance low precision implementations [Buttari et al., 2008].

The dot products can also become a performance bottleneck in certain cases, especially in a parallel execution environment. The reduction operation required to calculate the final result has limited parallelism, logarithmic to the input size *n*. Although large input sizes lead to adequate parallelism, undesirable side-effects may be introduced, such as cache misses or increased communication cost, hindering the exploitation of the available parallelism.

Implementing high-performance iterative solution methods for large sparse linear systems is indeed an extremely challenging task with multiple aspects that need a careful and in-depth examination. The advent of multicore and manycore architectures has introduced further challenges for the optimization of these methods, that should be successfully addressed in order to harness the performance capacity of the new hardware.

1.3 Challenges of multicore architectures

It is not very long ago that the quest for higher processor frequencies was abandoned by the leader chip manufacturers, due to the need for higher energy efficiency and the need to keep Moore's law alive at the same time. The interest of academia and industry has since shifted toward incorporating multiple cores inside the same physical processor, inaugurating the *multicore era* and setting up new challenges. The use of multiple cores or hardware threads inside the same physical processor has broaden the gap between the rate that the processor can now consume data and the rate that the memory subsystem can supply



(a) Symmetric shared memory. **(b)** NUMA.

Figure 1.2: The two current trends in modern multicore architectures: symmetric shared memory and non-uniform memory access (NUMA) architectures.

data, making the 'memory-wall' problem [Wulf and McKee, 1995] even tenser.

Symmetric shared memory multicore architectures are affected the most from the memory-processor speed gap. In these architectures, all the memory and interprocessor communication requests are routed through the same front-end bus to the central, off-chip memory controller and the peer processors (Figure 1.2a). Apparently, this centralized logic, in conjunction with the low bandwidth and high latency that the off-chip communication carries with, can easily become the hotspot of a memory-intensive application. Large and complex cache hierarchies, unfortunately, can limit this effect in the short term only.

The need to extract more parallelism from the hardware, given the slow DRAM speed evolution pace, demands a more decentralized approach. The Non-Uniform Memory Access (NUMA) architectures 'move' the memory controller inside the processor chip and use dedicated hardware for the interprocessor communication (Figure 1.2b). The main memory, though still shared, is no more uniformly accessible from every processor in the system: it is split into multiple nodes, each one assigned to a single processor. The available bandwidth is now ample for the communication between a processor and its local memory node, but accessing remote nodes requires multiple and costly hops. Two more challenges arise with the NUMA architectures for the memory-intensive kernels:

(a) The increase in the available memory bandwidth may reveal weaknesses in the computational part of the kernel, that where otherwise hidden by the very slow access to the main memory.

(b) The kernel's performance can be now very sensitive to the correct placement of its data on the different memory nodes.

The latter challenge poses an additional burden for the programmer, who might need to explicitly alter her kernel to fully exploit the NUMA capabilities of a system.

These challenges must be successfully addressed in order to implement high-performance iterative solvers for the future computer architectures. We believe that the communication[1] versus computation tradeoff will become increasingly important in the next decade, as advances in the hardware technology will allow a merger of the now special-purpose high-performance architectures (e.g., GPUs) and the flexible high-performance general-purpose processors.

1.3.1 Energy consumption considerations

The power dissipation of modern processors has been of increasing concern in the last decade. The shrinking of the integration scale and the accompanying increase in the operating frequency of the circuit would have led to such a power density that could render the processor cooling at least impractical, if not problematic. The realization of this problem has brought the processor clock frequency scaling to an abrupt halt, against the ambitious predictions back in the early 2000's [ITRS, 2001]. The performance, however, was not compromised as more processor cores are now integrated in the same chip offering increasingly higher performance levels by exploiting parallelism. Nonetheless, as the integration scale is still constantly shrinking toward a few nanometers, leakage or 'idle' power has started to dominate the total power dissipation of modern processors [Ahmed and Schuegraf, 2011]. Resource sharing in the hardware and frequency scaling can help in controlling the power dissipation of modern processors and increase their energy efficiency.

Modern multicore processors share hardware resources in different levels ranging from pipeline resources to high-level caches and the memory controller. Efficient resource sharing among the threads of a multithreaded application can be beneficial in terms of overall power dissipation, since 'unused' parts of the chip can be shut down, saving considerable amount of wasted energy [Kaxiras et al., 2001]. Similarly, the effect of scaling down the CPU frequency in the total power dissipation of the processor can be significant, due to the their superlinear relation [Brooks et al., 2000; Kaxiras and Martonosi, 2008]. The consequent loss in the overall performance can be minimized, while maintaining a high energy efficiency, by dynamically scaling down the frequency in non performance-critical parts [Kaxiras and Martonosi, 2008; Curtis-Maury et al., 2008]. Unfortunately, there is no magic recipe that maximizes the

[1] With the term *communication*, we denote all the off-chip communication traffic, incl. the memory operations.

performance and minimizes the energy consumption at the same time. There exists a set of optimal tradeoffs, instead, that must be balanced depending on the specific needs of each application and the underlying system [Karakasis et al., 2011]. Finding an optimal performance-energy tradeoff is not an easy task and must involve both the hardware and the software. The hardware must become more flexible offering facilities for monitoring the power dissipation of the different components and for dynamically switching on and off 'power-hungry' parts of the processor. The software, on the other hand, must be able to utilize these features efficiently either at the operating system level or through a power-aware runtime environment, in order to provide the means for high-performance energy-efficient computing.

1.3.2 The algorithmic nature of the SpMV kernel

The matrix-vector multiplication kernels, both dense and sparse, can be viewed as a sequence of dot product operations between each row vector of the matrix and the input vector. Algorithm 1.1 shows this high-level approach. The algorithm sweeps once through the whole matrix and performs a constant number of floating point operations per element. This automatically leads to a $\Theta(1)$ flop:byte ratio compared to the $\Theta(n)$ (n being the dimension of the matrix) of the matrix-matrix multiplication kernels. In practice, this means that in order to avoid bottlenecks, the memory hierarchy must be able to provide data to the processor at a comparable speed, which is hardly ever the case for any modern mainstream microarchitecture. The situation gets worse with sparse matrices, where the kernel must first retrieve the non-zero element's location information, before accessing it.

```
1: procedure MatVec($A$::in, $x$::in, $y$::out)
2:     foreach row vector $a_{i*}$ in $A$ do
3:         $y_i \leftarrow a_{i*}^T x$
4:     end for
```

Algorithm 1.1: High-level representation of the matrix-vector multiplication.

Figure 1.3 shows the speedup of a multithreaded SpMV baseline implementation and the consumed main memory bandwidth in GB/s in a two-way quad-core symmetric shared memory system. Despite exhibiting ample parallelism [Buluç et al., 2009], the SpMV kernel fails to scale beyond four threads due to the saturation of the system's memory bandwidth. An 85% of the available memory bandwidth is already consumed by the two threads and it is to-

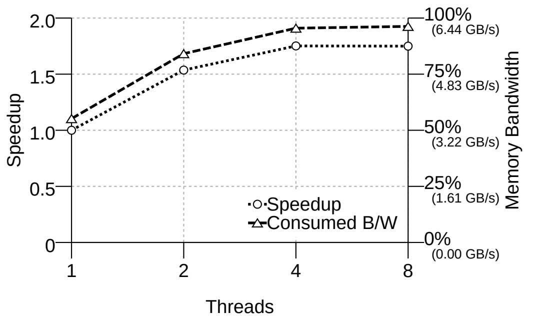


Figure 1.3: Demonstration of the SpMV kernel speedup in relation to the memory bandwidth consumption in a two-way quad-core symmetric shared memory system.

tally saturated from four threads onward. The minimization of the memory footprint of the sparse matrix, therefore, becomes of vital importance for the optimization of the SpMV kernel, especially for the symmetric shared memory architectures.

Despite being the most prominent, memory intensity is not the sole problem of the SpMV kernel. Even more challenging is that the performance of the kernel depends heavily on the sparsity structure of the input matrix, leading to dramatic performance variations. For example, matrices with a very irregular structure can lead to a considerable amount of cache misses and a significant load imbalance that will ruin performance. Conversely, regular ones may expose more the computational part of the kernel and require a less memory-centric approach. It is therefore imperative for a successful SpMV optimization to adapt to the specificities of the unknown input matrices during the runtime, and this is probably one of the greatest challenges in optimizing this particular kernel.

1.3.3 The energy efficiency of the SpMV kernel

The streaming nature of the SpMV kernel renders its performance very sensitive to the sharing of the levels of the cache hierarchy among the threads. Multiple threads executing the SpMV kernel and sharing the same cache level contend for cache space, causing an increase in capacity misses, thus deteriorating significantly the performance of the kernel. On the other hand, 'spreading' the threads across different caches, it is not a power-friendly configuration, despite the increase in performance, since caches take up a significant amount of the total power dissipation of the processor [Kaxiras and Martonosi, 2008].

This increase in power dissipation, however, can be compensated with the use of multiple threads running at a lower frequency, without compromising the kernel's performance. Selecting the appropriate thread placement and processor frequency for obtaining an optimal compromise between performance and energy consumption is very challenging for the SpMV kernel, since its performance is highly dependent on the structure of the input matrix. Examining the structure of the matrix beforehand and using an advanced classification scheme might be necessary for selecting an execution configuration for the kernel, in order to achieve an optimal performance-energy tradeoff.

1.4 Contribution of this thesis

In this thesis we focus on the optimization of the SpMV kernel in modern multicore architectures as being the most time-consuming kernel in iterative methods for the solution of sparse linear systems. Despite being an integral part of successful iterative methods and possibly very time-consuming under certain circumstances, we do not focus in optimizing preconditioner operations. These operations are actually direct solution methods and can be therefore considered not to be directly related to the iterative solution process itself. Conversely, they belong to the intriguing problem of direct methods for sparse matrices and their optimization can be considered as another important, though distinct, computational problem [Davis, 2006; Asanovic et al., 2006].

1.4.1 In-depth performance analysis and prediction models

The performance of the SpMV kernel has been of significant concern in the past; however, a robust performance analysis that would quantify the impact of the alleged performance bottlenecks has not been conducted. As a result, there has not been a clear conclusion as of what are the major performance problems of the SpMV kernel. In this thesis, we take an in-depth look at the algorithmic characteristics of the SpMV kernel and perform a quantitative analysis on a wide range of sparse matrices and modern multicore architectures, in order to assess the exact impact on the performance of the kernel. The conducted analysis offers a clear understanding of the performance of the SpMV kernel and serves as a guide for efficient implementations. Indeed, the identification of the memory bandwidth bottleneck as the key and inherent performance problem of SpMV in a multithreaded context has guided us to the adoption of explicit compression techniques in optimizing the kernel. Additionally, based on this analysis, we have developed performance models for the prediction of the optimal block for fixed size blocking methods.

1.4.2 The CSX storage format for sparse matrices

The major contribution of this thesis is the Compressed Sparse eXtended (CSX) storage format for sparse matrices. Based on the idea of explicitly compressing the indexing structure of a sparse matrix storage format, in order to minimize the memory footprint of the matrix [Willcock and Lumsdaine, 2006; Kourtis et al., 2010], CSX incorporates the notion of blocking in order to achieve even higher compression rates and better computational characteristics [Karakasis et al., 2009a,b]. The idea of explicitly compressing the data involved in a memory-intensive application, such as the SpMV kernel, aims at reducing the pressure to the memory subsystem at the expense of the decompression computations. However, if the memory intensity of an application is mitigated by a higher available memory bandwidth, the decompression computations might be exposed and should be therefore reduced. In such a case, the computational part of the SpMV is also further exposed, an issue that must be addressed successfully in order to achieve high performance.

CSX is a compact storage format for sparse matrices that is able to detect and encode in a single representation a diverse set of matrix substructures. A *substructure* is any regular one- or two-dimensional sequence of non-zero elements inside the sparse matrix. The number of all the possible substructures detected from CSX is indefinitely large and a priori unknown for a specific sparse matrix. For this reason, CSX employs runtime code generation in order to provide high-performance SpMV implementations adapted to the specificities of every matrix. Compared to other storage formats that exploit a single substructure type, e.g., the BCSR format that exploits only dense block substructures [Im and Yelick, 2001], CSX is able to achieve consistent high performance by successfully adapting to a great variety of sparse matrices, ranging from regular ones to those with a rather irregular structure. The CSX format can accelerate the SpMV kernel more than 50% on average in symmetric shared memory systems, where the memory intensity of the kernel is more prominent. NUMA architectures are also supported efficiently by the CSX format, which is adapted to employ a less aggressive compression scheme, in order to mitigate the cost of decompression; in these architectures, CSX is able to provide a nearly 20% performance improvement over typical SpMV implementations.

A key design goal for the CSX format was to be practical for use within existing iterative solution methods. For this reason, we have spent considerable effort in minimizing the preprocessing cost of CSX, which consists of the cost of mining the matrix for substructures and the cost of constructing the final CSX format. Using a combination of statistical sampling for the input matrix, careful memory management and parallelization, we were able to shrink the full preprocessing cost to a few dozens of serial SpMV operations. Indeed, despite its preprocessing cost, CSX is able to accelerate the SpMV component of the Bi-CG Stabilized solver of the Elmer multiphysics software nearly 40% and offers a 15% average performance improvement to the overall solver.

An early version of CSX was initially conceived and presented by Kourtis [2010] as a first attempt to achieve higher compression ratios by exploiting linear substructures of the sparse matrix (horizontal, vertical, diagonal and anti-diagonal). In this thesis, CSX was substantially extended in order to become a viable high-performance approach for the optimization of SpMV in modern multicore architectures. A set of new features were implemented that allowed CSX to become one of the state-of-the-art storage formats for sparse matrices:

(a) *Support for two-dimensional substructures.* Two-dimensional substructures allow CSX to achieve compression ratios close to the theoretical maximum and also exhibit better computational characteristics.

(b) *Adaptive compression scheme.* This scheme allows the relaxation of compression in NUMA architectures, where the memory bottleneck is not so intense, by sparing processor cycles from the decompression cost.

(c) *Advanced substructure selection heuristic.* The new heuristic allows an efficient balancing of the compression benefit and the decompression overhead depending on the underlying architecture. In conjunction with the adaptive compression scheme, CSX is able to provide considerable performance benefits also in NUMA architectures.

(d) *Optimized preprocessing.* The preprocessing of the matrix needed to detect and encode the CSX substructures has been considerably reduced, allowing CSX to be used 'as-is' in the Elmer multiphysics software [Lyly et al., 1999–2000] and speedup the solver.

(e) *Enhanced runtime code generation.* The code generation module has been substantially revised in order to allow more straightforward and efficient implementations of the substructure-specific SpMV routines.

(f) *Support for symmetric matrices.* CSX supports also symmetric matrices leading to considerable gains in SpMV performance, due to the significant reduction of the matrix's memory footprint.

Efficient implementation for symmetric matrices

We introduce a variant CSX for symmetric matrices, capable of achieving nearly $3\times$ reduction in the matrix representation size. Multithreaded symmetric SpMV implementations, however, must cope with a RAW dependency on the output vector. In order to avoid the prohibitive cost of locking, the dependency is typically solved with the use of per-thread local vectors, which are reduced to the final output vector at the end of the SpMV computation. However, this reduction phase entails a significant performance overhead that increases linearly with the thread count, preventing SpMV from scaling. We address this problem by building an indexing scheme for local vectors that allows the selective reduction of the conflicting elements only, while all other update operations are pushed directly to the output vector. Our scheme effectively decouples the reduction overhead from the thread count, since the local vectors become sparser as the number of participating threads increases and, therefore, the index size is reduced. As a result, symmetric SpMV kernel scales as expected. Symmetric CSX in conjunction with our low-overhead reduction scheme, symmetric CSX was able to accelerate more than 2× the performance of the SpMV kernel.

Data compression beyond SpMV

The notion of data compression employed by CSX for the optimization of the SpMV kernel has broader repercussions in the modern multicore era. Every new processor generation not only augments the memory-processor speed gap, but also increases the memory-processor 'power gap'. Main memories are major contributors to the overall power dissipation of a computer system and memory-intensive applications tend to consume more energy than CPU-intensive ones [Kamil et al., 2008]. As a result, explicit data compression techniques are likely to become more mainstream in the future, as a means not only for mitigating the memory contention problem, but also for reducing the energy footprint of future HPC applications.

1.4.3 Toward an energy-efficient SpMV implementation

In this thesis, we take a first step in investigating the performance-energy tradeoffs of the SpMV kernel and propose a methodology for selecting the processor frequency and the thread placement that lead to optimal tradeoffs for the SpMV kernel. Based on the notion of the Pareto optimality, we show that there exist a set of optimal performance-energy tradeoffs and that the commonly used energy-delay products are indeed optimal solutions in the Pareto sense. The key goal of our methodology is not only to predict the best energy-delay configurations, but also provide a representative set of optimal tradeoffs. Based on the observation that matrices with similar structural characteristics have a similar performance and energy consumption behavior, we build our method on matrix clustering techniques and construct a representative set of optimal configurations for each cluster. Our prediction method is able to provide near-optimal sets of execution configurations for the SpMV kernel for a wide range of sparse matrices.

1.5 Outline

The rest of the thesis proceeds with a detailed and in-depth analysis of the involved subjects and our contribution.

Chapter 2 is a review of the storage formats for sparse matrices proposed in the literature. Starting from conventional and widely used formats, we pass through the multiple blocking storage variants to microarchitecture-specific formats and formats relying on explicit compression techniques. We present also symmetric matrix storage formats and discuss the major challenges of the symmetric SpMV kernel.

Chapter 3 presents an in-depth performance analysis of the SpMV kernel in modern multicore architectures. We highlight the possible performance impediments from an algorithmic perspective and proceed in studying thoroughly their impact in modern multicore architectures in single- and multi-threaded contexts. In this chapter, we present also the matrix suite and the experimental platforms and methodology we use throughout this thesis.

Chapter 4 focuses specifically on blocking storage formats and investigates their optimization opportunities. We study the compression capabilities and computational characteristics of each format and investigate their impact on modern multicore architectures. Based on this analysis, we propose two alternative and simple performance models for predicting the optimal block of BCSR.

Chapter 5 presents in detail the Compressed Sparse eXtended format. The presentation of the data structures and the matrix construction process is followed by a thorough performance evaluation, which confirms the superiority of CSX compared to other alternative storage formats in terms of absolute performance and performance stability across a variety of sparse matrices and architectures. The chapter closes with the benchmark results of the integration of CSX into the Elmer multiphysics simulation solver.

Chapter 6 presents our approach for optimizing the symmetric SpMV kernel. We start with a presentation and evaluation of the impact of the reduction phase optimization that allows SpMV to scale and continue with the CSX variant for symmetric sparse matrices. The chapter closes with an evaluation of our optimizations in the context of the CG iterative algorithm.

Chapter 7 departs from the purely performance-oriented view of the SpMV kernel and inserts the dimension of energy-efficiency. Starting with an introductory section on the fundamentals of processor power dissipation, we then investigate the performance-energy tradeoffs of the SpMV kernel in relation to the input matrix, the processor frequency and the thread placement. We finally propose a machine learning based approach for predicting the execution configurations (core frequency, thread placement) that lead to the optimal tradeoffs.

Chapter 8 concludes this thesis, summarizing its contribution and achievements, and discusses a future research orientation in the field of SpMV optimizations and HPC applications in general.

2

Storing Sparse Matrices

The density of a typical large sparse matrix, i.e., the ratio of its non-zero elements over the product of its dimensions, is well below 1% in most of the cases. Storing efficiently a sparse matrix in terms of storage space, therefore, requires to store only its non-zero values along with some location information for iterating over them. However, building a storage format for the efficient execution of the SpMV kernel is not a trivial task. The distribution of the non-zero elements and the underlying computer architecture play a significant role in the performance of this kernel, and an efficient storage format must adapt successfully to these requirements. Several storage formats for sparse matrices have been proposed for favoring the execution of the SpMV kernel, each one with own advantages and disadvantages.

In this chapter, we attempt a comprehensive description of the most widely used storage formats for sparse matrices and also present the most recent approaches to the optimization of the SpMV kernel through the use of advanced sparse matrix storage formats. We focus specifically on static generic formats, i.e., formats that can be used to store sparse matrices with any non-zero element structure, and do not take any particular measure for the dynamic insertion of new non-zero elements. Finally, although we present briefly two approaches that compress the non-zero values of the matrix, we focus mainly on storage formats that target the minimization of the non-zero element indexing structures.

2.1 Conventional storage formats

The most straightforward way to store a sparse matrix is to represent it as a sequence of (i, j, v) tuples, where v is the non-zero element's value and i, j are the corresponding row and column indices. This format is known as the *Coordinate (COO)* format [Tewarson, 1973; Pooch and Nieder, 1973; Duff and Reid,

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

`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
`values:`	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 2.1: The Coordinate sparse matrix storage format.

1979; Saad, 1992][1]. Figure 2.1 shows a typical implementation of the COO format. The `rowind` and `colind` data structures are typically 32-bit integers, while the non-zero values are double precision floating point numbers. Algorithm 2.1 shows an SpMV kernel implementation using this format. The main advantage of this format is its flexibility in inserting and deleting non-zero elements, since there is no restriction in the order of the elements. However, its key disadvantage for use in the SpMV kernel is its large memory footprint ($\approx 16NNZ$ bytes) and the abundance of indirect, and possibly irregular, references in both the input and output vectors during the execution of the kernel.

```
1: procedure MatVecCoo($A$::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:         $yi \leftarrow rowind[i]$
4:         $y[yi] \leftarrow y[yi] + values[i] \cdot x[colind[i]]$
5:     end for
```

Algorithm 2.1: Implementation of the SpMV kernel using the COO format.

The most widely used storage format for sparse matrices is the *Compressed Sparse Row (CSR)* format [Tinney and Walker, 1967; Pooch and Nieder, 1973; Duff and Reid, 1979; Saad, 1992]. This format eliminates the `rowind` data

[1] For the completeness of our description, we should note here that the early sparse matrix storage format nomenclature was 'standardized' by Saad [1992]. Earlier research works or surveys usually describe the different formats without coining specific terms.

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

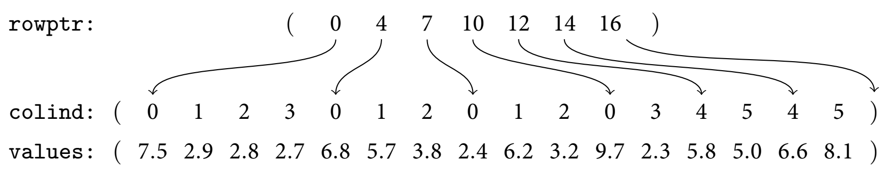

rowptr: (0 4 7 10 12 14 16)
colind: (0 1 2 3 0 1 2 0 1 2 0 3 4 5 4 5)
values: (7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1)

Figure 2.2: The Compressed Sparse Row storage format.

structure of the COO format, which stores the row indices explicitly, and replaces it with a set of pointers to the start of each row of the matrix. Figure 2.2 shows schematically a typical implementation of the CSR format. The `colind` and `values` arrays remain the same as in the COO format, while the `rowind` array is replaced by the `rowptr`, which stores the position of the first element of each row inside the `colind` and `values` arrays. For an $N \times M$ sparse matrix, the size of the `rowptr` array is $N + 1$ and its last element points always to the end of the `colind` and `values` arrays. Since $N \ll NNZ$ for the majority of the sparse matrices, the memory footprint of the CSR format is considerably reduced compared to the COO format ($\approx 12NNZ$), while its construction is also straightforward. This also constitutes the main advantage of CSR that has largely promoted its ubiquity: it is a compact and easy to use format. Nonetheless, CSR imposes a row-wise iteration order of the non-zero elements of the matrix, which implies a lexicographic sort of the elements based on their coordinates. The row indexing of CSR facilitates the random access of non-zero elements ($O(N)$ complexity), but complicates the insertion and deletion of non-zero elements. The column-wise counterpart of CSR is the *Compressed Sparse Column (CSC)* format. CSC stores the non-zero elements column-wise and keeps the row indices explicitly, while indexing the columns of the matrix.

Algorithm 2.2 shows a typical implementation of the SpMV kernel using the CSR format. The performance characteristics of this kernel and its optimization opportunities will be discussed in detail in subsequent chapters.

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

Algorithm 2.2: Implementation of the SpMV kernel using the CSR format.

2.2 Exploiting the density structure of the matrix

Despite being relatively compact, the CSR format has a lot of redundant information in its `colind` structure, which stores the column indices of the non-zero elements. The non-zero elements of sparse matrices arising in real-life applications expose some regularities in their structure. For example, some elements might be arranged in horizontal, vertical or diagonal sequences, while others might form small dense two-dimensional blocks. We will call each of these regularities in the structure of the matrix a *substructure*. Since the sequence of elements inside a substructure is known by default, one could keep a single column index per substructure for decsribing all its elements, therefore reducing the size of the `colind` array. This technique of grouping together neighboring non-zero elements and representing them by a single column index is also known as *blocking*. Apart from the apparent advantage of reducing the column indexing structure of the matrix, blocking comes with an important side-effect: it provides the means for further optimizing the computational part of the SpMV kernel [Im and Yelick, 2001; Karakasis et al., 2009b]. The inner loop of the SpMV kernel which computes the dot product between a row vector and the input vector (Algorithm 2.2, lines 3–5) does not proceed element-by-element, but rather substructure-by-substructure, exposing further the computational part of the kernel and allowing specific optimizations.

There has been proposed several types of blocking formats, each one exploiting a different type of substructures. However, we could distinguish two large categories of blocking: fixed and variable size blocking. In the following, we will describe these categories in more detail and present the most representative formats from each category. We will also present briefly some more specialized approaches.

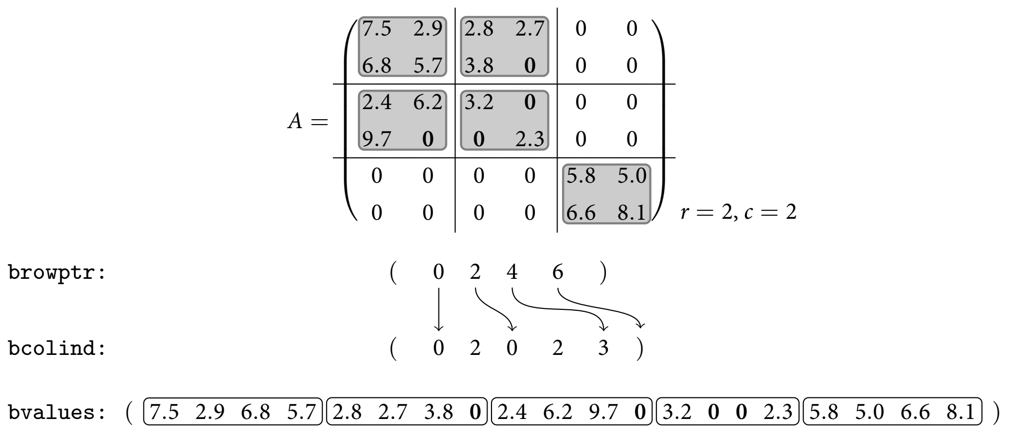


Figure 2.3: The Blocked Compressed Sparse Row storage format.

2.2.1 Fixed size blocking

Fixed size blocking methods exploit one- or two-dimensional substructures by trying to construct full fixed size blocks. The size and the dimensions of the blocks are fixed throughout the matrix, while padding with explicit zeros is used to construct full blocks. These methods involve an initial preprocessing step, during which the matrix is scanned for dense blocks and a decision is drawn as to what block size to utilize. This decision is based mainly on the minimization of the padding elements [Im and Yelick, 2001; Im et al., 2004], but other approaches consider furthermore the computational characteristics of the resulting code [Karakasis et al., 2009c].

The most representative storage format from this category is the *Blocked Compressed Sparse Row (BCSR)* format [Pooch and Nieder, 1973; Saad, 1992; Im and Yelick, 2001]. BCSR is essentially the blocked version of the standard CSR format, storing fixed size blocks instead of simple non-zero elements. Figure 2.3 shows an example implementation of the BCSR format. The `bcolind` structure stores the column index of the upper left element of each block, while the `browptr` now splits the matrix into block-rows. The non-zero elements are stored block-wise following a row-major order. BCSR imposes a strict alignment on the its blocks, requiring an $r \times c$ block to start at r-rows and c-columns boundaries. Although this alignment requirement contributes to a faster preprocessing, it usually introduces unnecessary padding. Alternatives to BCSR that relax these requirements have been proposed by Vuduc and Moon [2005].

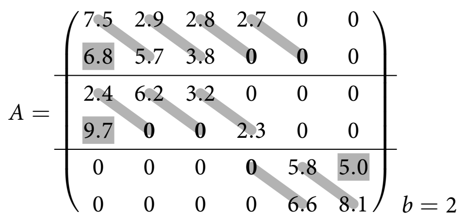
Algorithm 2.3 shows a typical implementation of the SpMV kernel using the BCSR format. The double loop in lines 7–11 computes the matrix-vector product for an $r \times c$ block. In practice, the block dimensions (r, c) are fixed and known during the compilation time, thus, the aforementioned loop can be optimized significantly with the use of common compiler optimization techniques, such as loop unrolling, vectorization etc. Indeed, the OSKI sparse matrix optimization library, which is an 'out-of-the-box' BCSR implementation, implements specific optimized versions for all block sizes up to 8 × 8, falling back to the generic implementation shown in Algorithm 2.3 for larger blocks [Vuduc et al., 2005].

```
1:  procedure MatVecBcsr($A$::in, $x$::in, $y$::out, $r$::in, $c$::out)
    $A$: matrix in BCSR format
    $x$: input vector
    $y$: output vector
    $r, c$: block dimensions
2:      $i_r \leftarrow 0$
3:      for $i \leftarrow 0$ to $N$ step by $r$ do
4:          for $j \leftarrow browptr[i_r]$ to $browptr[i_r + 1]$ step by $r \cdot c$ do
5:              $j_b \leftarrow \frac{j}{r \cdot c}$
6:              $x_0 \leftarrow bcolind[j_b]$
7:              for $k \leftarrow 0$ to $r$ do
8:                  for $l \leftarrow 0$ to $c$ do
9:                      $y[i + k] = y[i + k] + bvalues[j + k \cdot c + l] \cdot x[x_0 + l]$
10:                 end for
11:             end for
12:         end for
13:         $i_r \leftarrow i_r + 1$
14:     end for
```

Algorithm 2.3: Implementation of the SpMV kernel using the BCSR format.

A common substructure encountered in many sparse matrices are sequences of diagonal elements that do not necessarily lie on the main diagonal of the matrix. Due to its nature, BCSR cannot handle these substructures at all and inserts excessive padding. A common approach for applying blocking in such matrices is to segment the matrix into fixed size bands and try to construct diagonal full blocks using zero-padding. This format is known as *Row Segmented Diagonal (RSDIAG)* or *Blocked Compressed Sparse Diagonal (BCSD)*, similarly to BCSR [Agarwal et al., 1992; Vuduc, 2003; Karakasis et al., 2009a]. Figure 2.4 shows an example implementation of the RSDIAG format.

Despite a simple representation and a straightforward implementation with enough optimization opportunities, fixed size blocking methods fall victim of



$$A = \left(\begin{array}{cccccc} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & \mathbf{0} & \mathbf{0} & 0 \\ \hline 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & \mathbf{0} & \mathbf{0} & 2.3 & 0 & 0 \\ \hline 0 & 0 & 0 & \mathbf{0} & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{array}\right) \quad b = 2$$

Figure 2.4: The Row Segmented Diagonal storage format.

the zero-padding they employ to construct full blocks. It is often the case with matrices without the desired non-zero element pattern, e.g., matrices with a lot of diagonal elements in the case of BCSR, that the final memory footprint of the matrix exceeds considerably that of the baseline CSR format, leading even to considerable performance degradation. The size and the orientation of the constructed blocks play a significant role in reducing the final matrix size. It is therefore imperative that a preprocessing step exists for scanning the matrix and deciding on the best block to use. The use of heuristics is a common approach for selecting the most appropriate block size for the input matrix. Im and Yelick [2001] estimate the padding overhead of each candidate block by applying random uniform sampling of the matrix and combine this metric with an estimation of the performance of each candidate block, obtained after an offline benchmarking process. Vuduc et al. [2002] fine-tune this heuristic and provide upper and lower bounds on BCSR performance by modeling the cache behavior. In the next chapter, we will present our approach on selecting the best block size for BCSR, which apart from an estimation of the resulting memory footprint of the matrix, takes into account the computational characteristics of the candidate blocks.

Storage space requirements

Although fixed size blocking can reduce significantly the matrix footprint in case of matrices with a rather regular non-zero element structure, the compression potential of these methods remains limited. Suppose a $N \times N$ sparse matrix with NNZ non-zero elements that can be perfectly grouped in $r \times c$ blocks without padding ($r, c \ll N$) using BCSR. The size of the `bcolind` structure will then be $4\frac{NNZ}{r \times c} = \Theta(NNZ)$ bytes, still remaining in the order of NNZ, as is the case of CSR.

2.2.2 Variable size blocking

The variable size blocking methods avoid the insertion of padding elements by constructing blocks with variable size. To achieve this, they introduce ad-

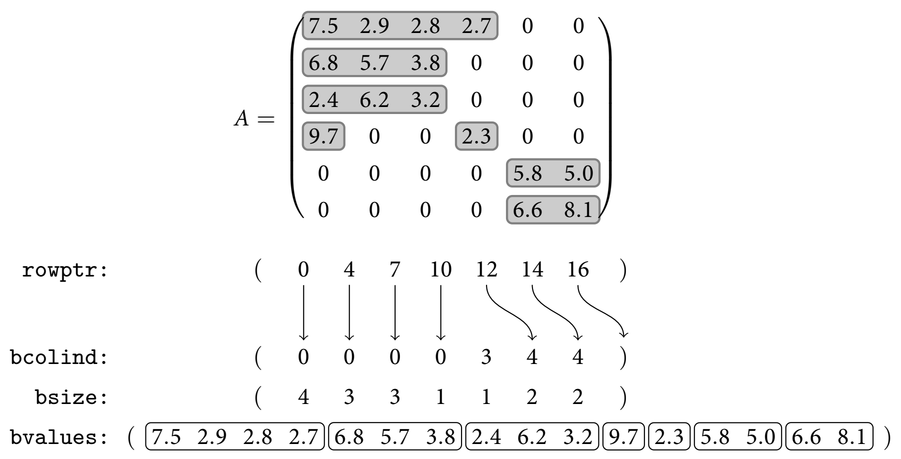

A =
7.5 2.9 2.8 2.7 0 0
6.8 5.7 3.8 0 0 0
2.4 6.2 3.2 0 0 0
9.7 0 0 2.3 0 0
0 0 0 0 5.8 5.0
0 0 0 0 6.6 8.1
rowptr: (0 4 7 10 12 14 16)
bcolind: (0 0 0 0 3 4 4)
bsize: (4 3 3 1 1 2 2)
bvalues: (7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1)

Figure 2.5: The Variable Block Length storage format.

ditional data structures that store either the size of each block or the starting column and row indices of each block. The total size of these additional data structures can be kept low with the use of short, one- or two-byte, integers.

The most representative formats in this category are the *Variable Block Length (VBL)* format, which exploits one-dimensional horizontal blocks, and the *Variable Block Row (VBR)* format, which exploits two-dimensional blocks [Saad, 1994; Pinar and Heath, 1999; Vuduc and Moon, 2005].

Figure 2.5 shows a typical implementation of the VBL format. The `values` and `rowptr` arrays are exactly the same as in the CSR format, since VBL constructs one-dimensional horizontal blocks only. The `bcolind` array holds the column index of the first element of each block, while `bsize` stores the size of each block. Since very large blocks are not a frequent encounter in sparse matrices, using a single byte to represent the block size in the `bsize` array is enough for most of the cases; in the case of a very large block, though, this is split into 255-element chunks. A characteristic of the VBL format is that all non-zero elements of the matrix are grouped into blocks; even strain non-zero elements form degenerate size-one blocks. It is obvious that for matrices with an irregular or non horizontally oriented non-zero element structure, the overhead of the block size structure can be significant for VBL, reducing its compression capability.

Algorithm 2.4 shows an implementation of the SpMV kernel using the VBL format. The algorithm is very similar to the CSR implementation, but it also keeps track on the block currently being traversed. Since the block size is not fixed, the algorithm must also keep track of the size of the current block. This adds additional operations to the algorithm and increases the inner loop overhead, a problem that becomes more prominent in the case of very small blocks.

```
procedure MatVecVbl($A$::in, $x$::in, $y$::out)
$A$: matrix in VBL format
$x$: input vector
$y$: output vector
    $i_b \leftarrow 0$
    for $i \leftarrow 0$ to $N$ do
        $j \leftarrow rowptr[i]$
        $s \leftarrow bsize[i_b]$
        while $j < rowptr[i+1]$ do
            $b_0 \leftarrow bcolind[i_b]$
            for $k \leftarrow 0$ to $s$ do
                $y[i] \leftarrow y[i] + values[j+k] * x[b_0 + k]$
            end for
            $j \leftarrow j + s$
            $i_b \leftarrow i_b + 1$
            $s \leftarrow bsize[i_b]$
    end for
```

Algorithm 2.4: Implementation of the SpMV kernel using the VBL format.

A step further from VBL is the VBR format, depicted in Figure 2.6, which is able to exploit variable size two-dimensional substructures. VBR splits the matrix into block rows and block columns with varying dimensions. The non-zero values are stored in the `bvalues` array in row-major block-wise order, while the `rowind` and `colind` arrays store the starting indices of each block row and block column, respectively. The `bvalptr` marks the start of each block in the `bvalues` array, while the `bcolind` stores the block column index by referencing the `colind` array. Finally, the `browptr` array marks the start of block rows by indexing the `bcolind` and `bvalptr` arrays.

Although VBR can detect arbitrary block substructures in the sparse matrix, the space and computational overhead of the supporting data structures can be excessive. For large sparse matrices, all the required indexing information will most probably require 32-bit integers, diminishing to a large extent the advantage of grouping non-zero elements into blocks. A compression of the VBR data structures would be achieved if the `rowind` and `colind` arrays stored the dimensions of the formed blocks, in which case, one-byte integers would be enough for these data structures. Even in this case, however, VBR has two more additional indexing data structures compared to the much simpler

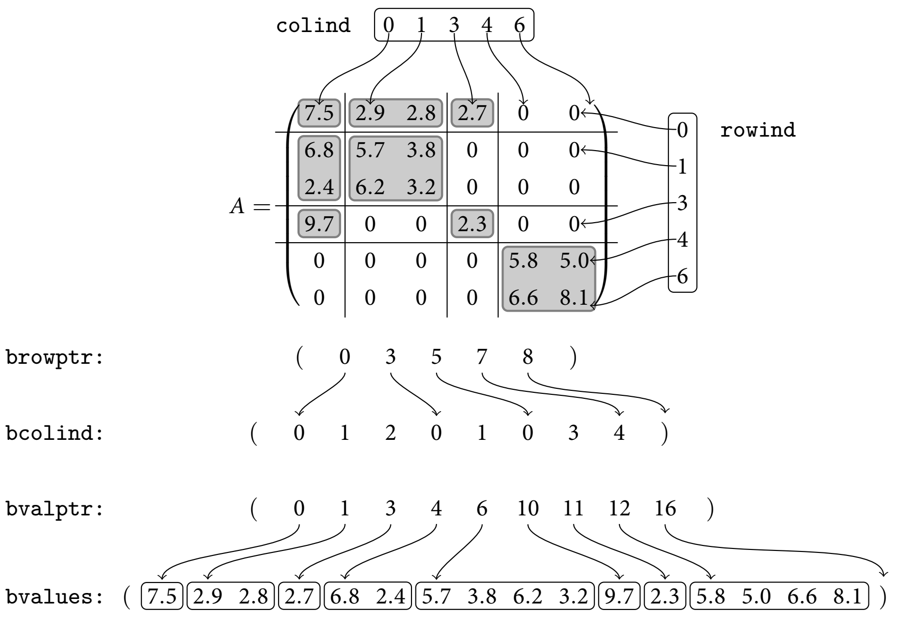


Figure 2.6: The Variable Block Row storage format.

VBL format. Futhermore, all this indexing metadata adds additional computational overhead, since multiple hops are needed to reach the actual non-zero values.

Storage space requirements

The compression potential of variable size blocking methods is higher than that of their fixed size counterparts, especially for methods without many additional data structures. Considering the case of VBL, suppose an $N \times N$ sparse matrix with NNZ_r elements per row and that k, $k \ll NNZ_r$, blocks are formed on average per row. Since in a typical sparse matrix, $NNZ_r \ll N$, k can be considered as constant. Therefore, the size of the compressed `bcolind` data structure will be $4kN$ and the required size for holding the block sizes will be kN (assuming one-byte size representations). These sum up to a size of $5kN = \Omega(N)$ bytes, which brings the total matrix size very close to the theoretical lower bound[2]. However, in sparse matrices with a non-horizontal non-zero element structure, VBL will form degenerate size-one blocks, and thus, the number of blocks per

[2] The theoretical lower bound for the size of a sparse matrix can be obtained if the indexing structures are completely eliminated. For example, suppose a fictional matrix that the location of its non-zero elements could be computed in the runtime. Since $N \ll NNZ$ for most matrices, compressing the indexing structures to the order of N can lead to sizes very close to the theoretical lower bound.

row cannot be decoupled from the non-zero elements per row. In this case, the size of `bcolind` will be in the order of *NNZ*. Nonetheless, this cannot overshadow the higher compression potential of the variable size block methods over their fixed size counterparts, which are restricted mainly by the large number of resulting blocks. In practice, VBL achieves almost always higher compression ratios than BCSR.

2.2.3 Other approches

The approaches we have already discussed exploit a single type of substructures in the sparse matrix leaving outside other types, leading to a performance degradation for 'unsuitable' matrices. A common approach for exploiting multiple substructures is to decompose the input matrix into multiple matrices, such that each one exploits a different substructure and their sum produces the initial matrix [Agarwal et al., 1992; Geus and Röllin, 2001]. The last addend is always a matrix in CSR format, containing the remainder elements that could not be grouped in a substructure. These formats avoid padding and therefore their memory footprint can be kept low. The matrix-vector multiplication using these formats consists of running the SpMV kernel for every addend and then accumulating the intermediate results to the final output vector. There are two possible performance problems with the decomposed formats, however. First, the accumulation of the intermediate results may introduce additional overhead and limit parallelism, second, the large segmentation of the non-zero elements among the multiple addends increases the sparsity of the involved matrices and can possibly lead to a deterioration of the SpMV performance.

The most recent approach to use the decomposition technique is the *Pattern Block Row (PBR)* format [Belgin et al., 2009]. PBR partitions the matrix into square blocks up to 8×8 and encodes the non-zero pattern in every block using a 64-bit bit vector. It then splits the matrix into multiple addends, each one storing a different block pattern; the remainder elements are stored in the last addend using the CSR format. Since the pattern structure is not known a priori, PBR generates in the runtime pattern-specific SpMV routines.

Another interesting approach in blocking storage formats is the *Compressed Sparse Block (CSB)* format [Buluç et al., 2009]. The motivation behind this format is the efficient support of both Ax and $A^T x$ matrix operations, the second being less efficient with the row-oriented common storage formats. For this reason, CSB divides the matrix into large sparse square blocks, which are stored with the Coordinate format using small integers for the row and column indices. In the runtime, it employs task parallelism to schedule the execution of the resulting blocks.

Condensing the non-zero values

The non-zero values consume 2/3 of the overall matrix size in the CSR format. The straightforward approach of using single-precision arithmetic is not an option in most real-life application, since 64-bit accuracy is important for the stability and the convergence of the iterative methods. The most common approach currently for compressing the non-zero values is to index them, i.e., store only the unique values and use an index for accessing them. The idea behind this technique is that the non-zero elements of the system's coefficient matrix often have the same value. Two approaches that exploit this property are the *Super-Sparse (SS)* storage format of Escudero [1984] and the *CSR Value Indexed (CSR-VI)* of Kourtis et al. [2008b]. The downside of non-zero value indexing is that the value index must be rebuilt every time a non-zero element is changed, thus rendering these formats less flexible.

Microarchitecture-specific formats

Although matrix- or architecture-specific formats are beyond the scope of this presentation, it is relevant to present briefly the *Ellpack-Itpack (ELL, ELLPACK or ITPACK)* storage format [Saad, 1992]. This format was initially conceived for vector processors [Oppe and Kincaid, 1987], but has come recently into the foreplay with the emergence of general-purpose GPU architectures. The ELLPACK format stores an $N \times M$ sparse matrix using two two-dimensional arrays (Figure 2.7). The `values` array is an $N \times K$ matrix storing the non-zero elements row-by-row with K being the maximum number of non-zero elements per row; rows with less elements are padded with zeros. The `colind` stores the corresponding column indices with irrelevant values in the place of inexistent elements. The advantage of the ELLPACK format is that it favors the streaming fetches of vector processors. On the other hand, matrices with an irregular non-zero element structure might add considerable amount of padding. Bell and Garland [2009] use a combination of the ELLPACK and COO formats for accelerating the performance of the SpMV kernel in modern GPU architectures, while Choi et al. [2010], inspired from BCSR, extend ELLPACK to store block rows.

Relevant to the ELLPACK format is, to some extent, the *Streaming CSR (S-CSR)* and *Streaming BCSR* (S-BCSR) formats proposed by Guo and Gropp [2011]. The motivation behind these formats is to allow the efficient use of the prefetch data stream hardware component of the IBM POWER processors. S-CSR splits the matrix into n streams and stores its rows in a cyclic way, such that row i is stored in stream $i \bmod n$. Each stream of rows is stored using the ELLPACK format. This S-CSR format and its blocked counterpart, S-BCSR,

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

$$\texttt{values:} \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 \\ 6.8 & 5.7 & 3.8 & 0 \\ 2.4 & 6.2 & 3.2 & 0 \\ 9.7 & 2.3 & 0 & 0 \\ 5.8 & 5.0 & 0 & 0 \\ 6.6 & 8.1 & 0 & 0 \end{pmatrix} \qquad \texttt{colind:} \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & * \\ 0 & 1 & 2 & * \\ 0 & 3 & * & * \\ 4 & 5 & * & * \\ 4 & 5 & * & * \end{pmatrix}$$

Figure 2.7: The Ellpack-Itpack storage format, suitable for vector processors and modern GPU architectures.

have the advantage of being able to trigger *n* independent prefetching streams, leading to a better utilization of the available memory bandwidth and a higher SpMV performance in this specific computer architecture.

2.3 Explicit compression of the matrix indices

An alternative approach for minimizing the memory footprint of a sparse matrix is the explicit compression of the CSR's `colind` data structure. Contrary to blocking, explicit compression techniques do not rely on the matrix structure directly, but treat the `colind` array as a common sequence of 32-bit integers and try to minimize its size. Due to the accentuation of the 'memory-wall' problem in modern multicore computer architectures, such approaches have gained an increasing interest recently [Willcock and Lumsdaine, 2006; Kourtis et al., 2008b].

The most common approach in explicitly compressing the `colind` data structure is the *delta indexing* or *delta encoding* of the non-zero elements column indices. Instead of storing the full column index of a non-zero element, delta indexing stores the delta distance from the column index of the previous element and keeps the full column index of the first element per row only [Pooch and Nieder, 1973]. The idea behind delta indexing for sparse matrices is that a delta distance is much more likely to fit in a small integer, allowing therefore a significant reduction in the size of `colind`.

A step further from delta indexing is the *run-length encoding* of the column

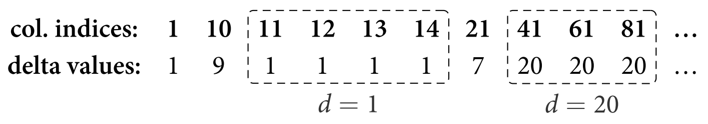

col. indices: 1 10 11 12 13 14 21 41 61 81 …
delta values: 1 9 1 1 1 1 7 20 20 20 …
$d = 1$
$d = 20$

Figure 2.8: Run-length encoding of the matrix column indices.

indices. Run-length encoding compresses delta value sequences by grouping together the same delta values and representing them as a two-element tuple containing the common delta value and the size of the sequence. Figure 2.8 shows schematically the notion of run-length encoding of the column indices.

Although delta encoding of the column indices has been discussed several years ago [Pooch and Nieder, 1973], it is not until recently that this technique has been utilized for the optimization of the SpMV kernel. Willcock and Lumsdaine [2006] apply delta encoding in the column indices and propose the *Delta-Coded Sparse Row (DCSR)* format. DCSR encodes the indexing information of the matrix (`rowptr` and `colind`) as a sequence of (*command, argument*) tuples. The argument is always a single byte and the commands—six in total—are responsible for regenerating the row and column index of the current non-zero element. The decompression process consists of reading and applying the command and then performing the appropriate computation with the corresponding non-zero value. The authors also propose a variation of DCSR that employs run-length encoding for detecting up to four contiguous non-zero elements. The downside of DCSR, however, is that the decompression cost is large, requiring non-portable implementations in order to amortize the cost and achieve high performance.

A simpler and more portable approach for delta encoding the column indices of a sparse matrix is the *CSR Delta Units* (CSR-DU) format of Kourtis et al. [2008b]. CSR-DU views the matrix as a sequence of *delta units*, defining three types of units depending on the bytes required to store a column delta distance (one-, two- or four-byte units). CSR-DU replaces both the `rowptr` and `colind` data structures with a byte sequence encoding the type, the initial column index and the delta distances of each unit. During the runtime, the CSR-DU code switches on the unit type in order to decompress correctly the delta distances and proceeds with the SpMV operations. CSR-DU is easily extended to encode horizontal sequences of non-zero elements of an arbitrary delta distance by applying run-length encoding [Kourtis et al., 2010]. With this extension CSR-DU can be viewed as a generalization of the VBL format, but in a more compact form.

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

$$\begin{tikzcd}
\texttt{rowptr:} & (& 0 \arrow[d] & 1 \arrow[d] & 3 \arrow[dr] & 3 \arrow[d] & 4 \arrow[d] & 5 \arrow[d] &) \\
\texttt{colind:} & (& 0 & 0 & 1 & 0 & 4 &) & \\
\texttt{values:} & (& 6.8 & 2.4 & 6.2 & 9.7 & 6.6 &) & \\
\texttt{dvalues:} & (& 7.5 & 5.7 & 3.2 & 2.3 & 5.8 & 8.1 &)
\end{tikzcd}$$

Figure 2.9: The Symmetric Sparse Skyline format.

2.4 Exploiting symmetry in sparse matrices

Finite element methods often involve the solution of large linear systems with sparse, structured, symmetric coefficient matrices. Exploiting the symmetry in non-zero element structure and values can reduce the memory footprint of the matrix to the half, alleviating significantly the pressure to the memory hierarchy of the underlying architecture. Several approaches have been proposed in the past targeting the optimization of the symmetric SpMV kernel.

The most common approach in storing a symmetric sparse matrix is the *Symmetric Sparse Skyline* (SSS) format [Eisenstat et al., 1982; Saad, 1992]. SSS is essentially the symmetric version of the CSR format; Figure 2.9 shows an example implementation. The values of the main diagonal, which is full in a well-defined symmetric problem, are stored separately in the `dvalues` array, while the strictly lower triangular matrix is stored using the standard CSR format.

The SpMV kernel for symmetric matrices consists of iterating once over the elements of the lower triangular matrix and performing the computations of both the lower and upper triangular elements at the same time. Algorithm 2.5 shows a typical SpMV implementation using the SSS storage format. The key issue in the performance of this kernel are the write operations in the output vector that are performed for every element of the upper triangular matrix (Algorithm 2.5, line 7). Despite being somewhat irregular, there is not a significant problem with the write operations in the serial execution of the kernel. However, the parallel execution of the kernel becomes problematic, because these operations insert RAW dependencies on specific output vector elements; Fig-

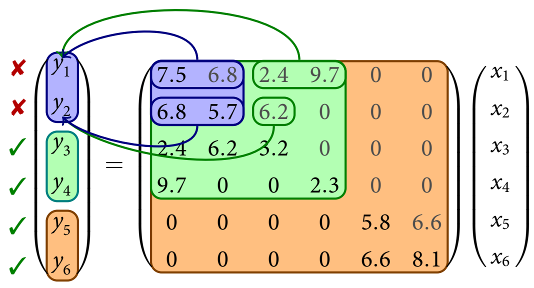


Figure 2.10: The RAW dependency on the output vector in a parallel execution of the symmetric SpMV kernel. The different thread partitions are shown in distinct colors.

ure 2.10 depicts this problem schematically.

```
1: procedure MatVecSss(A::in, x::in, y::out)
   A: matrix in SSS format
   x: input vector
   y: output vector
2:     for $i \leftarrow 0$ to $N$ do
3:         $y[i] \leftarrow dvalues[i] \cdot x[i]$
4:         for $j \leftarrow rowptr[i]$ to $rowptr[i+1]$ do
5:             $k \leftarrow colind[j]$
6:             $y[i] \leftarrow y[i] + values[j] \cdot x[k]$
7:             $y[k] \leftarrow y[k] + values[j] \cdot x[i]$
8:         end for
9:     end for
```

Algorithm 2.5: Implementation of the SpMV kernel using the SSS symmetric storage format.

The cost of protecting the access in the output vector with the use of locks can be prohibitive and it is not adopted in practice. The most common approaches in optimizing the symmetric SpMV kernel reside in local, per-thread output buffers, which are then reduced into the resulting output vector. The overhead of this reduction phase can also be significant, especially as the number of threads increases, and it becomes crucial to exchange the least possible information among the threads. The 'conflicting' elements of the output vector depend on the structure of the matrix and, more particularly, on the matrix *bandwidth*. The bandwidth a of sparse matrix is the maximum distance of its non-zero elements from the main diagonal; if the most distant element of the lower triangular matrix lies in the l-th diagonal and the most distant of the upper triangular lies in the u-th diagonal, then the bandwidth b of the matrix is defined as $b = l + u + 1$. It is easy to see that the higher the bandwidth, the higher the interaction between the threads of the symmetric SpMV kernel. Several algorithms have been proposed for reducing the matrix bandwidth, based chiefly on the permutation of the matrix rows and columns [Cuthill and McKee, 1969; George, 1971; Gibbs et al., 1976; Karypis and Kumar, 1995; Çatalyürek and Aykanat, 1999]. These algorithms utilize heuristics for selecting the best matrix reordering, since the matrix bandwidth minimization problem has been proved to be NP-complete [Papadimitriou, 1976].

Despite the reduction in the matrix bandwidth achieved by the reordering techniques, the execution of the SpMV parallel threads is not completely decoupled. There are also matrices with large bandwith, which cannot be significantly reduced with any reordering technique, keeping the thread interaction high. It is therefore necessary to overcome these interactions without limiting parallelism. The most common approach is to use local output vectors for each thread, which will then be reduced into the final output vector. Although this approach exhibits $\Theta(N)$ parallelism for an $N \times N$ matrix, in practice this method is limited mainly by the interaction of the final vector reduction operation with the memory hierarchy, especially for large vectors that do not fit in the system's cache. This problem becomes even more prominent if the vector data must be communicated over an interconnection network. Geus and Röllin [2001] examine ways for minimizing this cost by communicating only a small region of the local vector and overlapping the communication with useful computations. Similar is the approach of Batista et al. [2010] with their *Compressed Sparse Row-Column (CSRC)* format. The CSRC format is a hybrid of CSR and CSC for storing structurally symmetric matrices; the lower-triangular non-zeros are stored row-wise, while the upper-triangular ones are stored column-wise. The authors examine a number of techniques based on local vectors for solving the output vector dependency problem and also designate a solution for splitting the matrix in conflict-free partitions based on graph coloring.

An interesting approach for the parallelization of the symmetric SpMV kernel is that of Buluç et al. [2011]. The authors build on the CSB format (see Section 2.2.3) and provide an efficient parallel algorithm for the symmetric SpMV kernel that exhibits a lot of parallelism. The key idea is not to use a static thread partitioning scheme, but to employ task parallelism at the granularity of the CSB blocks instead. After having reduced the bandwidth with a reordering algorithm, the authors split the matrix in three block diagonals and then proceed to compute the matrix-vector product. Each of the three block diagonals is assigned a local vector on which it computes the partial matrix-vector product in two distinct parallel phases. Since each block diagonal is assigned a private vector, the SpMV computation can proceed in parallel for all the block diagonals, while the remainder elements, i.e., elements not contained in the three block diagonals, use atomic operations for updating directly the output vector. Finally, the private vectors are accumulated in parallel to the output vector. Apart from the use of task parallelism, which can offer a better load balancing, the key advantage of this method is that it decouples the number of intermediate vectors from the number of threads, therefore stabilizing the cost of the final reduction step.

3

The Performance of the Sparse Matrix-Vector Kernel

The Sparse Matrix-Vector Multiplication kernel (SpMV) lies at the core of iterative solution methods for sparse linear systems. As discussed in Chapter 1, the SpMV kernel dominates the execution time of these methods in modern multicore architectures and its efficient implementation and optimization has been of key concern in the high-performance computing community. The key performance problem of the SpMV kernel stems primarily from its algorithmic nature that imposes a very low arithmetic intensity. Unfortunately, this is not the sole problem of this kernel as other factors, relating mostly to the structure of the input sparse matrix, can be of definite importance.

In this chapter, we investigate the different performance problems of the SpMV kernel that have been reported in the literature and provide a quantitave evaluation, in order to better characterize the importance of each possible performance-limiting factor. We examine both symmetric shared memory and NUMA architectures and propose a simple and efficient technique for building NUMA-aware versions of the SpMV kernel.

3.1 An algorithmic view

The SpMV kernel poses a variety of possible performance bottlenecks that we will first present and discuss from a more theoretical point of view. For the completeness of our presentation and in order for the reader to easily follow the subsequent discussion, we repeat here the SpMV algorithm using the CSR storage format (Algorithm 3.1), presented in detail in Chapter 2. CSR is the most widely adopted storage format for sparse matrices and we will use this format as a baseline reference throughout the text. We have deliberately altered slightly the SpMV algorithm presented here to better match a real implementation, where the computed products in a matrix row are not directly accumulated in the corresponding output vector element, but in a local variable, typically kept in a register.

```
1: procedure MatVecCsr($A$::in, $x$::in, $y$::out)
   $A$: matrix in CSR format
   $x$: input vector
   $y$: output vector
2:     for $i \leftarrow 0$ to $N$ do
3:         $y_i \leftarrow 0$
4:         for $j \leftarrow rowptr[i]$ to $rowptr[i+1]$ do
5:             $y_i \leftarrow y_i + values[j] \cdot x[colind[j]]$
6:         end for
7:         $y[i] \leftarrow y_i$
8:     end for
```

Algorithm 3.1: Implementation of the SpMV kernel using the CSR format (see Chapter 2, Section 2.1 for a detailed description).

A number of algorithmic characteristics of the SpMV kernel have been identified as possible performance bottleneks in the literature. In the following, we discuss these characteristics from a theoretical standpoint.

Low arithmetic intensity The term *arithmetic intensity*, also known as *flop:byte ratio*, is a an algorithmic metric for denoting the amount of useful arithmetic operations performed by the processor per the amount of data necessary for performing these operations [Harris, 2005; Williams et al., 2009]. This metric is purely algorithmic in the sense that it does not account for hardware side effects, such as cache line evictions, which might incur additional memory traffic. With the advent of multicore and manycore architectures the arithmetic intensity metric is becoming increasingly important, as the speed gap between the processor and the main memory is continuously growing. The higher the flop:byte ratio of a computational kernel, the higher is the potential of an efficient utilization of the processor. Conversely, a low flop:byte ratio denotes that the kernel will be most likely bound from the memory subsystem.

Matrix-vector products, either dense or sparse, exhibit a rather low flop:byte ratio due to their streaming nature: the algorithms proceed in fetch and compute phases exhibiting no temporal locality in the accesses of the matrix elements. Assuming the typical layout of using 8-byte double precision floating point values for the non-zero elements, the flop:byte ratio r_{dense} of the matrix-vector product for an $N \times N$ dense matrix stored in the typical dense format can be easily calculated as

$$r_{dense} = \frac{2N^2}{8N^2 + 16N} = \frac{1}{4 + \frac{8}{N}} \approx 0.25 \qquad (3.1)$$

This is a fairly low ratio, since four bytes must be fetched for every floating point operation. Assuming an architecture with 64-bit words, this ratio implies that the memory hierarchy must be able to provide data to the processor at the half of its speed, which is hardly ever the case for modern high-performance mutlicore architectures.

The case with the sparse matrix-vector product becomes worse, since the additional indexing data must also be fetched in order to access the actual non-zero element information. Assuming 4-byte integers for the indexing information, the flop:byte ratio r_{sparse} for an $N \times N$ sparse matrix with NNZ non-zero elements stored in CSR can be calculated as follows:

$$r_{sparse} = \frac{2NNZ}{\underbrace{8NNZ}_{values} + \underbrace{4NNZ}_{colind} + \underbrace{4N}_{rowptr} + \underbrace{16N}_{x+y}} = \frac{1}{6 + 10\frac{N}{NNZ}} \approx 0.167 \qquad (3.2)$$

This ratio deteriorates further for very sparse matrices, where N is at the order of NNZ, since the size of the `rowptr` array becomes also significant. As a comparison, the arithmetic intensity of a multigrid PDE stencil kernel ranges between 0.33 and 0.50, that of a 3D FFT reaches 1.64 [Williams et al., 2009], while the arithmetic intensity of the matrix-matrix multiplication is at the order of N. It is clear from this analysis that the performance of the SpMV kernel in modern multicore architectures is expected to be bound from the performance of the memory subsystem, and especially the main memory bandwidth [Mellor-Crummey and Garvin, 2004; Buttari et al., 2007; Williams et al., 2007; Goumas et al., 2008].

Irregular accesses in the input vector A key difference between dense and sparse matrix-vector kernels is that in the latter case, the access pattern in the input vector is not sequential, but it depends on the non-zero element structure of the matrix. This may result in an increased amount of cache misses in the input vector for matrices with a rather irregular non-zero structure [Im, 2000; Geus and Röllin, 2001; Pichel et al., 2004].

Indirect references The need to save space in the matrix storage dictates the use of additional data structures for storing the location metadata of the non-zero elements. These data structures not only decrease the arithmetic intensity of the kernel (see equation (3.2)), but also introduce additional operations, unrelated to the actual computation, and add considerable interference to the cache hierarchy [Pinar and Heath, 1999].

Short Rows The trip count of the inner loop of the SpMV kernel (Algorithm 3.1, lines 4–6) depends on the size of the corresponding row. Therefore, smaller rows are likely to incur a significant loop overhead that can overwhelm the useful computations [White and Sadayappan, 1997; Buttari et al., 2007]. Worse, the remedy of unrolling cannot be applied to the inner loop without an introspection of the matrix structure, since the trip count is unknown during compilation.

Another subtle implication on the performance of the SpMV kernel is that very short rows affect negatively the arithmetic intensity of the kernel. The $\frac{N}{NNZ}$ ratio in the denominator of equation (3.2) is actually the inverse of the average row size of the matrix. This leads to a monotonically increasing relation between the average row size and the flop:byte ratio, meaning that smaller row sizes lead to a lower arithmetic intensity of the kernel, therefore accentuating the bottleneck in the memory hierarchy.

Load imbalance The SpMV kernel can be parallelized easily across the rows of the matrix. Each thread takes a band of the matrix and proceeds independently with the matrix-vector computation. A suitable static partitioning scheme would split the matrix into partitions with roughly the same number of non-zero elements, in order to achieve a fair distribution of the computational load. For matrices with an irregular non-zero element distribution, however, this scheme may still lead to load imbalances, since the computational characteristics of the partitions may be quite different. For example, a thread operating on a denser partition is expected to be faster compared to a thread operating on a rather sparse one, which may suffer from a lower flop:byte ratio or from cache misses due to irregular accesses in the input vector.

3.2 Experimental preliminaries

Before proceeding with the quantitave evaluation of the SpMV kernel, it is essential to present the matrices, the test platforms and the experimental methodology we used for our benchmarking. We dedicate a separate section for this description, since we use this experimental setup throughout the thesis. If not stated differently, references to specific matrices and platforms will be resolved in this section and descriptions of performance measurements and metrics will refer to the material presented hereafter.

3.2.1 Matrix suite

The matrix suite used in our experiments consists of 30 matrices selected 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 sparse matrices in the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms. We have selected matrices for our experimental suite based on a number of criteria:

- *Variety:* The selected matrices are derived from a large variety of applications, including problems without an underlying 2D/3D geometry. Such problems usually lead to matrices with a more irregular structure and we expect to stress different aspects of the SpMV kernel.
- *Size:* Since SpMV is a bottleneck for the solution of large sparse linear systems, the selected matrices are large enough so as not to fit in the aggregate cache of a typical high-end multiprocessor system.
- *Algebraic criteria:* Half of the selected matrices are symmetric and positive definite. These matrix characteristics are required for the CG iterative method, which we use in later chapters for benchmarking purposes.

Table 3.1 details the characteristics of the matrices in our experimental matrix suite. Two thirds of the matrices are derived from problems with an underlying 2D/3D, 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.

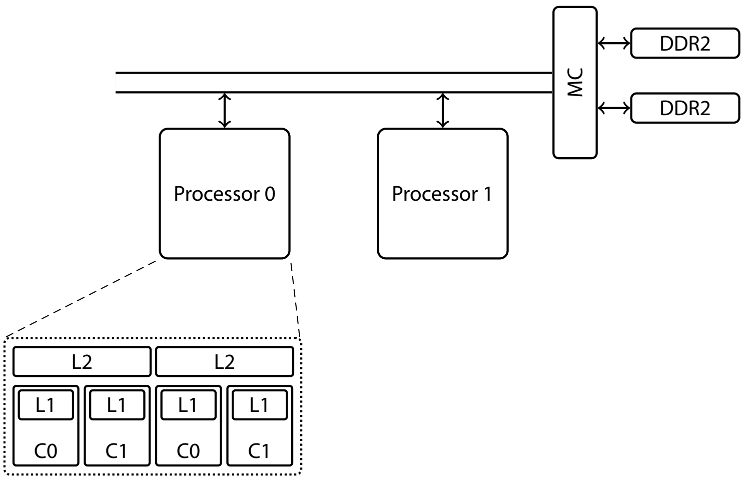
3.2.2 Hardware platforms

The hardware platforms we use for our quantitative analysis comprise of two symmetric shared memory (SMP) and one cache-coherent non-uniform memory access (cc-NUMA) multiprocessor systems. The SMP systems are a two-way quad-core Intel Xeon E5405 (codename *Harpertown*) and a four-way six-core Intel Xeon X7460 (codename *Dunnington*) multiprocessors. The NUMA system is a two-way quad-core Intel Xeon W5580 (codename *Gainestown*) multiprocessor. In the following, we will refer to each platform using its codename. Table 3.2 lists the technical specifications of our experimental platforms, while Figure 3.1 presents their block diagrams.

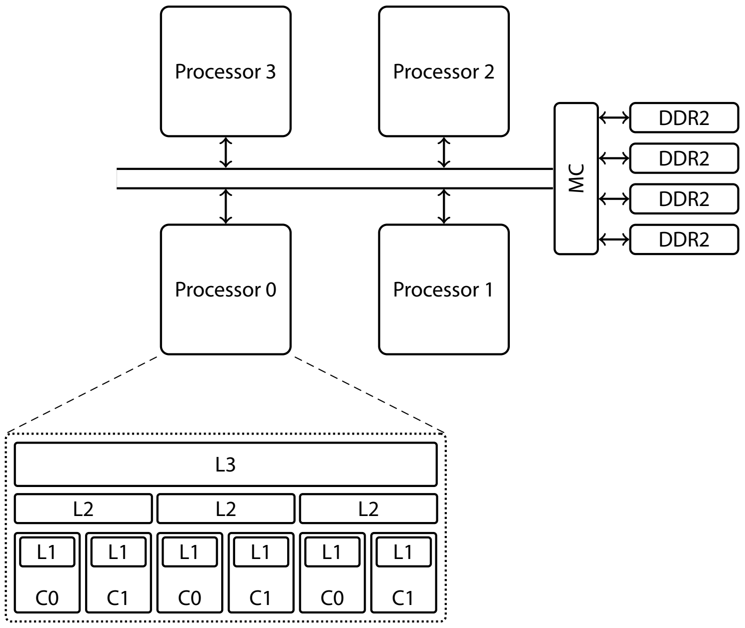
From an architectural point of view, the Dunnington system is an extension of Harpertown, not only system-wise but also within the socket. The Dunnington socket contains another dual core processor module, reaching a total of six

Matrix	Rows	Non-zeros	Size (MiB)	f:b ratio	P. D.	Problem	2D/3D
xenon2	157,464	3,866,688	44.85	0.156	No	Materials	Yes
ASIC_680k	682,862	3,871,773	46.91	0.129	No	Circuit Sim.	No
torso3	259,156	4,429,042	51.67	0.152	No	Other	Yes
Chebyshev4	68,121	5,377,761	61.80	0.163	No	Structural	Yes
Hamrle3	1,447,360	5,514,242	68.63	0.116	No	Circuit Sim.	No
pre2	659,033	5,959,282	70.71	0.141	No	Circuit Sim.	No
cage13	445,315	7,479,343	87.29	0.152	No	Graph	No
atmosmodj	1,270,432	8,814,880	105.72	0.135	No	C.F.D.	Yes
ohne2	181,343	11,063,545	127.30	0.162	No	Semiconductor	Yes
kkt_power	2,063,494	14,612,663	175.10	0.135	No	Optimization	No
TSOPF_RS_b2383	38,120	16,171,169	185.21	0.166	No	Power	No
Ga41As41H72	268,096	18,488,476	212.61	0.163	No	Chemistry	No
Freescale1	3,428,755	18,920,347	229.61	0.128	No	Circuit Sim.	No
rajat31	4,690,002	20,316,253	250.39	0.120	No	Circuit Sim.	No
F1	3,428,755	26,837,113	308.44	0.163	No	Structural	Yes
parabolic_fem	525,825	3,674,625	44.06	0.135	Yes	C.F.D.	Yes
offshore	259,789	4,242,673	49.54	0.151	Yes	Electromagnetics	Yes
consph	83,334	6,010,480	69.10	0.163	Yes	F.E.M.	Yes
bmw7st_1	141,347	7,339,667	84.54	0.161	Yes	Structural	Yes
G3_circuit	1,585,478	7,660,826	93.72	0.124	Yes	Circuit Sim.	No
thermal2	1,228,045	8,580,313	102.88	0.135	Yes	Thermal	Yes
m_t1	97,578	9,753,570	111.99	0.164	Yes	Structural	Yes
bmwcra_1	148,770	10,644,002	122.38	0.163	Yes	Structural	Yes
hood	220,542	10,768,436	124.08	0.161	Yes	Structural	Yes
crankseg_2	63,838	14,148,858	162.16	0.165	Yes	Structural	Yes
nd12k	36,000	14,220,946	162.88	0.166	Yes	Other	Yes
af_5_k101	503,625	17,550,675	202.77	0.159	Yes	Structural	Yes
inline_1	503,712	36,816,342	423.25	0.163	Yes	Structural	Yes
ldoor	952,203	46,522,475	536.04	0.161	Yes	Structural	Yes
boneS10	914,898	55,468,422	638.28	0.162	Yes	Model Reduction	Yes

Table 3.1: The matrix suite used for experimental evaluation. For each matrix, it is shown the row count (square matrices only), non-zero elements, size and arithmetic intensity (flop:byte ratio) in CSR format, positive-definiteness (P. D.), problem category and whether it is derived from a problem with an underlying 2D/3D geometry. For further matrix-specific properties, the reader is referred to the University of Florida Sparse Matrix Collection [Davis and Hu, 2011].



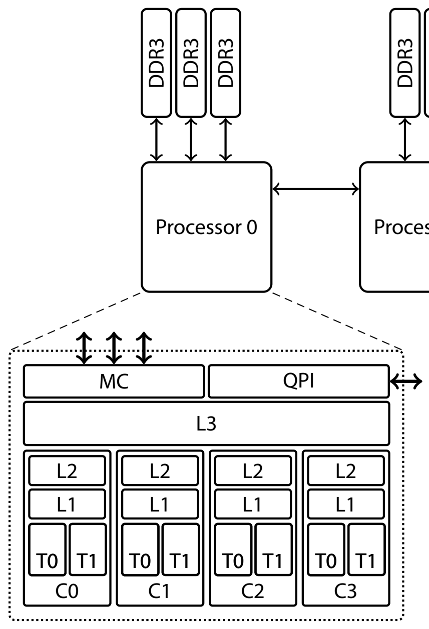
(a) Harpertown: two-way quad-core SMP system; two memory channels.



(b) Dunnington: four-way six-core SMP system; four memory channels.

Figure 3.1: The block diagrams of the multiprocessor systems used for the experimental evaluations in this thesis (continues at page 42).

cores, and a very large 16 MiB L3 cache, in order to reduce the contention in the front-end bus incurred by the plethora of cores. Indeed, thanks to its huge L3



(c) Gainestown: two-way quad-core NUMA system; two threads per core; three memory channels per integrated memory controller.

Figure 3.1 (Cont.): The block diagrams of the multiprocessor systems used for the experimental evaluations in this thesis.

cache, Dunnington can sustain a memory bandwidth very close to the theoretical peak. Nonetheless, the four Dunnington sockets are all using 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 system departs from the centralized SMP logic by moving the memory controller inside the socket and splitting, as a result, the physical memory into per-processor nodes. Each memory controller can serve up to three DDR3 memory channels leading to a sustained memory bandwidth of 15.5 GB/s, being almost three times faster than the one offered by the SMP architectures. Interprocessor communication, as well as remote memory accesses, are routed through a dedicated interconnection network (Intel QuickPath Interconnect – QPI [Kurd et al., 2008]) with a sustained bandwidth of 9.4 GB/s, being almost 40% less 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.

	Harpertown	**Dunnington**	**Gainestown**
Model	Intel Xeon E5405	Intel Xeon X7460	Intel Xeon W5580
Microarchitecture	Intel Core	Intel Core	Intel Nehalem
Clock freq.	2.00 GHz	2.66 GHz	3.20 GHz
L1 cache (D/I)	32 KiB/32 KiB	32 KiB/32 KiB	32 KiB/32 KiB
L2 cache	6 MiB *(per 2 cores)*	3 MiB *(per 2 cores)*	256 KiB *(per core)*
L3 cache	–	16 MiB	8 MiB
Cores/Threads	4/4	6/6	4/8
Peak Front-end b/w	10.7 GB/s	8.5 GB/s	2× 30 GB/s
Interconnection b/w	–	–	25.6 GB/s
Sustained Mem. b/w	5.8 GB/s	8.1 GB/s	2× 15.5 GB/s
Sustained i/c b/w	–	–	9.4 GB/s
Multiprocessor Configurations			
Sockets	2	4	2
Cores/Threads	8/8	24/24	8/16

Table 3.2: 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. The sustained interconnection (i/c) bandwidth for Gainestown was measured with a modified version of STREAM, where all the memory traffic was routed through the remote memory controller.

Software setup

All systems run a 64-bit version of the Linux OS (kernel version 2.6.30.5 or higher) 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 code are written using explicit, native threading with the Pthreads userspace library (NPTL, version 2.7). Finally, NUMA-aware versions are built with the *numactl* library (version 2.0.7), which is a wrapper userspace library of the low-level memory allocation system call interface of the Linux kernel.

3.2.3 Measurements procedures and policies

In order to guarantee reliable and fair measurements, we have built a common measurements framework for all the experiments. This framework communicates with the storage format implementations through a well-defined sparse matrix-vector multiplication interface. We measure the performance of 128 consecutive SpMV operations with randomly created input vectors. We make no attempt to artificially pollute the cache after each iteration, in order to better simulate the behavior of iterative scientific applications, where matrix and vector data are present in the cache hierarchy, either because they have just been produced or they have been recently accessed. We repeat every experiment three or five times and keep the median performance values.

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 non-zero elements. For formats with additional data structures, e.g., VBL, we will specify their size in the corresponding discussion inside the text.

Modern multicore processors have their resources (pipeline, cache hierarchy, bus/memory interfaces) shared at different levels among the threads of a multithreaded application. For example, two threads in Gainestown may share all the preprocessor resources from the pipeline up to the memory controller[1], but they may share nothing, if they are placed on different sockets. Depending on the application's workload, different assignments of threads to the cores may lead to considerable performance variations. It is therefore essential not only to pin threads to specific cores during measurements, but also define a policy for assigning threads to cores. We perform the thread pinning using the `sched_setaffinity()` Linux kernel system call, which allows to pin 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 (Hyperthreading feature[2]). 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 is the exact opposite policy: it 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.

In our evaluation and experiments, we routinely use the 'share-all' policy, unless otherwise stated.

[1] This technology is known as Simultaneous Multithreading (SMT) [Tullsen et al., 1995] and has been initially commercialized as *Hyperthreading* by Intel in its Netburst microarchitecture [Koufaty and Marr, 2003].

[2] We do not use the Hyperthreading feature by default, except for the 16-threaded configuration in Gainestown.

3.3 A quantitative evaluation

In Section 3.1, we have discussed a series of SpMV characteristics that can pose a performance overhead in the execution of the kernel. In the following sections, we perform a quantitave evaluation of these characteristics, in order to assess their real impact on the kernel's performance on a variety of matrices and modern mainstream computer architectures.

3.3.1 Single-threaded performance

In order to quantify the effect of the possible performance problems, we implement a series of microbenchmarks, each one alleviating an alleged performance problem. These microbenchmarks are actually 'stripped-down' versions of the CSR SpMV kernel, where we have eliminated the source of a problem, by applying simple code transformations. We apply these transformations incrementally and expect the performance of SpMV to increase, until it reaches the performance of dense matrix-vector (DMV) multiplication kernel, which is a straightforward upper bound of the SpMV performance. The more the performance of the transformed CSR is increased, the higher is the impact of the alleviated problem on the original SpMV kernel. The key requirement for all the microbenchmarks is that they must access all the non-zero elements and rows of the original matrix. There is no restriction on the order the elements are accessed or how many elements are accessed in a row. This allows to relax successively the performance limiting factors (e.g., accesses in the input vector, additional data structures etc.), while keeping the required data structure (non-zero elements) within the working set of the algorithm.

Table 3.3 presents a summary of the microbenchmarks implemented and their main goal. The noxmiss microbenchmark eliminates the irregular accesses in the input vector by serializing the access pattern. This microbenchmark does not transform the SpMV code; it only serializes the column indices in the `colind` data structure. The norowptr microbenchmark eliminates the use of the `rowptr` data structure by assuming each row has the same number of non-zero elements (equal to the average row size of the matrix), whereas the nocolind microbenchmark eliminates the use of the `colind` data structure by assuming a serial access pattern of the input vector within each row. Applying incrementally all these transformations eliminates every major performance problem related to the serial execution of the SpMV kernel. The transformed code can be considered as if it assumes a sparse matrix with a perfect access pattern for the input vector and a known arrangement of the non-zero elements, that eliminates the need of keeping additional data structures. The only difference of such a fictional matrix to a dense one is that it might contain short rows, since the row size remains the same with the original matrix's. We ex-

Benchmark	Goal	Action
noxmiss	Irregular accesses in the input vector	Serialize access pattern in `colind`
norowptr	Effect of the `rowptr` data structure	Rows with the same number of non-zero elements with a size equal to the average row size of the original matrix
nocolind	Effect of the `colind` data structure	Serial access pattern within each matrix row

Table 3.3: Microbenchmarks for assessing the performance problems of the SpMV kernel. Applying incrementally all the microbenchmarks gives also a measure for the effect of very short rows on the performance of the kernel (see discussion in text).

pect, therefore, the performance of the transformed code to approach that of the DMV kernel; any differences can be attributed to loop overheads due to very short rows of the original matrix.

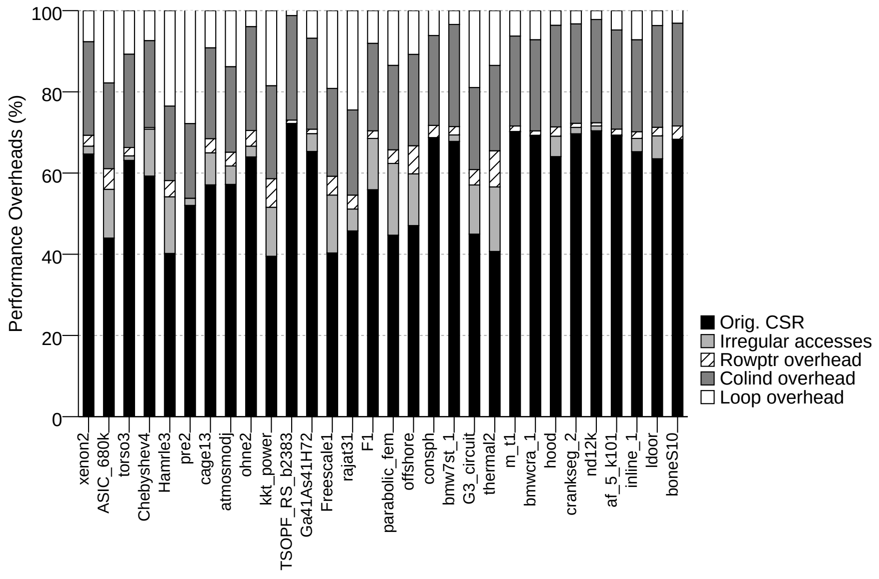
Figure 3.2 shows the performance overheads incurred by the special nature of the SpMV kernel. The very first observation is that the overhead of the `colind` data structure is quite important in SMP architectures exceeding 20% in most of the cases, being more prominent in Dunnington, due to its faster clock and lower memory bandwidth compared to Harpertown. The overhead of the `rowptr` structure is rather small, since it accounts for a very small amount of the total matrix size in most of the cases. The situation becomes more interesting with the unknown access pattern of the input vector. In fact, this pattern seems to be quite regular in most of the cases, especially for matrices with an underlying 2D/3D geometry (e.g., xenon2, boneS10 etc.). These regular accesses facilitate the hardware prefetching mechanism of modern microarchitectures, allowing it fetch the correct future data into the last level cache, increasing significantly the cache hit ratio. However, there exist a certain number of sparse matrices (e.g., Hamrle3, parabolic_fem) exhibiting a rather irregular access pattern. The SpMV computation proceeds back and forth in the input vector using rather large strides, not only ruining spatial locality, but also rendering impossible a successful prediction by the hardware prefetcher. For example, the accesses for the matrix Hamrle3 consist of non-constant strides of 4,480 up to 755,520 elements. We have also observed an important correlation between matrices with very short rows and an irregular access pattern. Matrices with very short rows usually come from non-structural problems (see Table 3.1) and their elements are scattered in small groups over the whole span of the row, leading to an irregular access pattern in the input vector. Finally, the loop overhead due to very short rows (4–9 elements) can be quite important in some cases, surpassing 20% (e.g., Hamrle3, pre2, rajat31 etc.), since only a few operations are performed inside the loop. The downside for such cases in SpMV is that it is not straightforward to apply common techniques, such as loop unrolling, in order to decrease the overhead, since the trip count of the loop is not only unknown, but may also vary within a single matrix.

The benchmark results in NUMA architectures (Figure 3.2c) exhibit a somewhat different behavior. The key observation here is that the `colind` overhead is not so prominent, due to the ample memory bandwidth offered by the integrated memory controller. In fact, the more regular matrices manage to achieve almost 80% of the DMV performance. Conversely, the `rowptr` overhead is rather important surpassing 10% for irregular matrices. Despite being a small data structure, the unknown loop bounds and the indirect references for accessing them restrain the compiler from generating highly optimized code for the inner loop. Given the high memory bandwidth available, this computational overhead is more exposed, while the loop overheads for matrices with very short rows are also more acute.

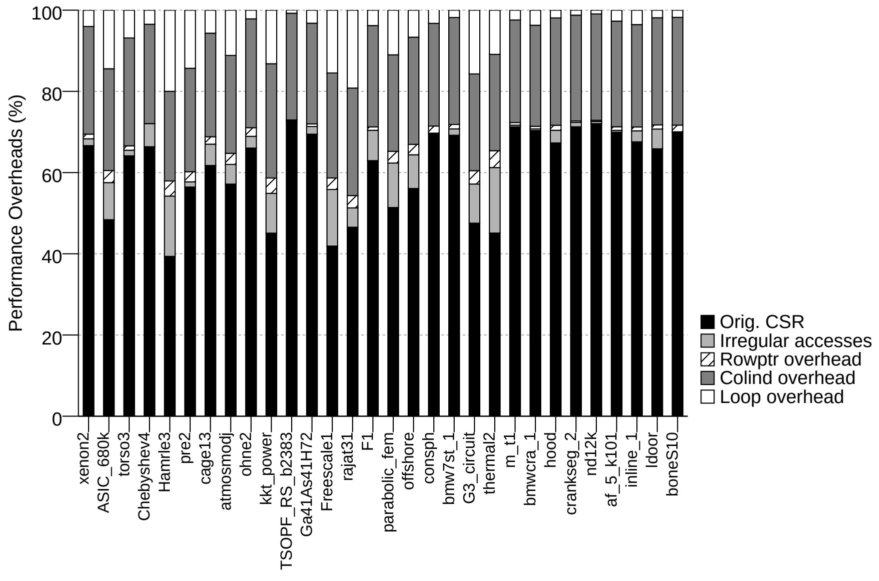
3.3.2 Multithreaded performance

The SpMV kernel exhibits ample parallelism [Buluç et al., 2011]. Indeed, if the sparse matrix is split row-wise and each thread is assigned a set of rows, the participating threads can proceed completely independently without any communication. The only concern is a well-balanced distribution of the work among the threads. Apparently, assigning each thread the same number of rows, as is the case for a dense matrix, is not an option for sparse matrices, since the distribution of non-zero elements is not always uniform. The best static load-balancing scheme, therefore, is to split the matrix row-wise so that each thread is assigned roughly the same number of non-zero elements. Unfortunately, despite the ample parallelism and a fair load-balancing scheme, SpMV does not scale.

Figure 3.3 shows the speedup of the SpMV kernel using the CSR format in the Harpertown and Dunnington systems using a 'share-nothing' core-filling policy (see Section 3.2.3). In Harpertown, SpMV stops to scale at four cores reaching a mere 1.8 speedup. The scaling in Dunnington is favored by the large L3 cache of each socket, which provide a total of 64 MiB system-wide aggregate cache. SpMV scales linearly up to two cores and continues to scale

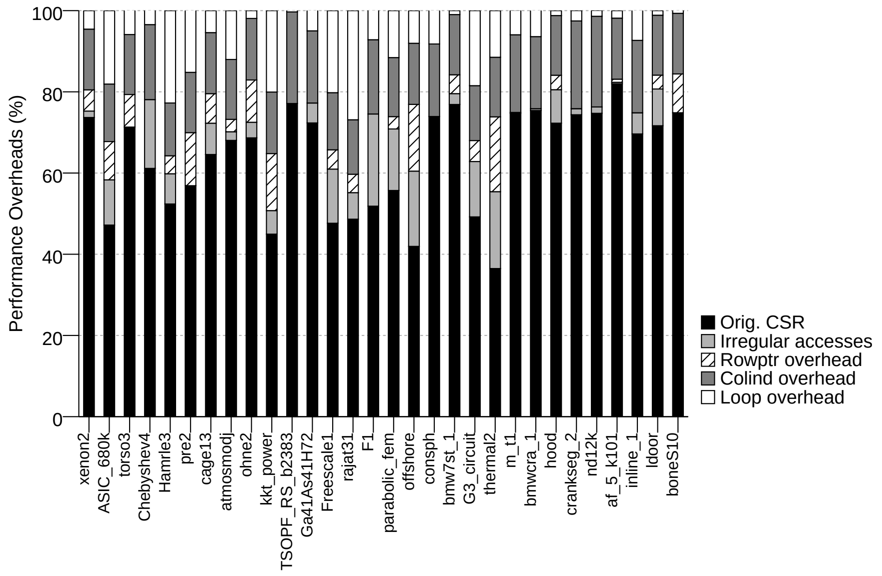


(a) Harpertown.



(b) Dunnington.

Figure 3.2: Performance overheads of the SpMV kernel due to its additional data structures and the unknown access pattern in the input vector (continues at page 49). *[The bars are produced by incrementally applying the microbenchmarks described in Table 3.3: "Irregular accesses" $\rightarrow$ noxmiss, "Rowptr overhead" $\rightarrow$ norowptr, "Colind overhead" $\rightarrow$ nocolind, "Loop overhead" $\rightarrow$ the remainder from* 100%*].*

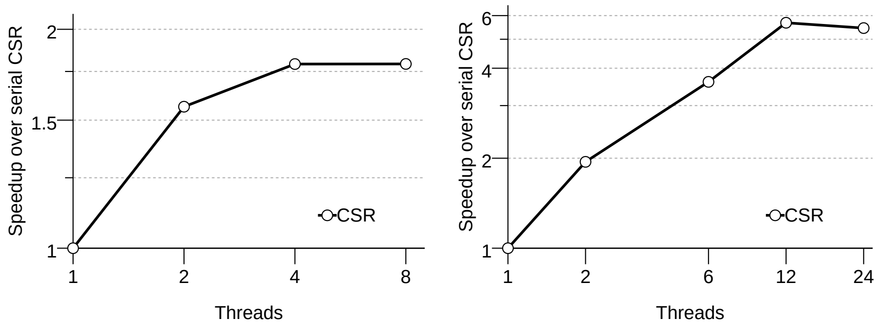


(c) Gainestown.

Figure 3.2 (Cont.): Performance overheads of the SpMV kernel due to its additional data structures and the unknown access pattern in the input vector. *[The bars are produced by incrementally applying the microbenchmarks described in Table 3.3: "Irregular accesses" → noxmiss, "Rowptr overhead" → norowptr, "Colind overhead" → nocolind, "Loop overhead" → the remainder from* 100%*].*

at almost half the rate up to 12 cores, where scaling stops abruptly, and SpMV experiences a performance slowdown as we move to the 24-threaded configuration. The key bottleneck in SMP architectures for the majority of matrices is the limited main memory bandwidth. All threads must contend for the shared front-end bus resources and the algorithm is implicitly 'serialized' at this point, since the threads must wait to be served. The inherently low flop:byte ratio of SpMV renders this problem inevitable and the kernel will eventually hit the 'memory-wall' sooner or later. Indeed, 85% of the available memory bandwidth in Harpertown is consumed at the two-threaded configuration and it is completely saturated from the four-threaded configuration onward. Similarly in Dunnington, large regular matrices consume 75% of the available bandwidth at the six-threaded configuration and saturate it at 12 cores.

In order to further demonstrate the effect of memory bandwidth, we present in Figure 3.4 the speedup achieved in Dunnington using a 'share-all' core-filling policy. According to this policy, we assign threads to cores so that the avail-



(a) Harpertown. (b) Dunnington.

Figure 3.3: The speedup of the SpMV kernel in two SMP systems using the 'share-nothing' core-filling policy that maximizes the utilization of the available memory bandwidth.

able sockets are filled successively. For example, the six-threaded configuration uses a single socket, while the 12-threaded uses only two. The difference in performance between the two policies is tremendous up to the 12-threaded configuration, since the threads not only contend for cache space (now only 16 MiB and 32 MiB are available, respectively), but also for access to the common front-end bus controller. The sustained bandwidth through a single FSB controller was measured at 2.6 GB/s and is almost completely consumed by a single thread, eliminating any possibility for scaling. When moving to the next socket, both the available aggregate cache and memory bandwidth are doubled, so does the speedup, which continues to increase rapidly up to the 24-threaded configuration.

It might has become clear already that the contention for memory bandwidth is a key scalability issue for the SpMV kernel in SMP architectures. Due to its streaming nature and very low arithmetic intensity, the kernel tends to saturate rapidly the available resources as we add more threads to the computation. This behavior results in an almost direct relation between SpMV performance and the matrix representation size, as it is depicted in Figure 3.5. In this figure, it is displayed the performance of BCSR for a variety of blocks for the relatively dense and block-dominated TSOPF_RS_b2383 matrix. While there is a tendency in the single-threaded configuration toward lower performance as the matrix representation size increases, the relation becomes completely linear as we add more threads stressing the common front-end bus. Matrix-wise, this relation is better depicted in Figure 3.6, where we have plotted the SpMV performance for every matrix in our suite against its corresponding arithmetic

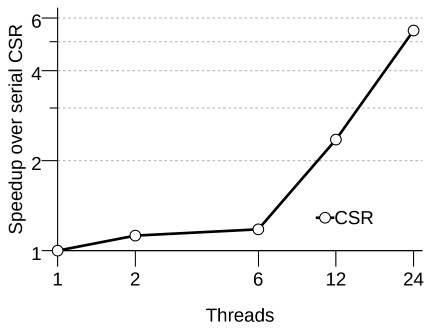
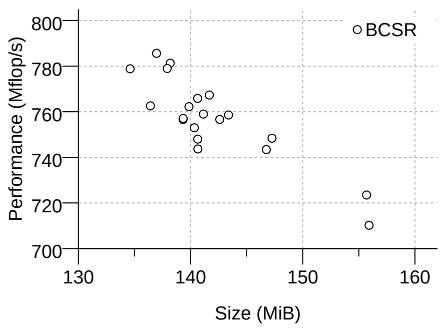


Figure 3.4: SpMV speedup in Dunnington using the 'share-all' core-filling policy: the memory bandwidth saturation is clear within a single socket (up to six threads), where almost no speedup is encountered.

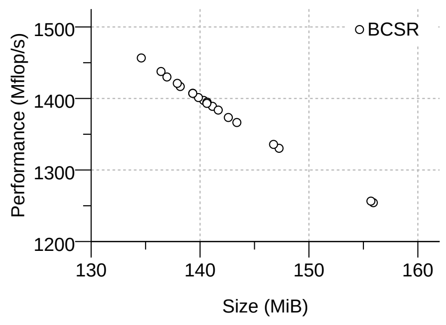
intensity (flop:byte ratio) for the single- and eight-threaded configurations in Harpertown. The trend toward higher performance as the flop:byte ratio increases, especially in multithreaded configurations, is a clear indication of the memory bandwidth contention in the SpMV kernel[3]. In fact, we can easily separate two distinct groups of matrices: 'low-performing' ones with a flop:byte ratio < 1.5 and 'high-performing' ones with a ratio ≥ 1.5. The variation in performance at the low-end is greater, since these matrices not only suffer from additional overheads (irregular accesses, short rows), but are also more likely to exhibit significant load imbalances, due to an irregular distribution of their non-zero elements.

Load balancing issues Splitting the input matrix statically based on the non-zero elements is a reasonable load balancing scheme for SpMV and can provide a fair distribution of the work among the threads. Nonetheless, as depicted in Figure 3.7, this scheme seems to be inadequate for some matrices with a very irregular non-zero element structure. The key reason for such behavior is that this scheme does not take into account the distribution of the non-zero element within a thread partition. For example, two partitions may have the same number of non-zero elements, but different access patterns in the input vector with one being very irregular; SpMV will be rather slow in this partition, hampering the whole multithreaded execution. The most typical examples of this situation are the Hamrle3 and G3_circuit matrices. Figure 3.8 shows the variation of the flop:byte ratio across these matrices. In both cases the distribution of non-zero elements is rather irregular with Hamrle3 being very sparse at the

[3] Williams et al. [2009] also indicate this behavior in their Roofline performance model, where the performance of memory bandwidth bound applications increases with the flop:byte ratio.

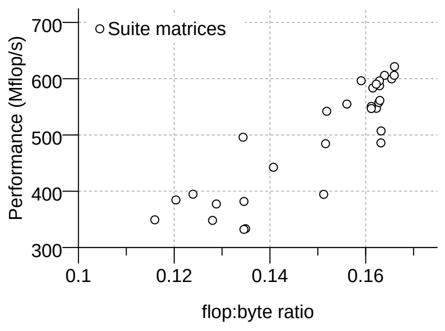


(a) Harpertown, single-threaded.

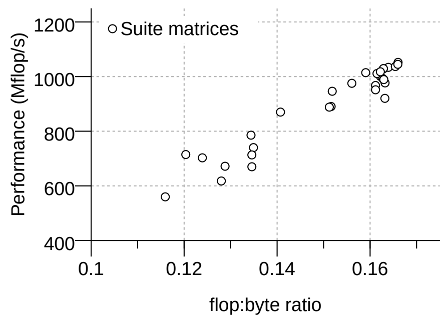


(b) Harpertown, eight-threaded.

Figure 3.5: The contention in the memory bus of SMP architectures, as more threads are added to the computation, renders the performance of the SpMV kernel very sensitive to the matrix representation size. Results shown are from the BCSR format using different block sizes for matrix TSOPF_RS_b2383.



(a) Harpertown, single-threaded.



(b) Harpertown, eight-threaded.

Figure 3.6: Performance of the SpMV kernel in relation to the arithmetic intensity (flop:byte ratio). The tendency toward higher performance as the flop:byte ratio increases becomes more prominent in multithreaded configurations, indicating the bottleneck in memory subsystem.

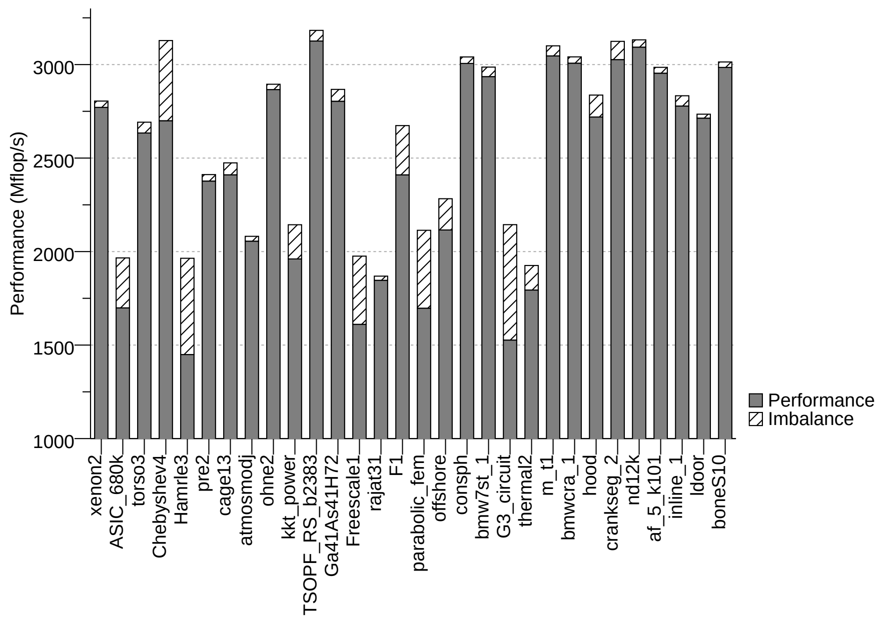
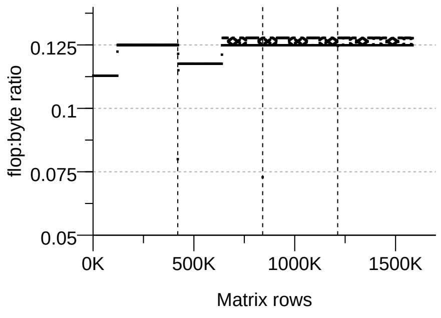


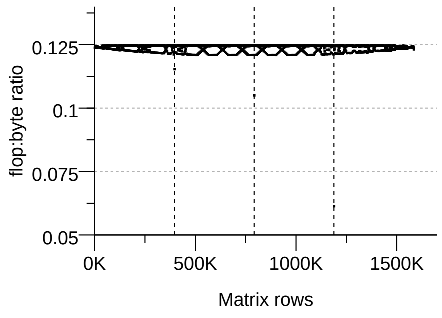
Figure 3.7: Load imbalance in the SpMV kernel. Matrices with an irregular distribution of non-zero elements suffer from significant load imbalances, despite the 'fair' non-zeros-based static balancing scheme. Results shown are from a single socket (four threads) in Gainestown.

latter partitions. A closer inspection at the structure of Hamrle3 reveals that the first half of the matrix is quite regular, while the rest is very irregular with large and variable distances between the non-zero elements of a row. G3_circuit has more uniform non-zero structure; however, the first partition is burdened with a more sparse structure leading to lower arithmetic intensities and increased loop overheads.

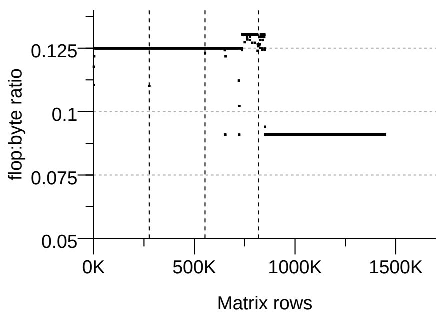
Matrix bandwidth minimization techniques (see Section 2.4) provide a common way for homogenizing the non-zero element distribution of a sparse matrix. These techniques try to bring all the non-zero elements as close to the main diagonal as possible by reordering the matrix rows and columns. While initially conceived for balancing the amount of communication data (input and output vector) in distributed memory SpMV implementations, they prove to be beneficial also in modern multicore architectures. Moving the non-zero elements toward the main diagonal generates a more regular access pattern in the input vector with smaller and more predictable strides, while the objective of a balanced communication creates a more homogeneous non-zero element distribution across the matrix. Figure 3.8 shows the flop:byte ratio variation for the G3_circuit before and after the application of the reverse Cuthill-McKee



(a) Matrix G3_circuit.



(b) Matrix G3_circuit (reordered).



(c) Matrix Hamrle3.

Figure 3.8: Variation of flop:byte ratio across the matrix. Partitions are marked with the vertical dashed lines. The different arithmetic intensity across the partitions leads to considerable load imbalances. Matrix reordering techniques homogenize the matrix structure and balance the computations, but they operate only on structurally symmetric matrices; Hamrle3 is not.

(RCM) matrix reordering algorithm [Cuthill and McKee, 1969][4]; the distribution of the non-zero elements is now completely homogenized across the matrix and the load imbalance has been eliminated, as depicted in Figure 3.9. Another important aspect of matrix reordering illustrated in this figure is that the gain in performance is not only due to the better balanced computations; the more regular access pattern in the input vector, as a result of the lower matrix bandwidth, provides also a significant performance benefit to the SpMV computation.

[4] RCM, as well as other matrix reordering algorithms, e.g., METIS [Karypis and Kumar, 1995], operates on structurally symmetric matrices. Therefore, it is not possible to reorder the Hamrle3 matrix in our example.

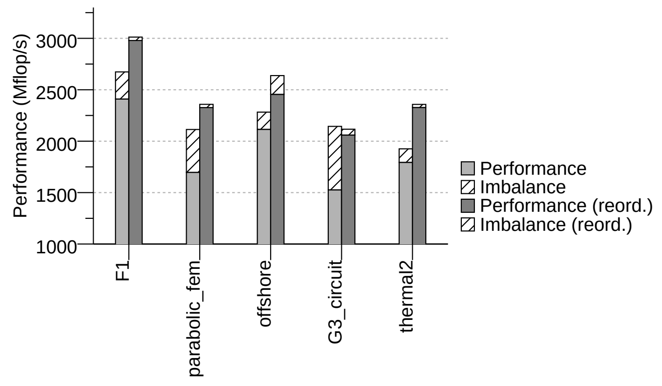
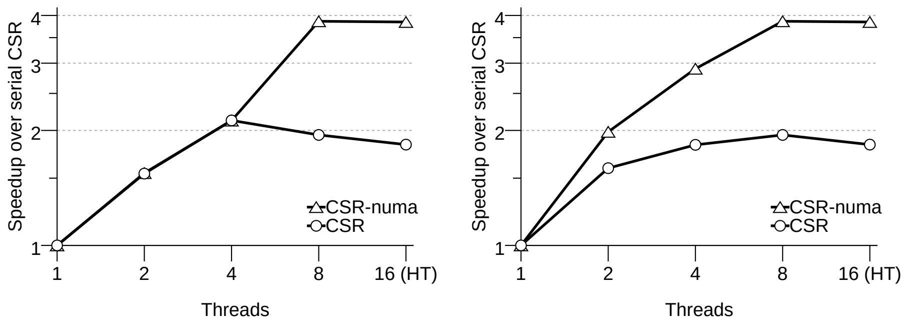


Figure 3.9: The benefit of matrix reordering. The balanced computations and the more regular matrix structure, allowing a better reuse of the input vector, lead to a significant increase in the SpMV performance.

NUMA architectures

The behavior of the multithreaded SpMV kernel in NUMA architectures stems also from the streaming nature and the very large working set of the algorithm. In a NUMA system (Figure 3.1c), each processor has its own integrated memory controller, which is responsible for accessing a specific partition (node) of the main memory. All memory accesses originated from a processor to its local memory node are served by its dedicated memory controller, while all accesses to remote nodes are routed through the memory controller of a peer processor, using a high-speed processor interconnection network or link. In the case of Gainestown, the sustained bandwidth of the interconnection link is significantly lower than the memory bandwidth sustained by an integrated memory controller (see Table 3.2). Remote accesses, therefore, are likely to saturate the interconnection network, limiting the speedup of a parallel application.

The key in achieving high performance for a streaming application in a NUMA system is the correct placement of the involved data on the available memory nodes, in order to minimize the remote memory accesses and avoid any bottlenecks in the interconnection link. This is better illustrated in Figure 3.10 that shows the scaling of the SpMV kernel in Gainestown using our two core-filling policies for the typical and NUMA-aware implementations. The typical implementation does not care about the placement of the SpMV data on the memory nodes and relies on the operating system for the correct placement. Linux uses a copy-on-write allocation of physical pages and will allocate a physical page on the memory node where the thread that writes first to the corresponding virtual page is running. Therefore, in the typical SpMV implementation, the allocation of the required physical pages will take place during



(a) Share-all policy. (b) Share-nothing policy.

Figure 3.10: The speedup of the SpMV kernel in the Gainestown NUMA system. The effect of data placement is very important, since remote memory accesses can easily saturate the interconnection link of the processors. The '-numa' suffix denotes a NUMA-aware data placement, while 'HT' in the thread configurations denotes the activation of the Hyperthreading feature.

the algorithm's initialization, where the matrix and the associated vectors are constructed and initialized. This phase is usually single-threaded, therefore all SpMV data will lie on a single memory node. As a result, if an SpMV computation thread is assigned to a different node, all of its memory accesses will be routed through the interconnection link, possibly saturating it.

Using the 'share-all' policy (Figure 3.10a), threads are assigned to a single memory node (socket) up to the four-threaded configuration, while the eight- and 16-threaded configurations use both memory nodes. The substantial memory bandwidth offered by the integrated memory controller allows SpMV to scale well in the single-socket configuration, achieving a 2.1 speedup, significantly overwhelming not only the single-socket SMP configurations, but also the full system speedup in Harpertown. The two-threaded configuration consumes 70% of the available memory bandwidth, whereas the four-threaded saturates the controller. When starting using the second socket, however, the data placement plays a significant role in the performance of the kernel. The typical SpMV implementation encounters a significant performance slowdown reaching 14% when the full system is utilized, while the NUMA-aware version continues to scale significantly up to the eight-threaded configuration, where the both memory controllers are saturated. Examining closer the individual thread execution times in the typical SpMV implementation for the eight-threaded configuration (Figure 3.11) reveals a significant load imbalance between the threads assigned to the local and remote sockets. In fact, a performance difference of approximately 45% is observed, matching the differ-

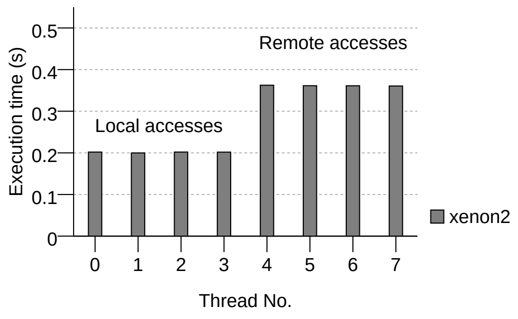


Figure 3.11: The saturation of the interconnection link in NUMA architectures increases the execution times of threads performing remote accesses, leading to considerable load imbalance. The SpMV performance is therefore determined by the link bandwidth. Results shown are for the xenon2 matrix, but are similar for all other regular matrices.

ence in sustained bandwidth of the memory controller and the interconnection link, denoting the contention in this link.

The speedup diagram of the 'share-nothing' core-filling policy (Figure 3.10b) reveals better the difference in the bandwidth between the memory controller and the interconnection link. The behavior of the typical CSR implementation resembles that of an SMP system. Half of the memory traffic in the multithreaded configurations must pass through the interconnection link, 80% of which is already consumed from the four-threaded configuration, leading to a less than $2\times$ maximum speedup at the eight-threaded configuration. The behavior of the NUMA-aware implementation is completely opposite: the ample memory bandwidth ($\approx$30 GB/s) offered by the two integrated memory controllers allows SpMV to scale significantly up to the eight-threaded configuration, achieving even linear speedup for the two-threaded configuration, where contention is not encountered at all.

Transparent data placement in NUMA architectures

The discussion on the performance results of SpMV in NUMA architectures has revealed the correct placement of the SpMV data as a key issue in achieving high performance. Such placement requires each thread to allocate its own data (matrix and output vector partitions, input vector) to its local node. However, such allocation needs to modify the SpMV algorithm itself. Essentially, the matrix must be split into individual submatrices per thread, and the SpMV computation must be modified to operate on these. Similarly, the output vector must be either split in subvectors or allocated as a whole on a single memory node. These modifications for converting to NUMA-aware an SMP-based code require a significant code refactoring and impose an important programming burden, although only the placement of the algorithm's data must be changed.

We address the data placement problem by developing a custom low-level allocation scheme. This scheme allows to maintain the SMP 'look-and-feel' of the SpMV kernel by keeping the data placement completely transparent to the application code. In order to keep the SMP 'look-and-feel', one needs to assure a contiguous allocation for the matrix structures in the virtual address space, while the NUMA-aware placement is achieved transparently to the user by mapping the virtual memory pages to the correct physical memory node (see Figure 3.12). This can be easily achieved in Linux, using the low-level Linux memory management system calls. More specifically, we allocate the whole data structure contiguously in the virtual address space with a single call to `mmap()` and then bind every partition on its local node with subsequent calls to `mbind()`, rounding the partition sizes to the nearest multiple of the system's page size[5]. This mechanism relies on the copy-on-write physical page allocation scheme on Linux. The `mmap()` call creates only a virtual memory mapping for the calling thread and does not allocate any physical page, while the call to `mbind()` marks the mapped region to be physically allocated on the specified memory node; this allocation will not take place until a thread writes on this region. This user-defined interleaved allocation scheme allows the transparent conversion of SMP-based code to NUMA-aware, by just replacing the calls to standard memory allocation routines by calls to our interleaved allocator; the rest of the algorithm remains untouched. All the NUMA-aware SpMV kernels presented and discussed in this thesis are implemented using this technique with a minimal programming effort.

While the matrix data and the output vector can be allocated across the physical memory nodes using the interleaved allocator presented previously, the input vector, ideally, must be replicated across the memory nodes, since the access pattern is not known a priori. However, this is not practical in the context of a 'real-life' application, where the input vector changes during the SpMV iterations. For this reason, placing it on a single node is a more reasonable choice. Nonetheless, as depicted in Figure 3.13, this shared copy does not constitute a performance bottleneck. The average performance overhead amounts to 2.5% and can reach up to 7% in more irregular matrices, e.g., parabolic_fem, offshore etc. Based on our previous discussion on the SpMV performance, this behavior is quite expected, since the accesses in the input vector do not generate significant memory traffic for the majority of the sparse matrices and, therefore, the impact of the shared copy on the SpMV performance in NUMA systems is minimal.

[5] Linux allocates and manages memory at the granularity of a *page*, usually 4 KiB in 64-bit systems.

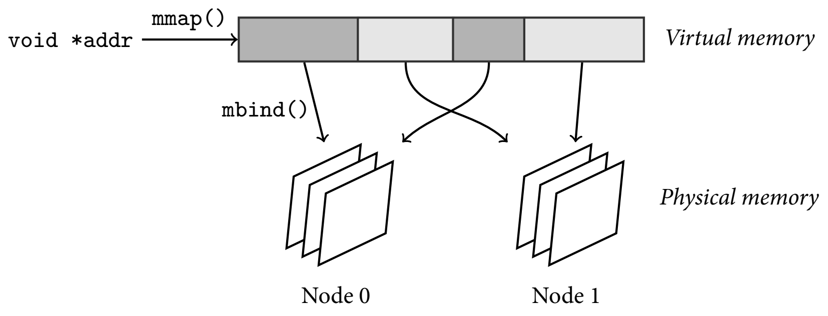


Figure 3.12: Transparent data placement in NUMA architectures on Linux. The space required for a data structure is allocated contiguously in the virtual address space, while individual partitions are then bound to specific physical nodes. User code is agnostic of the placement of the physical pages and 'sees' only the contiguous address space through the `addr` pointer, as expected.

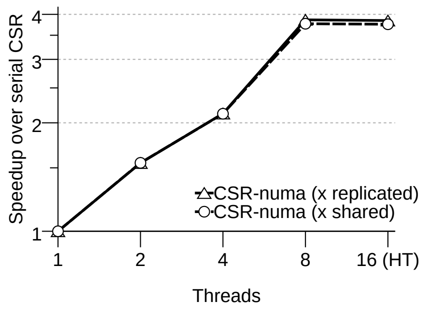


Figure 3.13: The effect of sharing the input vector in NUMA architectures is minimal, since SpMV is chiefly memory bandwidth bound and the accesses in the input vector do not constitute a performance problem for the majority of sparse matrices.

3.4 Related work

For several years, the irregular access pattern on the input vector has been characterized as the major performance problem of the SpMV kernel. Temam and Jalby [1992] perform a thorough analysis of the cache behavior due to the irregular access pattern in the input vector and discuss optimization techniques, such as matrix reordering and blocking. Under the same assumption, Toledo [1997] proposes a number of optimization techniques for improving the memory performance of the kernel, including reordering, blocking and software prefetching for the non-zero values and the column indices. Geus and Röllin [2001] focus on symmetric matrices and propose software pipelining techniques for optimizing instruction parallelism, matrix reordering for increasing cache reuse and blocking for minimizing indirect references. Im and Yelick [2001] focus specifically on fixed size blocking methods as a means for increasing register reuse and propose techniques for automatically selecting the optimal block size. In the same direction, Vuduc et al. [2002] study the interaction of blocked kernels with the cache hierarchy and investigates the performance bounds of the SpMV kernel, while Pinar and Heath [1999] investigate variable-size blocking as an alternative for reducing the indirect references in the kernel. The computational overhead of short rows has been highlighted by White and Sadayappan [1997], characterizing it as more crucial performance issue than the irregular access pattern, since it limits considerably instruction-level parallelism. Mellor-Crummey and Garvin [2004] insist further on this problem and propose compiler optimization techniques for attacking it.

It is not until recently that the memory bandwidth bottleneck has started to be considered explicitly as the most crucial problem of the SpMV execution. Indeed, previous work took a rather implicit approach. A typical example are blocking methods, whose increased performance was attributed to the reduction of the indirect references. In fact, this is a by-product of the real benefit, which is the considerable reduction of the size of the column indexing structure. Williams et al. [2007] and Goumas et al. [2008] are the first to highlight the important role of memory bandwidth in the execution of the SpMV kernel in modern multicore architectures, while Willcock and Lumsdaine [2006] take a first approach in directly tackling this problem through explicit compression of the matrix representation.

3.5 Summary

In this chapter, we have presented an in-depth performance analysis of the SpMV kernel in modern multicore architectures, identifying the key performance problems. The behavior of the SpMV kernel in modern SMP and NUMA architectures is determined by the streaming algorithmic nature of the kernel, which exhibits an extremely low flop:byte ratio. The lack of temporal and spatial reuse requires the memory subsystem to be able to supply the requested data at a CPU-comparable speed. The memory datapath, in terms of available memory bandwidth, becomes therefore the performance hotspot of the kernel, especially in multithreaded configurations, where multiple threads share this path. This problem is more pronounced in SMP architectures, where all memory and interprocessor traffic passes through a low-bandwidth common bus. NUMA architectures, on the other hand, offer significantly higher bandwidth through their integrated memory controller, which can easily accommodate the traffic produced by the cores of the socket. The downside of this type of architectures, however, is that the main memory is segmented into multiple physical nodes and the performance of the kernel becomes very sensitive to the placement of the involved data, since remote memory accesses tend to saturate the processor interconnection links. Overcoming this problem requires a correct placement of the algorithm's working data in the memory nodes, a task that is achieved effortlessly and transparently to the application code using the proposed NUMA-aware Linux-based interleaved allocator.

The performance of the SpMV kernel depends heavily on the sparsity structure of the input matrix. We can identify two large categories of sparse matrices: (a) matrices with a rather regular non-zero element structure and (b) very sparse matrices with an irregular non-zero structure. The first category comprises mostly matrices arising from PDE problems with an underlying 2D/3D geometry and is the typical case of sparse matrices encountered in practice. The key performance problem for these matrices is the contention for memory bandwidth and their performance is almost directly related to the matrix representation size. The second category consists of matrices that do not saturate the memory bandwidth of the system, since they suffer from a diverse set of performance problems, including irregular memory accesses in the input vector, large loop overheads due to very short rows and load imbalances.

We conclude this chapter with a note on the optimization directions for the SpMV kernel, as these were revealed from the performance analysis conducted in this chapter. The goal for a successful optimization of the SpMV kernel must be the minimization of the memory footprint of the matrix, since contention for memory bandwidth is the key performance problem in modern multicore architectures. Performance problems related to very irregular matrices are of secondary importance and can be treated orthogonally to the minimization of the matrix size. For instance, matrix reordering techniques not only treat successfully the load imbalance problem of matrices with an irregular non-zero element distribution, but also manage to optimize the access pattern in the input vector.

4

Optimization opportunities of blocking

Blocking storage formats for sparse matrices exploit the regular non-zero element structure of certain sparse matrices, in order to form one- or two-dimensional dense blocks by grouping together neighboring non-zero elements. By keeping a single column index per block, blocking formats manage to reduce significantly the matrix size and alleviate the pressure to the memory hierarchy of the system. Additionally, the known and regular structure of the blocks, especially in fixed size blocking storage formats, allows the optimization of the SpMV computations, which now become more exposed.

In this chapter, we examine the optimization opportunities of a variety of blocking methods in terms of their compression capabilities and computational characteristics. Based on this knowledge, we propose a simple performance model for selecting the optimal block for BCSR, taking into consideration both the memory and the computational part of the SpMV kernel. The proposed model is able not only to detect the best block, but also the most efficient implementation.

4.1 The effect of compression

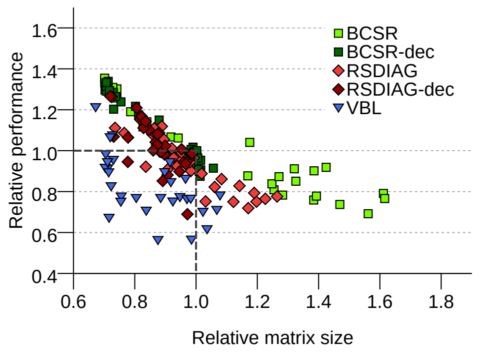
In Chapter 2, we have presented in detail the most representative types of blocking for sparse matrices, namely fixed-size, variable-size and decomposed blocking storage formats. The common characteristic of these formats is the compression of the `colind` CSR's data structure (Chapter 2, Figure 2.2) by keeping a single column index per formed block. Their key difference lies on the way the blocks are formed and, as we shall see in this chapter, characterizes their compression and computational capabilities. Fixed-size blocking methods form fixed-size one- or two-dimensional blocks and either employ zero-padding for forming full blocks or decompose the original matrix into a full-block submatrix and a CSR remainder. Variable-size blocking methods, on the other hand, store arbitrary one- or two-dimensional blocks and employ additional data structures to keep track of their blocks.

Figure 4.1 shows the correlation between the compression ratio and the performance improvement for a diverse set of blocking storage formats, including fixed-size blocking with zero-padding (BCSR, RSDIAG), fixed-size blocking with decomposition (BCSR-dec, RSDIAG-dec) and variable-size blocking (VBL). We show the results for the Harpertown and Gainestown systems in single- and multithreaded configurations for the 30 matrices of our matrix suite. The marked lower left region of the diagrams denotes matrices that a storage format could not offer a performance improvement over CSR, despite compressing its memory footprint. The first clear observation is that there is a tendency, becoming very pronounced in the multithreaded configurations, toward higher performance as the matrix size decreases. This is quite expected according to the quantitative analysis of the SpMV kernel in the previous chapter. It is remarkable that no storage format with a larger than CSR matrix representation did achieve a better performance, even in the single-threaded configurations, where the memory contention is not so intense. The inverse, however, is true for the majority of formats, revealing significant performance overheads unrelated to the matrix size. The most characteristic example is VBL, which, in single-threaded configurations, incurs more than 20% performance degradation, despite its rather compressed representation. This is due to the increased computational overhead of accessing the additional data structure for fetching the block sizes[1]. This situation is reversed as the number of threads increases and the memory contention becomes apparent: the significant compression achieved by VBL relaxes the memory bandwidth requirements and offers a higher performance potential.

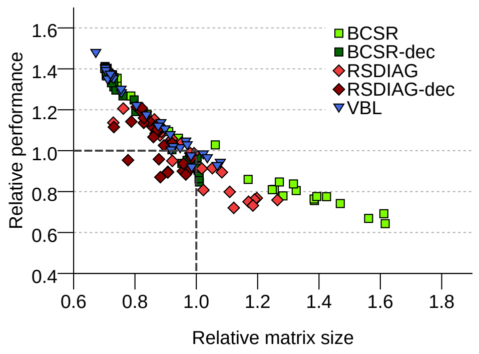
The case of diagonal storage formats is interesting. First, as expected, their compression capability is lower than BCSR's, since they detect only one-dimensional blocks. However, even in cases where they achieve comparable compression, their performance is lower, especially in the Gainestown system. This behavior can be explained by examining closer the access pattern of diagonal blocks: for each block of size b, the kernel must fetch $16b + 4$ bytes ($8b$ for the values, $8b$ for the input vector elements and four bytes for the block column index) and write back into main memory $8b$ bytes per block row[2]. For a 4×1 block, however, the BCSR kernel reads only $8b + 12$ bytes per block (a single input vector element is needed) and writes back again $8b$ bytes per block row.

[1] VBL must fetch the block size for every block encountered; therefore, the smaller the block, the higher the overhead. Additionally, compared to BCSR, it is not easy for a compiler to emit highly optimized code for the block traversal in VBL, since the loop bounds (block size) are unknown.

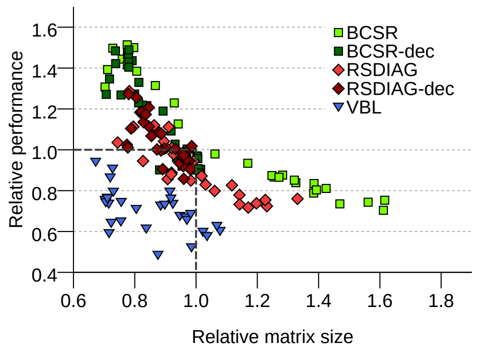
[2] The typical implementation of a blocked SpMV kernel unrolls the block computations and delays the output vector writes until the end of the matrix's block row, by accumulating the intermediate results into registers. This is done in order to minimize the demands on memory-write bandwidth, a resource that is more restricted.



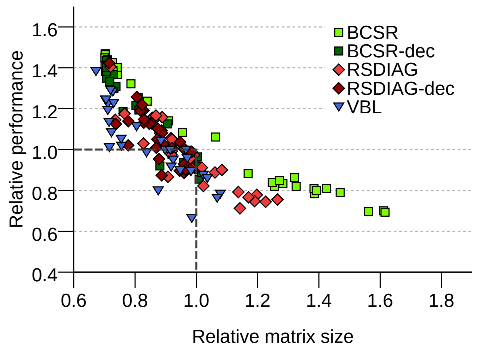
(a) Harpertown, single-threaded.



(b) Harpertown, single socket (four threads).



(c) Gainestown, single-threaded.



(d) Gainestown, single socket (four threads).

Figure 4.1: Correlation of the compression ratio and the performance improvement over CSR achieved by the different blocking storage formats in a set of 30 sparse matrices. The performance results for fixed-size blocking methods correspond to the best block.

The arithmetic intensity for diagonal blocks is therefore significantly lower, explaining their performance deficiency compared to BCSR blocks.

The decomposed formats solve the zero-padding problem of the fixed-size blocking formats, having their size barely exceeding CSR in the worst case, whereas padding in BCSR can lead up to a 60% increase in the matrix size, even for the best performing block. Nonetheless, decomposed formats have to pay the cost of the multiple SpMV operations, leading to a slightly lower performance than their zero-padded counterparts in cases of comparable compression. In cases where almost no compression is achieved, this overhead becomes more visible and can lead to a more than 10% performance degradation compared to CSR.

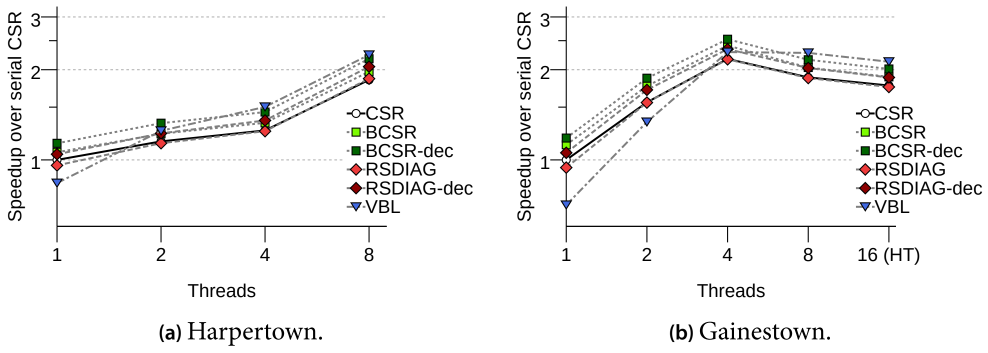
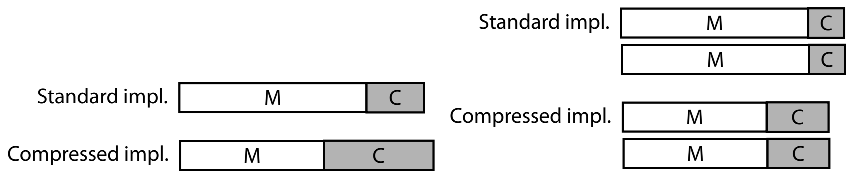


Figure 4.2: The speedup achieved by a variety of blocking storage formats. The performance slowdown in Gainestown from the eight-threaded configuration is due to the non-NUMA-aware implementation used.

The superiority of BCSR in the computations compared to the other blocking formats is depicted clearly in the single-threaded configuration in Gainestown, where it achieves the best performance at the same matrix sizes. This advantage is still preserved in the single-socket configuration in Gainestown, but it is completely overshadowed by the memory bottleneck in the single-socket configuration in Harpertown. Nonetheless, despite being very performant in matrices where it achieves high compression, BCSR's compression capabilities are limited; in almost half of the matrices in our suite, it increased the matrix size, deteriorating SpMV's performance.

The effect of compression as the memory bandwidth contention becomes visible is depicted in the speedup diagrams of Figure 4.2. VBL pays the cost of its additional 'decompression' operations in the few-threaded configurations, but it manages to successfully compensate it as the number of threads increases. From the four-threaded configuration onward in Harpertown, VBL manages to achieve the best SpMV performance on average, while in Gainestown (non-NUMA implementations), takes the lead from the eight-threaded configuration, where the second socket is used and the SpMV computation is now bound from the processor interconnection link. This behavior, depicted visually in Figure 4.3, is typical of highly compressed formats [Kourtis et al., 2008a], where the limited memory bandwidth in multithreaded configurations can easily hide the decompression cost. Finally, the decomposed formats manage to achieve a 5–8% better performance on average compared to their zero-padded counterparts, due to their more compact representation.



(a) Single-threaded. (b) Multithreaded.

Figure 4.3: The effect of compression. Highly compressed storage formats suffer in the single-threaded configuration from the increased decompression overhead. In multithreaded configurations, however, this cost is hidden by the contention in the memory subsystem and the performance benefit becomes evident.

4.2 The effect of the computations

The previous discussion on the compression capabilities of the different blocking storage formats has revealed the significance of the computational part of the kernel in configurations where the available main memory bandwidth is not yet saturated. In this section, we focus more on the computational part of BCSR, which is the most computationally friendly blocking storage format, in order to obtain further insight into the intricacies of the SpMV kernel.

4.2.1 Vectorization and the block shape

The fixed size blocks of BCSR provide a significant computational advantage compared to variable size blocking methods by allowing a number of performance optimizations. Not only modern compilers can provide highly optimized code for loops with fixed bounds, but 'manual' optimizations are also possible. One such optimization is *vectorization*, a technique that increases the throughput of floating point operations by executing the same instruction on multiple floating point data (*Single Instruction Multiple Data – SIMD*). Originated from the vector processors of the 1970's [Russell, 1978], SIMD instructions have been part of modern microprocessors for several years [Raman et al., 2000] and have come into the foreplay recently with the advent of general purpose GPU architectures [Lindholm et al., 2008]. The experimental platforms we use for the performance evaluations in this thesis support a set of SIMD instructions (*Streaming SIMD Extensions – SSE*) that operate on a separate register file comprising 16 'wide', 128-bit floating point registers. A dedicated arithmetic unit is responsible for implementing the SIMD extensions, accompanied by a special *shuffle unit*; this unit is responsible for moving individual SIMD vector elements either across multiple SIMD registers or within the same register. Instructions that engage the shuffle unit for their execution may encounter high latency and low throughput, due to the rather complex circuit logic of this unit.

Algorithms 4.1 and 4.2 show the SIMD implementations for the 1×2 and 2×1 BCSR blocks, respectively, indicating the instructions that involve the shuffle unit. The computation of a 1×2 block starts by loading the two input vector values and the corresponding two non-zero elements into two SIMD registers. It then proceeds with the actual SpMV computation on the SIMD registers. At this point, the typical implementation would add the two elements of the y_0 register using an *horizontal add* instruction. However, this instruction engages the shuffle unit and can be quite expensive, especially in older architectures; therefore, we delay its execution until the end of the block row, when the actual result must be written back to the output vector. The computation of 2×1 blocks starts by loading the two non-zero elements in the y_0 SIMD register, while the single input vector element is stored at the lower 64 bits of the y_1 register. This value is then duplicated (using the shuffle unit) at the upper 64 bits, in order for the SpMV to proceed in a vectorized fashion. The implementations shown for these two types of blocks can be expanded to larger one-dimensional and two-dimensional blocks in a straightforward manner.

SIMD vector loads There are two kinds of instructions for loading full SIMD vectors: *aligned* and *unaligned loads*. An aligned SIMD load is as fast as a normal load, but requires the loading address to be aligned at 16-byte boundaries. This restriction is lifted for the unaligned SIMD loads, but the shuffle unit is employed for properly aligning the vector elements inside the SIMD register. As a result, unaligned loads experience a performance overhead. Due to their strict alignment requirements, aligned SIMD loads cannot be used in BCSR blocks with odd dimensions, e.g., 3×1, 5×1 blocks etc., since the individual blocks cannot be properly aligned. Conversely, the correct alignment requirement can be achieved easily for blocks with even dimensions.

The vectorization of the BCSR blocks has introduced a new set of factors that influence the SpMV performance. Figures 4.5 and 4.4 show the effect of vectorization in the Harpertown and Gainestown, respectively, for the single- and multithreaded configurations. We show results only for the 15 symmetric matrices of our matrix suite, so that symmetric blocks (e.g., 1×2 and 2×1) lead to the same matrix sizes. The impact of vectorization in Harpertown is minimal reaching a maximum of 5% improvement for the 1×8 blocks. The low memory bandwidth of SMP architectures does not leave enough space for computational optimizations, such as vectorization. Additionally, the SIMD implementations for any block make use of the shuffle unit and cannot com-

```
 1: procedure MatVecBcsr1x2($A$::in, $x$::in, $y$::out)
    $A$: matrix in BCSR format
    $x$: input vector
    $y$: output vector
 2:     for $i \leftarrow 0$ to $N$ do
 3:         $\mathbf{y}_0 \leftarrow 0$
 4:         for $j \leftarrow browptr[i]$ to $browptr[i+1]$ step by 2 do
 5:             $j_b \leftarrow \frac{j}{2}$
 6:             $x_c \leftarrow bcolind[j_b]$
 7:             $\mathbf{x}_0 \leftarrow x[x_c]$
 8:             $\mathbf{v}_0 \leftarrow bvalues[j]$
 9:             $\mathbf{y}_0 \leftarrow \mathbf{y}_0 + \mathbf{v}_0 \cdot \mathbf{x}_0$
10:         end for
11:         $\mathbf{y}_0 \leftarrow \mathbf{y}_0 \oplus \mathbf{y}_0$                ▷ Shuffle unit
12:         $y[i] \leftarrow \text{lower}_{64}(\mathbf{y}_0)$                ▷ Shuffle unit
13:     end for
```

Algorithm 4.1: SIMD implementation of the BCSR kernel for 1×2 blocks. Vector registers are in bold typeface; the $\oplus$ operator performs the addition of the individual register elements (horizontal add), while the lower_{64} gets the 64 lower bits of a vector register.

pensate the overhead, since the blocks are quite small and the unit is not very optimized. This overhead becomes more obvious in cases where unaligned loads are used, where vectorization even degrades performance[3]. In Gainestown, on the other hand, the gain of vectorization is significant, reaching 58% on average for the 1×8 block. This behavior is due chiefly to the ample memory bandwidth available and the optimized shuffle unit of the Nehalem microarchitecture [Int, 2009; Fog, 2012].

A closer look at the Gainestown results (Figure 4.5) reveals some important characteristics of the BCSR blocks. First, the vertically oriented blocks (standard version) exhibit higher performance than their horizontally oriented counterparts, although both lead to the same matrix size. Horizontally oriented blocks resemble the case of diagonal blocks (Section 4.1), since for each block the kernel must fetch a series of input vector elements, leading to a low local arithmetic intensity. The performance differences are attenuated in the vectorized versions due to the increased overhead of the shuffle operations

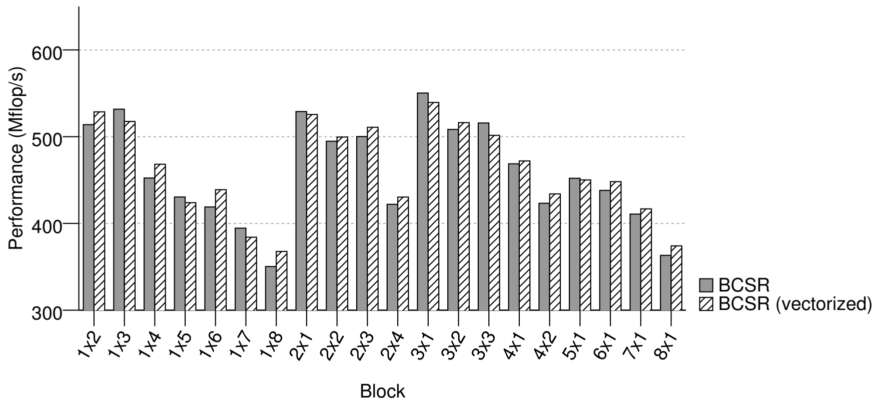
[3] The performance of 3×1, 5×1 blocks etc. is not degraded, since their unaligned store is not the in the critical path.

```
1: procedure MatVecBcsr2x1($A$::in, $x$::in, $y$::out)
   $A$: matrix in BCSR format
   $x$: input vector
   $y$: output vector
2:     $i_r \leftarrow 0$
3:     for $i \leftarrow 0$ to $N$ step by 2 do
4:         $\mathbf{y}_0 \leftarrow 0$
5:         for $j \leftarrow browptr[i_r]$ to $browptr[i_r + 1]$ step by 2 do
6:             $j_b \leftarrow \frac{j}{2}$
7:             $x_c \leftarrow bcolind[j_b]$
8:             $\text{lower}_{64}(\mathbf{x}_0) \leftarrow x[x_c]$                ▷ Shuffle unit
9:             $\text{upper}_{64}(\mathbf{x}_0) \leftarrow \text{lower}_{64}(\mathbf{x}_0)$     ▷ Shuffle unit
10:            $\mathbf{v}_0 \leftarrow bvalues[j]$
11:            $\mathbf{y}_0 \leftarrow \mathbf{y}_0 + \mathbf{v}_0 \cdot \mathbf{x}_0$
12:        end for
13:        $y[i] \leftarrow \mathbf{y}_0$
14:        $i_r \leftarrow i_r + 1$
15:    end for
```

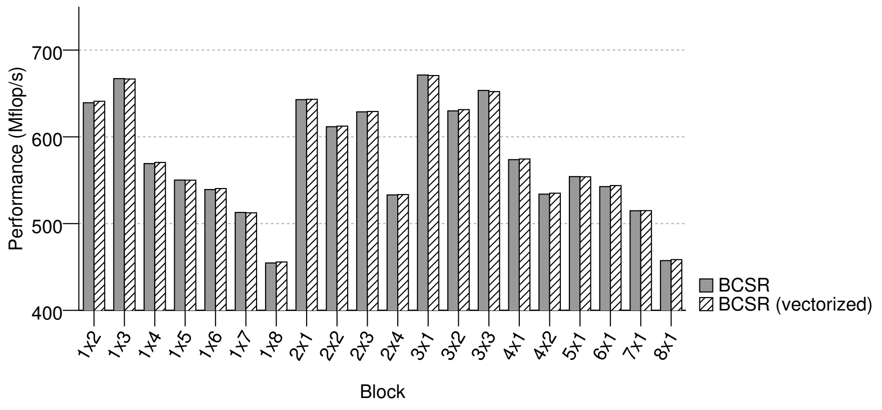
Algorithm 4.2: SIMD implementation of the BCSR kernel for 2×1 blocks. Vector registers are in bold typeface; the lower_{64} and upper_{64} operators get the 64 lower or upper bits of a vector register, respectively.

in the vertically oriented blocks. The second key observation is that the performance of the SIMD versions is more loosely coupled with the matrix size, due to the various overheads introduced by the strict SIMD implementation requirements. For example, one-dimensional blocks with an odd dimension perform worse that ones with an even dimension, despite leading to a smaller matrix representation and, similarly, 3×2 blocks are 17% faster than 2×3 blocks. Selecting the optimal BCSR block, therefore, is not a straightforward task and the computational part of the kernel must also be considered.

Closing the discussion on the vectorization of the BCSR blocks, we should highlight the multithreaded case in both Harpertown and Gainestown systems. The contention for memory bandwidth resources leaves almost no headroom for computational optimizations, such as vectorization. In Harpertown, which suffers more from the memory bottleneck, the gain of vectorization is negligible, while in Gainestown, it falls below 5%. In these cases, the criterion of matrix size seems rather safe for predicting the SpMV performance.



(a) Single-threaded performance.

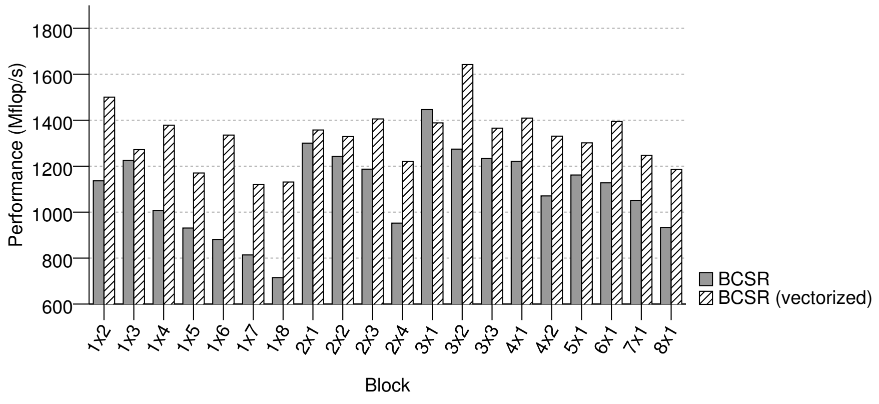


(b) Multithreaded performance (single socket, four threads).

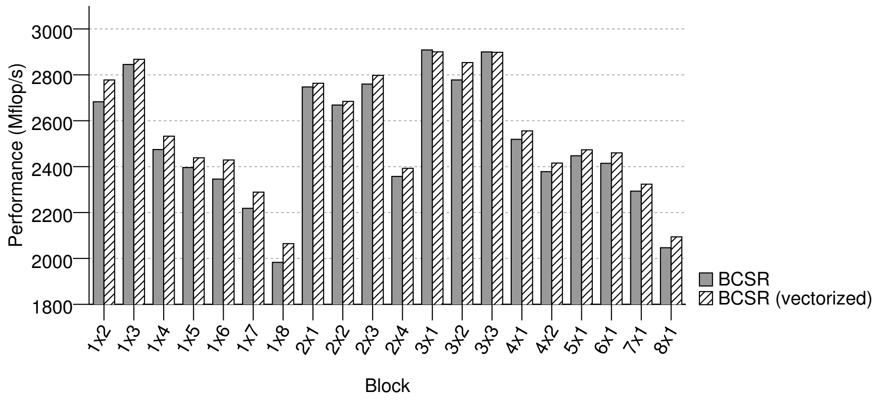
Figure 4.4: The effect of vectorization of BCSR in the Harpertown SMP platform.

4.3 Predicting the optimal block size

The discussion on the performance characteristics of BCSR blocks has raised the need for formulating a performance model for the SpMV kernel that will guide us through the selection of the correct optimization. This performance model must consider also the computational part of the kernel, since it can considerably influence the overall performance. In this section, we propose and evaluate the accuracy of two simple performance models for the BCSR storage format, namely the MemComp and Overlap models. These models consider also the computational part of the kernel and 'decide' on the optimal block and implementation (standard or vectorized). We focus specifically on the single-threaded performance, since the computational part of the kernel is



(a) Single-threaded performance.



(b) Multithreaded performance (single socket, four threads).

Figure 4.5: The effect of vectorization of BCSR in the Gainestown NUMA platform.

more significant in this case, rendering the optimal selection more difficult.

4.3.1 The Mem model

The Mem model is the most straightforward performance model for the SpMV kernel. Proposed by Gropp et al. [1999], this model is based solely on the algorithm's working set (matrix representation and input/output vectors sizes) and, therefore, it is not strictly focused on any storage format. Based on the streaming nature of the SpMV kernel, this model assumes that the sole performance-critical parameter is the effective memory bandwidth of the underlying system. Therefore, given a system with bandwidth B and a matrix with a working set S, the SpMV execution time according to the Mem model will be

$$t_{\textsc{Mem}} = \frac{S}{B} \tag{4.1}$$

The effective bandwidth of a system can be obtained through microbenchmarks, such as STREAM [McCalpin, 1995], BenchIT [Juckenland et al., 2004] etc.

4.3.2 The Sparsity model

Im et al. [2004] and Buttari et al. [2007] focus specifically on BCSR and propose a performance model for selecting the best block. This model, employed also by the OSKI sparse matrix optimization framework [Vuduc et al., 2005], considers both the memory and the computational part of the kernel, but in a more implicit manner. According to the Sparsity model, an estimate of the execution time for a sparse matrix A can be computed as follows:

$$t_A \propto \frac{\text{fill}_A(r, c)}{\text{perf}_A(r, c)} \tag{4.2}$$

The $\text{fill}_A(r, c)$ parameter is the fill-in ratio of the $r \times c$ block and denotes the zero padding needed to construct the full BCSR blocks. The fill-in ratio is an indication of the computational overhead of padding (redundant floating-point operations) and the resulting matrix size. The $\text{perf}_A(r, c)$ is the actual SpMV performance for the $r \times c$ block. This parameter is microarchitecture-dependent and captures the performance variation for the different blocks leading to the same fill-in ratio, i.e., the same matrix size. Since computing $\text{perf}_A(r, c)$ for an arbitrary matrix A is irrational, these values are obtained by profiling the SpMV execution of a large dense matrix, exceeding the last level cache size of the target system. Therefore, the execution time estimate of the Sparsity model for an $r \times c$ block is calculated as follows:

$$t^{r,c}_{\textsc{Sparsity}} = \text{fill}_A(r, c) t^{r,c}_{dense} \tag{4.3}$$

The fill-in ratio is the ratio of the BCSR non-zeros, including padding elements, over the non-zero elements of the original matrix. Computing the fill-in ratio for every relevant block can still be time-consuming[4]. For this reason, in practice, an estimate of the fill-in ratios is computed by sampling the sparse matrix.

The Sparsity model can be applied also to CSR, where $\text{fill}_A(r, c) = 1$ and $t^{r,c}_{dense}$ is the SpMV execution time for the profiled dense matrix stored in CSR format.

[4] The cost of converting a sparse matrix into the BCSR format is an order of magnitude higher than a single SpMV operation.

4.3.3 The MemComp model

The MemComp model is a generalization of the Mem model that considers also the computational part of the kernel. It regards SpMV as a two-phase operation consisting of the memory and the computational parts. The memory part is the one computed by the Mem model (equation (4.1)), while the computational part is computed explicitly using an estimate of the execution time of a single BCSR block. Assuming a sparse matrix of size S in BCSR format, using N_b $r \times c$ blocks, the SpMV execution time, according to the MemComp model, is computed as follows:

$$t^{r,c}_{\text{MemComp}} = \frac{S}{B} + N^{r,c}_b \cdot t^{r,c}_b \tag{4.4}$$

The first term of the equation corresponds to the memory part of the kernel and B is the effective memory bandwidth of the underlying system. The second term is the computational part and corresponds to the execution time of all N_b blocks. The execution time t_b of a single block is obtained by profiling the execution of a very small dense matrix, fitting in the L1 cache of the system, for all $r \times c$ blocks under consideration.

The MemComp model can be applied also to decomposed formats, where the formula (4.4) is computed for each submatrix. The case of CSR is treated as a degenerate BCSR with 1×1 blocks; therefore, $N^{r,c}_b = NNZ$ and $t^{r,c}_b$ is the execution time of the profiled small dense matrix stored in CSR.

4.3.4 The Overlap model

The MemComp model treats the memory and the computational parts of the SpMV kernel separately, as if they were two completely distinct phases. However, according to the quantitative analysis presented in Chapter 3, the computations of the SpMV kernel are not very exposed, due to its streaming nature, which tends to saturate the available memory bandwidth. The Overlap model assumes an overlapping of the memory and computational parts of the kernel and only a fraction of the computational part is eventually visible in the overall SpMV execution time. This behavior is possible in modern microarchitectures, even in a single-threaded configuration, thanks to *hardware prefetching*. Hardware prefetching is a mechanism for fetching into the processor's cache the data that is likely to be accessed by a program shortly, based on its current memory reference pattern. This mechanism not only hides the main memory access latency, but also allows streaming applications, such as SpMV, to fully utilize the available memory bandwidth of the system [Goumas et al., 2009].

The Overlap model adjusts the computational part term of the MemComp model by multiplying it by a factor, called *Non-Overlapping Factor* (NOF), indicating the percentage of SpMV computations exposed to the overall execution time. The NOF factor is defined for every $r \times c$ block as

$$NOF^{r,c} = \frac{t^{r,c}_{\text{real}} - t^{r,c}_{\textsc{Mem}}}{N^{r,c}_b \cdot t^{r,c}_b} \tag{4.5}$$

This factor is obtained by profiling the execution of a large dense sparse matrix, exceeding the last level cache of the system under consideration. It can be viewed as a correction factor of the computational part of the MemComp model, since it actually compares the real computations time (nominator) versus the theoretical time of the MemComp model (denominator). Given the NOF factor, the SpMV execution time according to the Overlap model is computed by the following formula:

$$t^{r,c}_{\textsc{Overlap}} = \frac{S}{B} + NOF^{r,c} \cdot N^{r,c}_b \cdot t^{r,c}_b \tag{4.6}$$

Similarly to MemComp, the Overlap model can be applied also to both decomposed formats and CSR, by considering it as a degenerate BCSR with 1×1 blocks.

4.3.5 Assessing the accuracy of the models

The main purpose of the performance models presented in this chapter is the selection of the optimal block for BCSR. However, with the exception of the Sparsity model, the rest three models, namely Mem and the proposed MemComp and Overlap models, take a more explicit approach in determining also the kernel's execution time. We could distinguish, therefore, two metrics for assessing the prediction quality of the considered models:

Selection accuracy: This metric determines how accurate is the performance model in determining the optimal BCSR block for a specific matrix. Quantitatively, the quality of the prediction can be assessed by how close to the optimal block's performance is the performance of the block selected by the model under consideration.

Execution time prediction accuracy: This metric determines how accurate the performance model at hand can predict the execution time of the SpMV kernel in absolute terms.

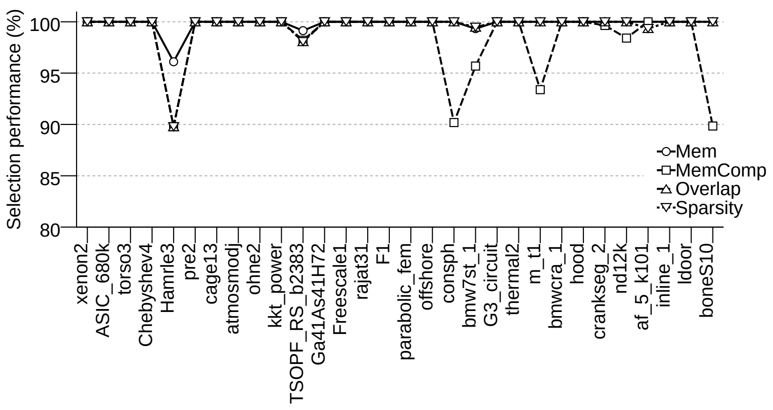
There exist an one-way relation between the two accuracy metrics described. If a performance model is fairly accurate in predicting the absolute execution time of the SpMV kernel using different blocks, it will also exhibit a high selection accuracy. The inverse, however, is not true; providing accurate block selections does not prescribe an accurate execution time prediction, since it requires only a good ranking of the candidate blocks.

	Harpertown			Gainestown		
	Correct	*Dist.*	*C.I.*	*Correct*	*Dist.*	*C.I.*
Mem	**27/30**	1.81%	±2.05%	10/30	8.50%	±1.86%
Sparsity	26/30	3.33%	±4.54%	26/30	5.89%	±4.90%
MemComp	22/30	5.61%	±2.88%	**28/30**	3.63%	±0.34%
Overlap	26/30	3.53%	±4.54%	26/30	5.89%	±4.90%

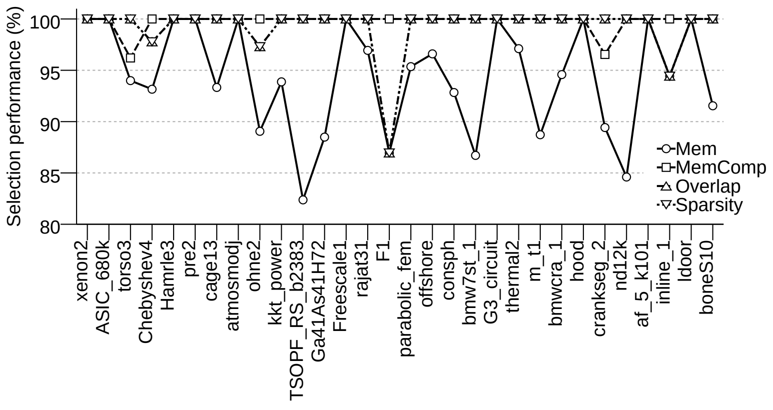
Table 4.1: Selection accuracy of the considered performance models. It is shown the total number of correct predictions, the average distance of mispredicted cases from the optimal performance (Dist.) and a 95% confidence interval (C.I.). In cases with a few mispredictions, the confidence intervals may exceed the average misprediction distance; this is due to the very small sample of mispredictions available.

The key concern in this chapter is providing high selection accuracy, since we are interested in selecting the best SpMV optimization. The performance models must be able select the best option among the baseline CSR and the standard and SIMD BCSR implementations. Apparently, the Mem model cannot differentiate between the simple and vectorized BCSR implementations, since it considers only the memory part of the kernel. For this reason, we empirically select the standard implementation for SMP systems and the SIMD version for NUMA ones. Figure 4.6 shows schematically the selection accuracy of the four performance models considered in the Harpertown and Gainestown systems, while Table 4.1 summarizes the accuracy results. The competence of the Mem model is clear in Harpertown, where it provides the correct selection for 27 out of the 30 matrices (90%) in our suite. This is more or less expected, since the SpMV kernel is bound from the low memory bandwidth of the Harpertown system. The MemComp model, on the other hand, overestimates the impact of the computations in this system, missing the correct prediction in eight cases. The Sparsity and Overlap models are clearly more adapted and manage an 87% accuracy (26 correct predictions). In case of a misprediction, all models, except MemComp, are within 5% of the optimal performance, which reveals a rather high prediction quality. The MemComp model falls victim of its overestimation of the computations and its prediction can lead up to 10% performance degradation.

The situation is completely reversed in the Gainestown system. The computations play a rather significant role in this case and the MemComp model excels, providing the correct predictions in almost all the matrices (28 correct predictions, 93%). The performance of the Mem model is disappointing, managing only 10 correct predictions (33%). This is quite expected, according



(a) Harpertown.



(b) Gainestown.

Figure 4.6: Selection accuracy of the considered performance models. The MEM model achieves a high selection accuracy in Harpertown, where the memory bandwidth contention is more pronounced, but it falls short in its predictions in Gainestown, where the computational part of the kernel plays a significant role in the overall SpMV performance. The inverse is true for the MEMCOMP model, while OVERLAP and SPARSITY balance better the memory and computational parts, achieving good predictions in both cases.

	Harpertown		Gainestown	
	Dist. (R)	*Dist. (A)*	*Dist. (R)*	*Dist. (A)*
Mem	$12.94 \pm 2.71\%$	$18.22 \pm 3.78\%$	$43.05 \pm 3.35\%$	$45.32 \pm 2.85\%$
MemComp	$28.41 \pm 3.93\%$	$23.13 \pm 4.36\%$	$23.98 \pm 3.35\%$	$19.76 \pm 3.75\%$
Overlap	$5.28 \pm 1.99\%$	$12.04 \pm 4.20\%$	$9.06 \pm 5.71\%$	$15.81 \pm 5.12\%$

Table 4.2: Execution time prediction accuracy of the considered SpMV performance models. It is shown the average distance between the predicted and the real performance for regular matrices (Dist. (R), 15 matrices) and for the whole matrix suite (Dist. (A)). The corresponding 95% confidence intervals are also presented.

to our earlier performance analysis, since the SpMV performance in Gainestown is not directly related to the matrix representation size. The Sparsity and Overlap models exhibit a rather stable behavior, providing a correct prediction in 26 matrices (87%). However, the cost of misprediction is higher in Gainestown, leading to an average 5.9% degradation for Sparsity and Overlap and 8.5% for the Mem model.

Although the key purpose of the performance models presented in this chapter is the correct prediction of the optimal BCSR block, the Mem, MemComp and Overlap models are derived from an abstraction of the execution time of the SpMV kernel. For this reason, we provide in Table 4.2 some statistics on the execution time prediction accuracy of the proposed models. Since the proposed models assume a streaming behavior of the SpMV kernel, we provide specific results for matrices with a high flop:byte ratio ($\geq$ 1.6), as these matrices do not suffer from collateral problems, such loop overheads, irregular memory accesses etc. (see Chapter 3 for a detailed analysis) and are typical of SpMV applications. The Overlap model manages the most accurate execution time prediction, leading to an average 5.3% absolute performance difference in Harpertown for regular matrices. In Gainestown, this difference increases, but still remains below 10%. When the irregular matrices are also considered, the prediction accuracy deteriorates, reaching a 12% distance in Harpertown and 15.8% in Gainestown. The Mem and MemComp models are less accurate, though, the first being more suitable for Harpertown and the latter for Gainestown, but still stay considerably lower than Overlap. We should note here, that the accuracy of MemComp model seems to be favored by the more irregular matrices. Due to the irregular memory reference pattern, these matrices tend to underutilize the available memory bandwidth and their computations are more exposed, allowing the MemComp model to perform better predictions.

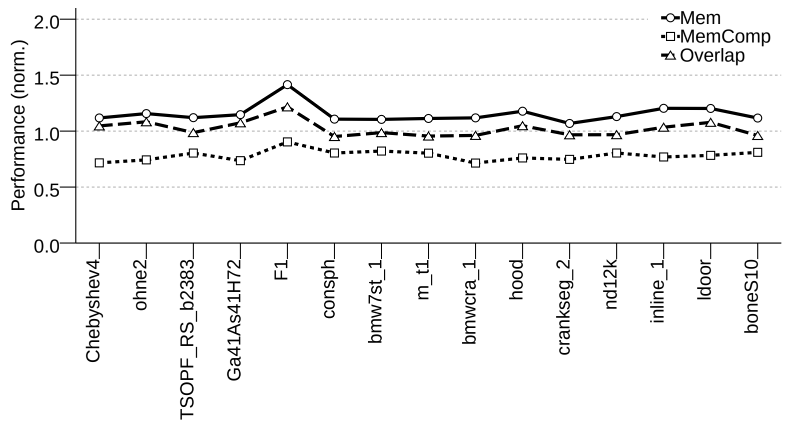
Finally, Figure 4.7 shows the predicted execution time by the three considered models for the more regular matrices. The Mem model tends to overestimate the SpMV performance, since it considers solely the memory part of the kernel. On the other hand, the MemComp model underestimates the performance, as it completely decouples the memory and the computational parts. The Overlap model, however, balances the two parts and manages a better approximation of the SpMV performance. Its performance line for our two architectures exhibits its better adaptability: in Harpertown, it is closer to the Mem's line, revealing that the memory part is more significant in this architecture; the inverse is true for Gainestown, where Overlap is closer to the MemComp's line.

4.3.6 Extensions

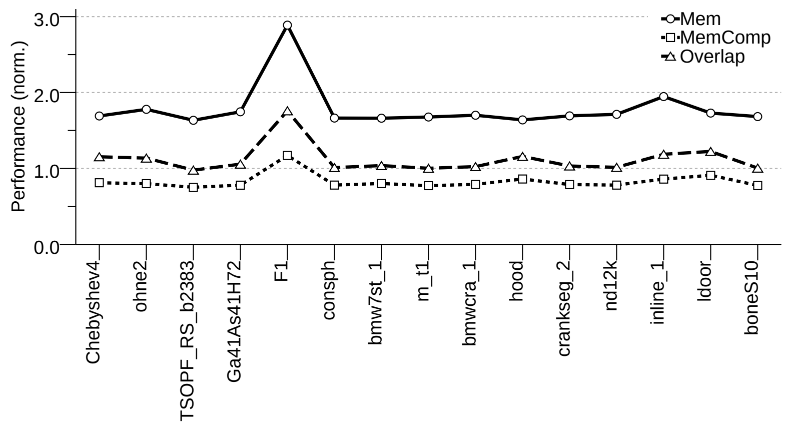
The performance models presented in this chapter are focused on the single-threaded SpMV performance. Our previous performance analysis of the SpMV kernel has revealed that the memory part of the kernel becomes more significant in multithreaded configurations, producing an almost linear relation between the matrix size representation and the SpMV performance. For this reason, we expect the Mem model to accurately predict not only the optimal block size, but also the actual execution time, provided no other performance impeding factors are present (e.g. load imbalances). Determining correctly the non-overlapping factor will allow the Overlap model to match the Mem's accuracy, since it has been proved to be rather adaptable. Similarly, the Sparsity model has also been proved to be adaptable. The MemComp model, on the other hand, is more monolithic and we expect it to be less accurate in a multithreaded context.

Achieving a total performance prediction for the SpMV kernel is not a very straightforward task. All presented models made the assumption of a streaming behavior of the kernel, which is a standard for the majority of matrices involved in sparse matrix computations. However, a more general performance model should take into account parameters such as the flop:byte ratio (or average non-zero elements per row[5]) and load imbalance in multithreaded configurations. The Overlap models showed considerable adaptability and we believe it will be able to 'catch' also these corner-cases, with a suitable tweaking of its control parameters.

[5] Buttari et al. [2007] present a performance model that also considers the average row elements per row, but the authors do not elaborate on multithreaded configurations and in a large variety of sparse matrices.



(a) Harpertown.



(b) Gainestown.

Figure 4.7: Execution time prediction accuracy of the considered performance models for regular matrices (flop:byte ratio $\geq$ 1.6). The Mem model tends to overestimate performance, while the MemComp model underestimates it. The Overlap manages to balance the memory and computational parts of the kernel and achieves high accuracy. The F1 matrix can be considered as an outlier, since it exhibits a rather irregular structure, despite the high flop:byte ratio.

4.4 Summary

In this chapter, we have discussed and evaluated the performance optimization opportunities of the blocking storage formats. We showed that single-threaded configurations do not saturate the memory bandwidth of the system and leave headroom for computational optimizations, especially in NUMA architectures. Indeed, employing the SIMD extensions of modern processors, we were able to accelerate the SpMV performance more than 50% for certain BCSR blocks. Motivated by this behavior, we proposed and evaluated two performance models (namely, MemComp and Overlap) for selection of the optimal BCSR block; these model consider both the memory and computational part of the kernel. In addition, the Overlap model was able to adapt successfully to the different architectures and predict quite accurately the actual SpMV performance in more regular matrices.

In a multithreaded context, the saturation of the memory bandwidth eliminates almost any benefit from a computational optimization. The compression capability of a storage format becomes of vital importance and formats that lead to the most compact matrix representation exhibit a high performance potential. A key conclusion of the analysis in this chapter is that there exists a clear tradeoff between the high compression capability and the computational advantage of a storage format. The most compressed formats, such as VBL, have to pay a high cost of decompression in single-threaded configurations (e.g., accessing additional data structures, performing time-consuming operations etc), but compensate it in a highly multithreaded context. In such a context, their performance advantage is clear, since the execution time is almost solely determined by the available memory bandwidth.

5

The Compressed Sparse eXtended Format

The performance analysis of the SpMV kernel and blocked storage formats in the previous chapters has revealed the contention for main memory bandwidth resources as the major performance bottleneck of the SpMV kernel in modern multicore architectures. As the number of SpMV threads is increasing, the memory bandwidth is quickly saturated, leaving no room for computational optimizations. This behavior is especially apparent in SMP architectures, where the sole performance-critical parameter in multithreaded configurations is the matrix size representation; the lower the representation size of a specific matrix, the higher its SpMV performance. Due to the a priori unknown distribution of the non-zero elements of a sparse matrix, a highly compressed storage format must rely on special data structures and decompression operations, in order to avoid zero padding. These operations may incur a significant computational overhead, which is, however, completely hidden in multithreaded configurations, where the memory bandwidth bottleneck leaves considerable slack for time-consuming operations.

Motivated by this inherent SpMV behavior, we propose in this chapter the *Compressed Sparse eXtended (CSX)* format, a highly compressed storage format for sparse matrices, that is able to detect and encode simultaneously a multitude of matrix substructures. CSX employs explicit compression techniques, such as delta and run-length encoding, in order to compress efficiently the `colind` data structure of the original CSR. In conjunction with the use of matrix coordinate transformations, CSX is able to detect a wide variety of non-zero elements substructures, including horizontal, vertical, diagonal and two-dimensional blocks. This feature not only increases its compression capability, but also allows CSX to successfully adapt to the specificities of every matrix without sacrificing performance. Indeed, CSX is able to accelerate the SpMV performance more than 60% on average in SMP systems and approximately 20% in NUMA architectures. Additionally, the ability to adapt to the non-zero element structure of each matrix provides a considerable performance stability, a merit lacking from the majority of CSR alternatives.

Format	Substructures	Padding	Loop overheads	Decompression overheads
CSR	None	No	No	No
BCSR	fixed 2-D	Yes	No	No
RSDIAG	fixed diagonal	Yes	No	No
Decomposed	fixed 1-D	No	Yes	No
VBL	variable horiz.	No	No	Yes
CSX	variable 1-D, 2-D	No	No	Yes

Table 5.1: Key features of the most important CSR alternatives.

CSX is a relatively complex storage format. In this chapter, we describe in detail every aspect of it and evaluate its performance on our computational testbed. We take particularly care of the preprocessing cost of CSX and evaluate its overall performance impact on a multiphysics simulation software.

5.1 The need for an integrated storage format

Sparse matrices arising from the discretization of partial differential equations have their non-zero elements arranged in substructures either extending to some one-dimensional direction (e.g., horizontal, vertical, diagonal) or expanding to two-dimensional blocks. The exact nature of these substructures depends chiefly on the underlying application domain and any preprocessing performed in the original matrix (e.g., bandwidth minimization techniques) that alters the distribution of the non-zero elements. As already discussed in Chapter 2, exploiting these substructures can result in a reduction of the matrix size. However, restraining a storage format to detect a single type of substructures may have diminishing returns in cases where such substructures do not exist in adequate quantities inside the sparse matrix, leading even to an increase in matrix size compared to the standard CSR. An example is shown in Figure 5.1, where a sample matrix with a variety of substructures is shown; trying to construct simply BCSR blocks results in excessive padding, while applying VBL incurs significant overhead for diagonally arranged elements. Other alternatives, such as decomposed matrices that split the original matrix into multiple submatrices, each one storing a different substructure, experience additional performance overheads, due to the multiple SpMV operations (see Chapter 4, Section 4.1). The need for an integrated storage format that could incorporate a multitude of substructures is seemingly the most viable approach to a performant CSR alternative. Table 5.1 summarizes the basic features of the most important alternatives.

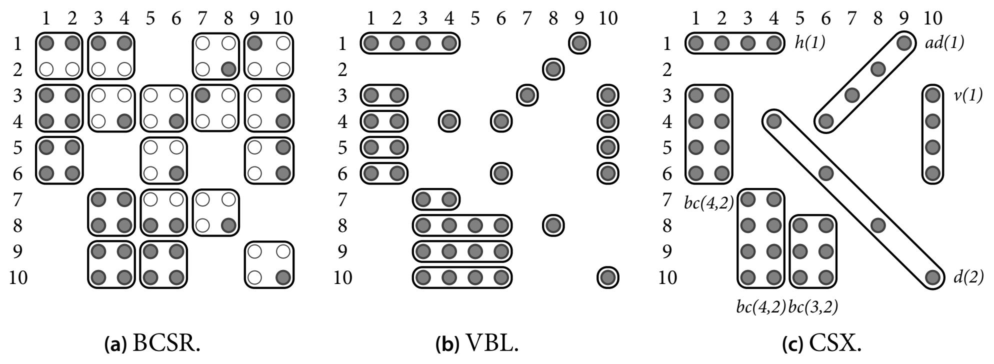


Figure 5.1: The advantage of detecting multiple substructures inside a sparse matrix (gray dots are non-zero elements, white dots are padding elements). The use of padding in fixed size blocking (BCSR) can be excessive for matrices with irregular non-zero element structure, while detecting only one substructure type (VBL) cannot exploit substructures in different directions. The proposed CSX format can detect a great variety of substructures without the need of padding. The annotations beside or below the detected substructures by CSX denote their exact instantiation (see text). *[CSX substructure legend – h(X): horizontal, v(X): vertical, d(X): diagonal, ad(X): anti-diagonal, bc(X,X): block column-aligned; numbers inside parentheses are the delta distances or the block dimensions; numbers in delta units denote their representation size in bits.]*

The proposed Compressed Sparse eXtended (CSX) format is a highly compressed storage format that integrates five different substructure categories and can be easily expanded to support more. When designing CSX we set a number of goals for the new format:

(a) it should specifically target on the minimization of the memory footprint of the matrix, since SpMV is a memory bandwidth bound kernel,

(b) it should cover a wide range of substructures inside the matrix, including horizontal, diagonal and 2-D blocks,

(c) it should expose a stable high performance behavior across different matrices and symmetric shared memory and NUMA architectures,

(d) it should be extensible and adaptive in the sense of supporting new substructures or implementing variations of the existing ones.

In order to meet the two first design goals, we employ an aggressive compression scheme for CSX. We believe that data compression techniques will play a more significant role in future multicore and manycore chips as a means not only for minimizing the communication cost in different levels (processor-to-memory, processor-to-processor etc.), but also for increasing energy-efficiency.

CSX builds on top of the CSR-DU format [Kourtis et al., 2008b] (see Chapter 2, Section 2.3), by adding run-length encoding to the delta encoding of the column indices performed by CSR-DU, in order to detect sequences of non-zero elements either continuous or separated by some constant delta distance. To detect non-horizontal substructures, we employ the notion of coordinate transformations to transform the elements of the matrix according to the desired iteration order (e.g., vertical, diagonal, block-wise etc.) and then reuse the very same detection process that we use for the horizontal substructures. This technique adds also to the design goal of the extendibility, since it suffices to define a '1-1' coordinate transformation in order to detect any substructure inside the matrix. The third goal is assured in SMP systems by the highly compressed representation of CSX, which leads to maximal compression ratios. For NUMA systems, we take particular care in optimizing the computations by generating optimized code at the runtime and relaxing the compression scheme. The runtime code generation employed in CSX not only allows a high performance SpMV implementation, but also facilitates the task of uniformly supporting multiple substructures.

5.2 CSX data structures

CSX replaces both the `rowptr` and `colind` arrays of the standard CSR with a single control byte-array containing all the required information, called `ctl` (Figure 5.2). Similar to CSR-DU, CSX divides the matrix into units. In CSX's terminology, a *unit* represents a sequence of non-zero elements inside the sparse matrix, either a substructure (*substructure unit*) or a sequence of column index delta distances represented by the same number of bytes (*delta unit*). A CSX unit is comprised of two parts: the *head* and the *body*. The head contains a two-byte descriptor of the unit and its initial column index (`ucol` field) encoded as a variable-size integer. The two byte descriptor stores a 6-bit ID of the unit (`id` field) and its size in non-zero elements. The `nr` bit denotes the start of a new row. Finally, the body part is present only in delta units and stores the column index delta distances as fixed-size integers.

The `rjmp` bit and the `ujmp` field serve a special purpose. Since CSX encodes non-horizontal substructures, it is possible for substructures to group together all the elements of subsequent rows. For example, the anti-diagonal substructure in Figure 5.1c starting at position $(1, 9)$ contains also the sole element of the second row, leaving it empty. These rows must be skipped when computing the SpMV. Therefore, the `rjmp` bit denotes the existence of empty rows, while the `ujmp` field stores the number of empty rows to skip in a variable-size integer.

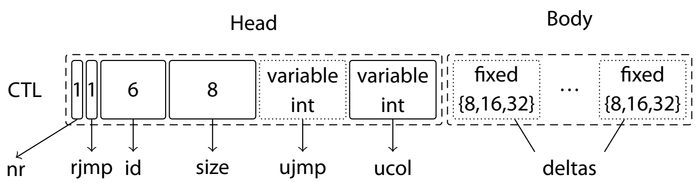


Figure 5.2: The `ctl` byte-array used by CSX to encode the location information of the non-zero elements of a sparse matrix. Optional fields are denoted with a dotted bounding box.

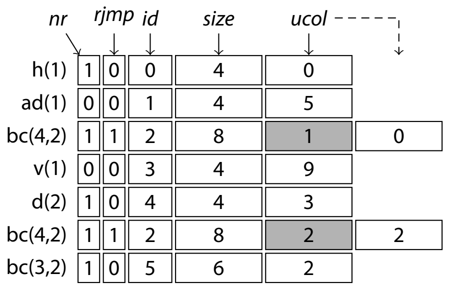


Figure 5.3: The resulting `ctl` sequence for the sample matrix of Figure 5.1c. The extra `ujmp` field for row jumps, present only when the `rjmp` bit is set, is shaded. The column delta distances are calculated between the upper leftmost elements of substructures, with the exception of horizontal substructures, from which the delta distances are calculated from their rightmost element (see `ucol` of the ad(1) substructure encoding). The total storage required for storing the column and row index information is only 23 bytes, while the `colind` and `rowptr` arrays of CSR would require 216 bytes.

Similar to CSR, CSX stores the non-zero elements of the matrix in a `values` array, but in a substructure row-wise order. For example, the substructures in Figure 5.1c will be stored in the following order: h(1), ad(1), bc(4,2), v(1), d(2), bc(4,2) and bc(3,2). Figure 5.3 shows the exact encoding of the `ctl` data structure for the sample matrix of Figure 5.1c.

Encoding variable-size integers in CSX

In an attempt to be as compact as possible, the CSX format employs the use of variable-size integers to encode the initial column indices of its units and the row jumps. In this encoding, an arbitrary integer is stored in 7-bit chunks reserving the most significant bit (bit 7) of each byte as a link between the different chunks (Figure 5.4). The gain in the total matrix size by the use of variable-size integers is 2–3% and has an almost direct impact on the performance of the SpMV kernel in symmetric shared memory systems, where the memory band-

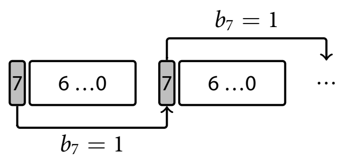


Figure 5.4: The variable-size integer encoding employed by CSX. Only the seven lower bits are used for storing an integer; numbers larger 127 are stored in 7-bit pieces with bit 8 serving as a continuation marker.

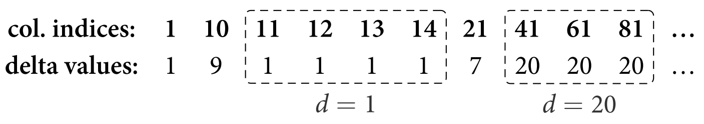
width saturation leaves enough space for the additional computational burden of the decoding process.

5.3 Detection and encoding of substructures

5.3.1 Mining the matrix for substructures

CSX can detect a variety of non-horizontal substructures with the use of coordinate transformations on the non-zero elements of the matrix. To facilitate the detection process, CSX uses a special internal representation for the sparse matrix, which is a hybrid of CSR and a generic version of the COO format. Instead of simple non-zero elements, the CSX's internal representation stores *generic elements*; a generic element is either a single non-zero element or a substructure. Similarly to the COO format, each generic element is stored as an (i, j, v, t) tuple, lexicographically sorted on the (i, j). In the case of a substructure, (i, j) represents the coordinate of the first element in the substructure, v is an array of its elements and t is its type. We also keep the CSR's row pointers for fast row accessing. This internal representation is constructed once during the 'loading' of the original matrix, either from CSR or from the disk.

CSX detects substructures by applying *run-length encoding* on the column indices of the matrix. The run-length encoding first computes the delta distances of the column indices and then assembles groups, called *runs*, from the same distance values (Figure 5.5, Algorithm 5.1). Each run is identified by the common delta value and its length. Runs with length greater or equal to two form a substructure. Nonetheless, we impose a restriction on the minimum length of a run (currently set to four) to avoid a proliferation of very small runs that could be more of an overhead than a benefit. There is also another subtle issue when mapping the detected runs into CSX units: all runs—except those that start at the beginning of a row—miss the first element of the real substructure. For example, in Figure 5.5, the length of the real units are 5 and 4, instead of the detected runs of 4 and 3, which miss the 10 and 21 column indices, respectively. This can be crucial for the detection of 2-D substructures, where additional alignment limitations exist. In our actual implementation,



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 5.5: Example of the run-length encoding of the column indices. Note the missed element at the left of each run. This inherent issue of run-length encoding is fixed in the CSX implementation and allows an even higher compression of the input matrix.

```
function RLEncode($colind$::in)
$colind$: sorted column indices
    $deltas \leftarrow$ DeltaEncode($colind$)
    $d \leftarrow deltas[0]$                         ▷ current delta value
    $l \leftarrow 1$                                 ▷ current sequence length
    $rle \leftarrow \{(d, l)\}$                      ▷ set of column index runs
    for $i \leftarrow 1$ to $N$ do
        if $deltas[i] = d$ then
            $l \leftarrow l + 1$
        else
            if $l \geq min\_run\_length$ then
                $rle \leftarrow rle \cup (d, l)$
            $d \leftarrow deltas[i]$
            $l \leftarrow 1$
    end for
    return $rle$
```

Algorithm 5.1: Run-length encoding of the column indices. This is the simple implementation without the fix of CSX (see text). The DeltaEncode() function performs a delta encoding of the column indices and returns the sequence of delta values.

we fix this issue and also split large runs into 255-element chunks, so as to fit in the one-byte size field of a CSX unit.

The process of scanning the whole matrix for a specific type of substructures is described in Algorithm 5.2. More specifically, we iterate over the rows of the matrix and for each row we gather all the column indices that are not yet part of a substructure and apply run-length encoding. For all the detected substructure instantiations (different delta values), we keep statistics (number of units and number of non-zero elements covered) to guide us on the final selection of the substructures to encode in CSX.

```
1: procedure DetectSubstr($matrix$::in, $stats$::inout)
   $matrix$: CSX's internal repr., lexicographically sorted
   $stats$: substructure statistics
2:     $colind \leftarrow \emptyset$                                ▷ Column indices to encode
3:     foreach row in $matrix$ do
4:         foreach generic element $e(i, j, v, t)$ in row do
5:             if $e.t = \text{NONE}$ then                       ▷ e is not a substructure
6:                 $colind \leftarrow colind \cup e.j$
7:                 continue
8:             $enc \leftarrow$ RLEncode($colind$)
9:             CollectStats($stats$, $enc$)    ▷ Collect statistics from this encoding
10:            $colind \leftarrow \emptyset$
11:        end for
12:        $enc \leftarrow$ RLEncode($colind$)
13:        CollectStats($stats$, $enc$)        ▷ Collect statistics from this encoding
14:        $colind \leftarrow \emptyset$
15:    end for
```

Algorithm 5.2: Detecting substructures in CSX's internal representation. The CollectStats() routine collects and keeps track of the statistics for each substructure type and its instantiations. The information gathered in *stats* will be used later for guiding the selection of the most suitable substructure for encoding.

Detecting non-horizontal substructures

In our previous discussion on the detection process, we have not mentioned about horizontal substructures specifically. Indeed, the detection process 'sees' just column indices and the only requirement is that these indices must be sorted. Detecting horizontal substructures is therefore straightforward, since the matrix elements are arranged in row-wise (horizontal) order and are lexicographically sorted by default. To detect non-horizontal substructures then, it suffices to transform the matrix elements into the desired iteration order, sort them lexicographically and use the DetectSubstr() procedure described in Algorithm 5.2. The case of one-dimensional, non-horizontal substructures is straightforward and Table 5.2 shows the exact coordinate transformations that we use for their detection. The case of 2-D substructures, though, is a bit more complex, since we need to 'linearize' the coordinates of the elements.

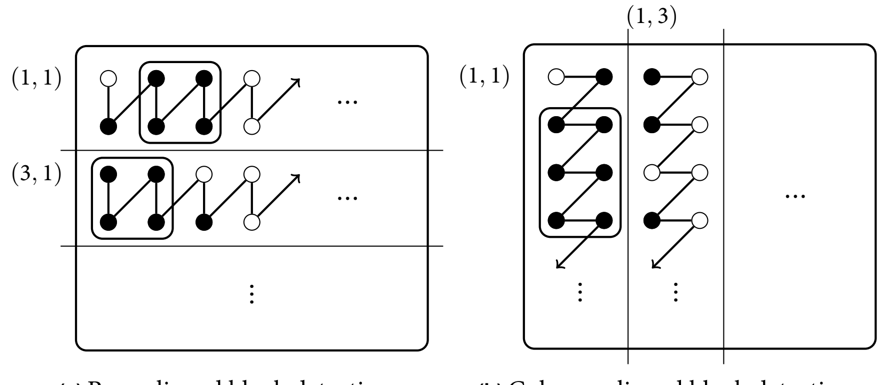
Unfortunately, a simple space-filling curve iterating over the non-zero elements is not an option for a sparse matrix for two major reasons: first, such a

Substructure type	Transformation
Horizontal	$(i', j') = (i, j)$
Vertical	$(i', j') = (j, i)$
Diagonal	$(i', j') = (N + j - i, \min(i, j))$
Anti-diagonal	$(i', j') = \begin{cases} (i + j - 1, i), & i + j - 1 \geq N \\ (i + j - 1, N + 1 - j), & i + j - 1 > N \end{cases}$
Block (row aligned)	$(i', j') = (\lfloor \frac{i-1}{r} \rfloor + 1, \mathrm{mod}(i - 1, r) + r(j - 1) + 1)$
Block (column aligned)	$(i', j') = (\lfloor \frac{j-1}{c} \rfloor + 1, c(i - 1) + \mathrm{mod}(j - 1, c))$

Table 5.2: The coordinate transformations applied by CSX on the matrix elements for enabling the detection of non-horizontal substructures (one-based indexing assumed).

curve would require an $\Theta(N^2)$ linear index space, which is far larger than the $\Theta(NNZ)$ original column index space, and, second, it would imply fixed size, strictly aligned blocks, an option we wanted to avoid in the first place. The approach we take for the 2-D substructures in CSX is depicted in Figure 5.6. We segment the matrix into fixed-width bands, either row- or column-aligned, and apply a space-filling transformation inside every band, passing the resulting sequence to the DetectSubstr() procedure. However, care must be taken during the detection, since now not all detected units are valid blocks; for a unit to be valid, it must begin at a column index (in the transformed space), which is a multiple of the band width. Using this segmentation technique, we have managed to detect loosely aligned, variable-sized blocks, further increasing the compression potential of CSX. We support two categories of block substructure types, namely row-aligned and column-aligned blocks, and each category defines seven different substructure types for different widths of the segmentation bands ($r, c \in [2, 8]$ in Table 5.2). In total, the use of coordinate transformations has allowed CSX to support seamlessly 18 different substructure types and enables the straightforward expansion to other substructure families, e.g., diagonal blocks.

Figure 5.7 shows the substructures detected by CSX in our matrix suite and is very characteristic of the capability of CSX in detecting a variety of substructures inside a sparse matrix. It is very interesting that CSX was able to detect and encode even 'weird' substructures, such as the diagonal ones with delta distances of 857 and 1714 in the pre2 matrix. In matrices dominated by dense blocks, CSX was able to detect large blocks in significant amounts, as is the case of the inline_1 and af_k_101 matrices. The detection of large dense blocks not only has a positive impact on the overall compression ratio, but also provides a performance advantage in the SpMV computations. Finally, the *substruc-*



(a) Row-aligned block detection. (b) Column-aligned block detection.

Figure 5.6: Detection of 2-D substructures in CSX (black dots denote non-zero elements). The matrix is segmented in row- or column-boundaries and a space-filling transformation is applied inside every segment. Simple run-length encoding of the resulting sequence suffices to detect variable-size, loosely aligned blocks (care must be taken only for truncating unnecessary elements at the beginning and end of the sequence).

ture map of a sparse matrix produced by CSX can be very revealing for the specific performance behavior of SpMV. For example, the matrices Freescale1, parabolic_fem and offshore are dominated by delta units with very large delta distances meaning that they are quite sparse and rather random. Indeed, their performance is poor using any format, suffering from irregular accesses in the input vector and load imbalances.

Substructure types and their instantiations

In CSX we make a distinction between a substructure type and its instantiation. In CSX's terminology, a substructure instantiation denotes how exactly a substructure type is encountered inside the sparse matrix. For example, the horizontal substructures with $d = 1$ and $d = 20$ of Figure 5.5 belong to the same substructure type (horizontal), but are different instantiations. Similarly, the blocks 2×10 and 2×20 are instantiations of the same block, row-aligned, $r = 2$ type. A substructure type, therefore, may have indefinitely many instantiations in a sparse matrix, which adds great flexibility to the CSX format, allowing it to detect almost every non-zero pattern inside a sparse matrix. The 6-bit `id` field in the `ctl` structure of CSX (Figure 5.2) refers to the exact substructure instantiation being encoded by the related unit; it does not refer to an abstract substructure type. This has the downside of limiting the total number of substructure instantiations in a sparse matrix to 64, but, in practice, only 4–5 instantiations are selected for encoding (see Figure 5.7).

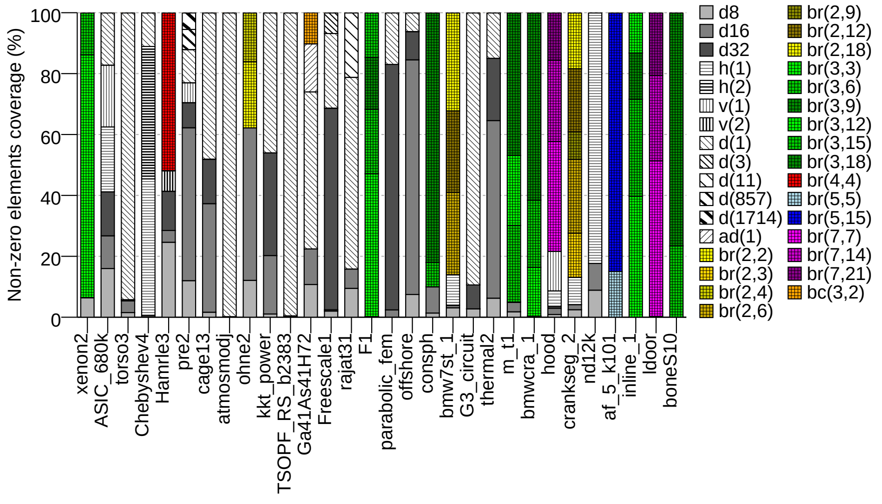


Figure 5.7: The substructures detected and encoded by CSX in a diverse set of sparse matrices. *[Substructure legend – dXX: delta units, h(X): horizontal, v(X): vertical, d(X): diagonal, ad(X): anti-diagonal, br(X,X): block row-aligned, bc(X,X): block column-aligned; numbers inside parentheses are the delta distances or the block dimensions; numbers in delta units denote their representation size in bits.]*

5.3.2 Selecting substructures for final encoding

The full detection process and the selection of the substructures for the final encoding is described in Algorithm 5.3, which is a typical local search optimization algorithm. More specifically, for each available substructure type, we transform the matrix to the corresponding iteration order and scan it, collecting statistics for the examined substructure type. After having collected statistics for all the available substructure types, we filter out all the instantiations that fail to surpass a certain threshold in encoding the non-zero elements of the matrix (in our current implementation, this is set to 5% of the total non-zero elements) and proceed with the selection of the most suitable type for encoding the matrix (SelectType()). The algorithm then proceeds by encoding the matrix with the selected substructure type (EncodeSubstr()) and repeating the same procedure until no type can be selected.

The criterion we use for selecting the substructures to encode is a rough estimate of the achieved reduction in the size of the original CSR's `colind` structure. Assuming that we keep a single, full column index per detected substructure (ignoring delta units), the size of the `ctl` structure will be

$$S_{ctl} = \overbrace{N_{units}}^{\text{encoded}} + \overbrace{NNZ - NNZ_{enc}}^{\text{unencoded}}, \tag{5.1}$$

```
1: procedure EncodeMatrix(matrix::inout)
   matrix: CSX's internal matrix in row-wise order
2:     repeat
3:         $stats \leftarrow \emptyset$
4:         for all available substructure types $t$ do
5:             Transform(matrix, t)
6:             LexSort(matrix)
7:             DetectSubstr(matrix, stats)
8:             $\text{Transform}^{-1}(matrix, t)$
9:         end for
10:        FilterStats(stats)          ▷ Filter out instantiations that encode less
                                         than 5% of the non-zero elements
11:        $s \leftarrow$ SelectType(stats)
12:        if $s \neq$ NONE then
13:            Transform(matrix, s)
14:            LexSort(matrix)
15:            EncodeSubstr(matrix)          ▷ Encode the selected substructure
16:    until $s =$ NONE
```

Algorithm 5.3: 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 (see Table 5.2), a lexicographic sort (LexSort()) of the non-zero elements is needed. The $\text{Transform}^{-1}()$ function transforms back the matrix into the original, horizontal iteration order.

where N_{units} is the total number of encoded substructure units and NNZ_{enc} 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

$$G = NNZ - S_{ctl} = NNZ_{enc} - N_{units}, \tag{5.2}$$

which is the metric that a substructure type must maximize in order to be selected for encoding.

5.3.3 Building the CSX data structures

The final step in the CSX matrix construction is the build of the actual CSX data structures, namely the `ctl` and `values` arrays (see Section 5.2). The process, depicted in Algorithm 5.4, is straightforward. We iterate the encoded internal matrix representation and for each encountered unit, either substructure or delta, we emit the appropriate fields of `ctl`. The delta units are constructed from stray elements not belonging to a specific substructure. These elements are collected during the iteration of the encoded internal representation and are "dumped" to the `ctl` and `values` arrays when either a new substructure is encountered or a new row starts. During construction, we also keep track of empty rows (not shown in Algorithm 5.4 for simplicity), in order to set appropriately the `rjmp` and `ujmp` fields of `ctl`, while at the start of each non-empty row we set the new row bit for the next encoded unit.

Responsible for constructing the `ctl` fields and emitting the non-zero values from the CSX units of the internal matrix representation is the set of the EMIT*() functions. The EMITVALUES() function is quite straightforward: it simply copies the non-zero values of its generic element(s) argument to the `values` array of the final CSX matrix structure. The EMITSUBSTRUCTURE() and EMITDELTAUNIT() functions perform similar operations: they both fill the `id`, `size` and `ucol` fields of the `ctl` byte array (see Figure 5.2), whereas EMITDELTAUNIT() emits also the delta values of the encoded delta unit. These functions take also care for setting unique IDs for every new substructure or delta unit instantiation encountered.

5.4 Generating the SpMV code

The indefinitely large and a priori unknown number of substructure instantiations inside a sparse matrix dictates the use of runtime code generation for the substructure-specific SpMV routines. We have built a new Just-In-Time (JIT) compilation framework for CSX based on Clang and LLVM [Lattner and Adve, 2004; Lattner, 2011]. LLVM is a low-level compiler infrastructure that provides a collection of modular and reusable compiler and toolchain technologies. Clang is a C language family front-end for the LLVM infrastructure. Its main purpose is to parse normal C/C++ source code, convert it to the LLVM's Intermediate Representation (IR) and pass it to the LLVM backend for the generation of the final native code. The great advantage of Clang and LLVM is that they provide a very rich programming interface that allows the development of efficient JIT code and its easy integration into an existing application.

In CSX, we generate simple C code for the substructure-specific SpMV routines and use Clang to generate optimized LLVM IR. The overhead of parsing the resulting C source (one translation unit, less than 150 l.o.c.) and generating the optimized LLVM IR is negligible compared to the matrix preprocessing

```
 1: function ConstructCsx(matrix::in)
    matrix: CSX's internal encoded matrix representation
 2:     $ctl \leftarrow \emptyset$                              ▷ CSX's control byte sequence
 3:     $values \leftarrow \emptyset$                           ▷ CSX's non-zero values
 4:     $stray \leftarrow \emptyset$                            ▷ Stray elements, not in substructures
 5:     foreach row in matrix do
 6:         $ctl \leftarrow ctl \oplus$ EmitNewRow( )             ▷ Set new row bit
 7:         foreach generic element $e(i, j, v, t)$ in row do
 8:             if $e.t \neq$ NONE then
 9:                 EmitDeltas(stray, ctl, values)        ▷ Dump and empty
                                                           stray elements
10:                 $ctl \leftarrow ctl \oplus$ EmitSubstructure($e$)
11:                 $values \leftarrow values \oplus$ EmitValues($e$)
12:             else
13:                 $stray \leftarrow stray \cup e$
14:         end for
15:         EmitDeltas(stray, ctl, values)           ▷ Dump and empty stray elements
16:     end for
17:     EmitDeltas(stray, ctl, values)               ▷ Dump and empty stray elements
18:     return (ctl, values)
```

Algorithm 5.4: Construction of the CSX data structures. The final matrix is built incrementally by iterating over the encoded internal representation. The Emit*() routines are responsible for emitting the actual `ctl` fields and the non-zero values of the encoded substructure. Stray elements form delta units (see Algorithm 5.5 for more details on the EmitDeltas() procedure). The $\oplus$ operator denotes sequence concatenation. Treatment of empty rows is not shown for simplicity.

time to justify the extra pain of maintaining a pure LLVM IR codebase or even an on-disk cache of precompiled SpMV routines. An alternative strategy would be to generate the LLVM IR directly through the related library interface. However, this is a tedious and error-prone task (also in terms of performance), since not only a good understanding of the intermediate representation is required, but also the actual SpMV code is 'obfuscated' by the complex calls needed to construct the IR.

Figure 5.8 shows how the just-in-time compilation is organized in CSX. CSX maintains a directory of C source templates[1], which define a set of text

[1] Not to be confused with the C++ templates.

```
1: procedure EmitDeltas(elems::inout, ctl::inout, values::inout)
   elems: set of generic elements
   ctl: CSX's ctl structure
   values: CSX's non-zero values
2:     if $elems \neq \emptyset$ then
3:         $ctl \leftarrow ctl \oplus$ EmitDeltaUnit($elems$)
4:         $values \leftarrow values \oplus$ EmitValues($elems$)
5:         $elems \leftarrow \emptyset$
```

Algorithm 5.5: Helper procedure for emitting delta units.

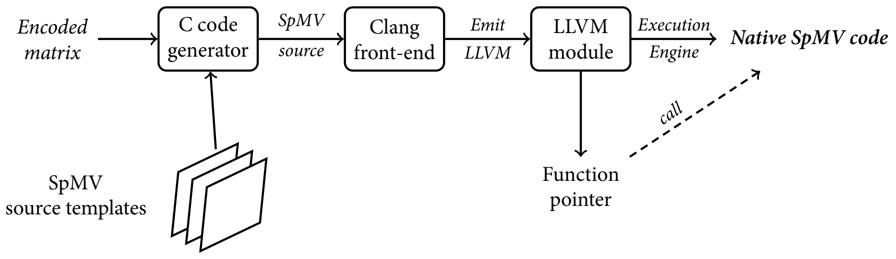


Figure 5.8: The just-in-time compilation framework of CSX.

hooks to be filled in the runtime. For each substructure type we maintain a source template for performing the SpMV computation inside the substructure[2], and we also have a 'top-level' template that is responsible for the execution of the full SpMV kernel in the CSX format. After we have selected the substructures for encoding and have generated the CSX's data structures, we pick the suitable templates and generate the corresponding SpMV C code passing it to the Clang front-end. We program the front-end to emit an optimized LLVM IR module, from which we get a function pointer to the generated SpMV kernel and, eventually, the LLVM back-end takes over to generate the final native code for the host machine.

The SpMV kernel that we execute for CSX is shown in Algorithm 5.6. First, we should note that in CSX we must explicitly zero out the output vector (lines 4–5), an artifact that is prescribed by the use of multiple non-horizontal substructures. However, the effect of this operation is very small, especially in multithreaded configurations, where the initialization is performed in parallel. The algorithm iterates over the `ctl` structure decoding a field at a time. If a new

[2] For block substructures we maintain just two generic templates: one for row-aligned blocks and one for column-aligned ones.

row is detected, we store the computed dot product (*yr*) in the output vector and advance the current position (y_{curr}) with the __NEW_ROW_HOOK(). If the matrix does not contain row jumps, the generated code simply advances y_{curr} by one. In the opposite case, it checks the `rjmp` bit and if there is a jump, it advances y_{curr} with the value of the variable size integer containing the jump size. The algorithm proceeds by decoding the variable size integer containing the delta distance from the previous column index (DecodeColumn()) and computes the starting column index (x_{curr}) of the current CSX unit. It then retrieves the ID of the unit and executes the unit-specific SpMV code within the __BODY_HOOK(). The __BODY_HOOK() is actually replaced by a C `switch` statement, switching on the *id* variable. The unit-specific SpMV routines are responsible for correctly advancing the current position in the `ctl` structure and the current column index, as well as updating the local accumulator *yr*, if necessary.

```
procedure CsxSpmv(ctl::in, values::in, x::in, y::out)
    $y_{curr} \leftarrow y$                          ▷ Current position in y vector
    $x_{curr} \leftarrow x$                          ▷ Current position in x vector
    for $i \leftarrow 0$ to $N$ do                   ▷ We must zero-out the output vector
        $y_{curr}[i] \leftarrow 0$
    end for
    $yr \leftarrow 0$                                ▷ Local accumulator
    repeat
        $flags \leftarrow *ctl$
        $size \leftarrow *(ctl + 1)$
        $ctl \leftarrow ctl + 2$
        if TestBit($flags$, 7) then                 ▷ Check if nr bit is set
            $*y_{curr} \leftarrow *y_{curr} + yr$
            $yr \leftarrow 0$
            __NEW_ROW_HOOK()                         ▷ Advances $y_{curr}$
            $x_{curr} \leftarrow x$
        $x_{curr} \leftarrow x_{curr} +$ DecodeColumn($ctl$)
        $id \leftarrow$ GetID($flags$)              ▷ Retrieve the ID of the next unit
        __BODY_HOOK()                                ▷ Unit-specific SpMV code
    until $ctl$ ends
```

Algorithm 5.6: The SpMV kernel template used the CSX storage format. The hooks are filled in during the runtime. The '$*$' is a dereference operator.

Since we encode specific substructure instantiations in CSX, we know during the compilation time the exact delta distances of the one-dimensional substructures and the exact block dimensions of the 2-D ones. This allows us to generate efficient code with hardcoded constant delta distances and constant block dimensions, matching the computational advantage of the fixed-size blocks of BCSR. Additionally, if only one substructure instantiation is encoded, we optimize out the switch statement completely.

Parallelization

CSX follows the parallelization pattern of CSR. During the construction of the CSX's internal representation, we split the input matrix row-wise maintaining roughly the same number of non-zero elements per partition. From this point on, the detection and encoding phases of CSX proceed independently, producing different final CSX submatrices per partition. The SpMV kernel in Algorithm 5.6 changes only in line 2, where we initialize y_{curr} to the beginning of the partition and in line 4, where we iterate only within the bounds of the partition.

5.5 Tackling the preprocessing cost

The preprocessing time of CSX is bounded by the multiple calls to the LexSort() routine, which performs a lexicographic sort on the transformed matrix elements, during the scanning for substructures (Algorithm 5.3, lines 4–9). We address this issue with a combination of partial sorting and sampling. More specifically, instead of scanning the matrix as a whole, we split it into constant size preprocessing windows based on the non-zero elements and scan every window separately. This modification reduces automatically the asymptotic complexity of the scanning phase from $\Theta(NNZ \lg NNZ)$ to $\Theta(NNZ)$ at the expense of missing the substructures that cross the boundaries of the windows. To further reduce the preprocessing cost, we only examine a certain number of windows uniformly distributed over the whole matrix covering only a small amount of the total non-zero elements. In our experiments, sampling a mere 1% of the matrix non-zero elements using 48 sampling windows reached the same SpMV performance levels as the full-fledged preprocessing at nearly an order of magnitude less preprocessing time.

Having minimized the substructure scanning phase, the preprocessing time is now bound from the final encoding of the matrix (Algorithm 5.3, lines 13–15), which is still in the order of $\Theta(NNZ \lg NNZ)$. In this phase, unfortunately, partitioning and, especially, sampling cannot be applied, since we need to encode the *whole* matrix, after all. Nonetheless, the impact of this phase on the overall preprocessing cost is not so crucial for two main reasons: in most of the cases a very small number of substructure types will be finally encoded, and the full cost of the sorting will be paid only during the encoding of the first substructure type, since every time we sort only the unencoded elements.

In addition to the use of sampling and preprocessing windows, we further reduce the preprocessing cost by parallelizing the whole preprocessing process. The parallelization is straightforward: after we have loaded the initial CSR matrix, we setup a new thread for every partition that proceeds independently with the scanning and the construction of the final CSX matrix.

Finally, we take particularly care of the memory management during the preprocessing, by avoiding completely memory reallocations that would result in huge memory copies. For this reason, we try to infer the exact allocation requirements for a data structure right from the beginning or, in case this is not possible, we are generous with the initial allocation and then truncate the extra space. This optimization has led to a considerable acceleration of the preprocessing phase, reaching a $2\times$ to $3\times$ speedup. The preprocessing cost of CSX for detecting all the supported substructure types ranges at the order of 100 serial CSR SpMV operations, rendering it viable even for online processing of the input matrix.

5.6 Porting to NUMA architectures

As discussed in Chapter 3, the key factor for the high performance of SpMV in NUMA architectures is the correct placement of the involved data on the system's memory nodes. The data each thread accesses must lie on its local memory node, in order to avoid the saturation of the interprocessor links and also the increased latency incurred by the multiple hops required for a remote access. A thread in the multithreaded SpMV kernel accesses the matrix data structures of its own matrix partition, the corresponding parts of the output vector and arbitrary parts of the input vector. The correct placement of the matrix partitions in CSX is straightforward: since the construction of the CSX matrix happens independently for every partition, we just need to make sure that the CSX's data structures are allocated on the correct node using calls to a NUMA-aware memory allocator, e.g., the `numa_alloc_onnode()` of the Linux *numactl* library. For the output vector, we use our low-level interleaved NUMA-aware allocator described in Chapter 3, in order to place correctly the different parts of the output vector. Finally, for the sake of our experimentation, we place a copy of the input vector on every memory node. This arrangement provides a better balancing of the overall memory operations and exposes further the computational part of the kernel. Although in practical applications this choice is less reasonable, the impact on performance of using a shared input vector copy is minimal on average (see Chapter 3, Figure 3.13).

5.6.1 Optimizing the computations

Modern NUMA architectures have an important side-effect on the execution of the SpMV kernel: the increased memory bandwidth when accessing a local memory node, exposes more the computational part of the kernel, which is no longer negligible. CSX performs some heavy decompression operations in decoding the variable size integers (see Figure 5.4) used to store the column delta distances and the row jumps. The case of row jumps is not of considerable concern, since we do not generate the decoding code at all for matrices that do not have them, and in cases they do, the decoding is not in the critical path. What stays in the critical path, though, is the decoding of the column delta distance before computing every substructure (Algorithm 5.6, line 17). The decoding is trivial if the delta distance is less than 128, but it demands hefty bit operations if it is larger. Unfortunately, when encoding substructures (and not only delta units[3]) there is a proliferation of multi-byte column delta distances, whose decoding can considerably hinder the overall SpMV performance. For this reason, we replace the column delta distance with the full starting column index of the substructure, stored in a standard 32-bit integer. This optimization degrades the overall compression ratio 2–3%, but the gain in performance can exceed 15% in certain matrices.

The proliferation of delta units when encoding multiple substructures can also overwhelm the benefit of compression in NUMA architectures due to the increased performance overhead of the switch statement (branch mispredictions). For this reason, we revise our score function (equation (5.2)) for NUMA architectures to include an estimate of the total switch statements executed as follows:

$$G = NNZ_{enc} - N_{units} - N_{switch}, \tag{5.3}$$

where $N_{switch} = N_{units} + N_{deltas}$, N_{deltas} being the total number of delta units generated. This heuristic balances better the tradeoff between the memory and the computational part of the kernel by penalizing the encodings that lead to a large computational cost. We have encountered more than 20% performance improvement for certain matrices in NUMA-aware thread configurations with the revised heuristic.

[3] The case of using only delta units is not critical, because we encode the full matrix row as a single delta unit.

5.7 Evaluating the performance of CSX

We evaluate the performance of the CSX storage format using the matrix suite and the hardware platforms detailed in Chapter 3, Section 3.2. As alternative storage formats, besides CSR, we consider BCSR and VBL, since these are the most established paradigms of CSR alternatives and the best representatives of the fixed and variable size blocking storage formats, respectively. We have implemented optimized versions of all these formats, including NUMA-aware implementations. Specifically for BCSR, we have implemented block-specific, optimized SpMV routines for all the block sizes (one- and two-dimensional) up to size eight plus the 3×3 block. The results reported for BCSR correspond to the best performing block, which was obtained after an exhaustive search of the 20 available blocks. In practice, where a heuristic will be most probably used (see Chapter 4, Section 4.3), the real performance of BCSR might be less. For the parallelization of the SpMV routines, we use a static, row-wise partitioning scheme based on the non-zero elements of the matrix. Specifically for BCSR, we partition the input matrix after we have built it, taking into account the zero-padding as well, in order to achieve a better load balance. We use 32-bit integers for the indexing structures of all the storage formats and 64-bit, double precision floating point values for the non-zero elements. In the case of VBL, we use one-byte block size fields.

Finally, we used LLVM 2.9 for the compilation of the SpMV routines for all the considered formats, in order to achieve a fair comparison. We should note here that beyond our initial expectations, LLVM 2.9 offered an average 5% performance improvement to the non-CSX[4] formats compared to GCC 4.5. For the parallelization of the SpMV routines and the preprocessing phase of CSX, we used explicit, native threading with the Pthreads library (NPTL 2.7) and bound the threads to specific logical processors using the Linux kernel's system call interface. We follow a 'share-all' policy for the assignment of threads to logical processors by default (see Chapter 3, Section 3.2.3 for more details on alternative core-filling strategies).

5.7.1 CSX compression potential

The compression potential of CSX is favored not only by its ability to detect multiple types of substructures, allowing to keep only a single column index per substructure, but also by the very compressed representation of the non-zero elements metadata. The column index information is further condensed with the use of delta indexing and variable size integers, while the row information is now represented by a single bit per row. This allows CSX to compress

[4] CSX routines are generated using LLVM by default.

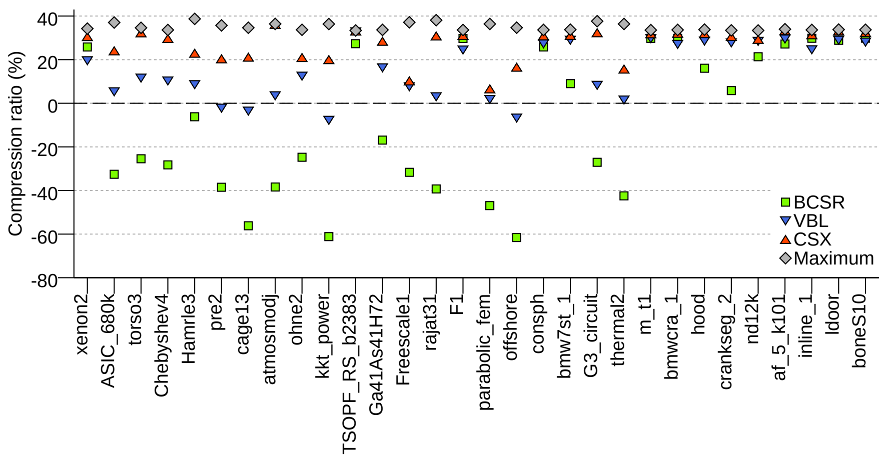


Figure 5.9: The compression potential of CSX. CSX manages to adapt successfully to the matrix structure and provides the best compression ratio among the alternative storage formats considered. Maximum compression ratios are calculated by assuming storage of the non-zero elements only, i.e., a matrix representation size of $8NNZ$ bytes (double-precision floating point values).

the representation size, even for matrices with an irregular structure, where no substructures are encoded. This is a key advantage of CSX compared to other alternative storage formats, especially in SMP systems.

Figure 5.9 shows the maximum compression ratio achieved by CSX compared to VBL and BCSR. The absolute maximum compression ratio (assuming only the non-zero elements are stored) is also reported. CSX achieves always the best compression ratio among the storage formats considered, while its ability to detect and encode multiple different substructures allows it to adapt to the specificities of each matrix. For example, it is able to surpass the compression ratio of BCSR in block-dominated matrices, such as xenon2, TSOPF_RS_b2383, consph, m_t1 etc., and also VBL's in less regular ones, such as Freescale1 and offshore. In matrices dominated by diagonal elements, e.g., torso3, cage13, atmosmodj etc., the predominance of CSX is clear: it manages a more than 20% compression ratio, while BCSR augments the matrix size and VBL stays close to the original CSR's size. The most characteristic example of this case is the atmosmodj matrix, where 99.8% of its non-zero elements are encoded in diagonal patterns by CSX, reaching the maximum possible compression (36.4%)[5]. At the same time, VBL manages a mere 3.4% compression, while BCSR increases the matrix size by nearly 60%!

[5] CSX actual compression ratio for this matrix is 36.1%.

5.7.2 CSX performance

SMP architectures

The large compression capability of CSX provides a strong performance potential in matrices arising from PDE problems, where the key performance bottleneck, especially in SMP systems, is the lack of adequate memory bandwidth resources. Figure 5.10 shows the speedups achieved by all the considered methods in our two symmetric shared memory platforms, Harpertown and Dunnington. The effect of compression is dominant in these architectures, especially in the multithreaded configurations. CSX and VBL take the lead with an average 26.4% and 18.5% performance improvement over CSR in the eight-threaded configuration in Harpertown, respectively, while BCSR gains a mere 4.1%. Similar is the picture in the 24-thread configuration in Dunnigton: CSX offers a 61% performance improvement over CSR on average, VBL follows further behind with a 28.8% improvement, while BCSR is able to achieve only 6.3% improvement. BCSR suffers from its inherently low compression capability and the use of padding to construct full blocks. It can be classified as an 'all-or-nothing' storage format, since in almost half of the matrices of our suite, it degraded SpMV performance more than 30% on average, while for the other half matrices it provided a more than 25% performance improvement. The higher compression potential of VBL and CSX is clearly reflected by the speedup diagrams in our two symmetric shared memory architectures. The ability of CSX to detect and encode multiple types of substructures, and especially blocks, is a significant advantage of CSX over VBL, not only in terms of pure compression ratio, but also in terms of the involved SpMV computations, since CSX keeps the computational advantage of the BCSR's fixed size blocks, thanks to the runtime code generation, without paying the padding overhead at all. Indeed, like BCSR, CSX matches the average performance of CSR in Harpertown using one thread, despite the additional overhead of zeroing the output vector in every iteration. In Dunnington, CSX is starting off with a significant 16.1% performance advantage over CSR right from the single-threaded configuration. We should note here the steep increase of speedup in Dunnington for 12 and 24 threads; this is due to the linear increase of the available memory bandwidth resources as we add more CPU sockets (i.e., front-end bus controllers) to the computation (see Chapter 3, Section 3.3.2), due to the 'share-all' core-filling strategy.

Using a 'share-nothing' core-filling strategy that utilizes the maximum available system's memory bandwidth (Figure 5.11), the big picture does not change; CSX's predominance in SMP systems is clear. It reaches a 47% average performance improvement over CSR for the 12-threaded configuration, while VBL

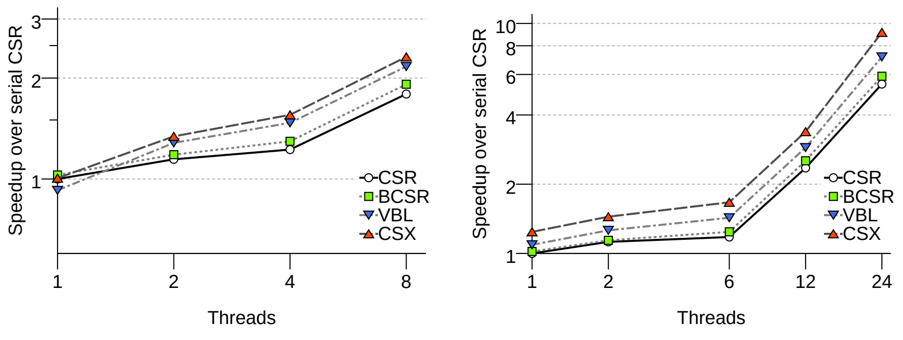
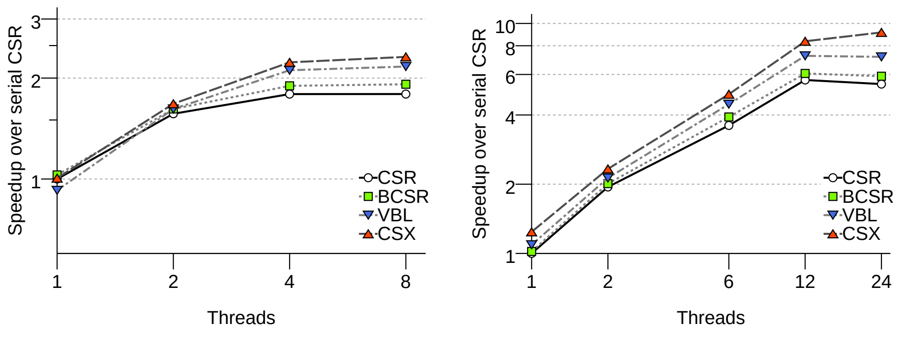


Figure 5.10: CSX speedup in SMP systems. The high compression potential of CSX allows it to improve significantly the SpMV performance, surpassing all other CSR alternatives. The average speedup is depicted using a 'share-all' core-filling policy.

achieves a 24% improvement and BCSR is almost at 5%. An important characteristic of CSX in Dunnington is that it continues to improve its performance as we increase the thread count, while all other formats experience a performance slowdown that reaches 4% in the case of CSR. In Harpertown, the common bus is almost completely saturated from the four-threaded configuration, while in Dunnington the speedup reaches a plateau at the 12-threaded configuration. The significant scaling encountered by all formats for the six- and 12-threaded configurations in Dunnington is due to the very large, 64 MiB, aggregate L3 cache available. Six matrices from our suite fit comfortably in the available cache even in CSR format, while three more are only a little larger. The system is finally completely saturated in the 24-threaded configuration, where all formats except CSX encounter a performance slowdown.

NUMA architectures

In NUMA architectures, where the available memory bandwidth is considerably higher, the performance landscape changes, but CSX remains still the most performant storage format across the full range of multithreaded configurations. Figure 5.12a shows the speedups of the considered storage formats in Gainestown using a NUMA-aware data placement. The very first observation is that CSR is now rather competitive, since the memory bottleneck is not so intense as before until the 16-threaded configuration, where HyperThreading is also enabled. CSX achieves a 17.5% performance improvement over CSR when using eight threads and increases the gap to 20.7% in the 16-threaded config-

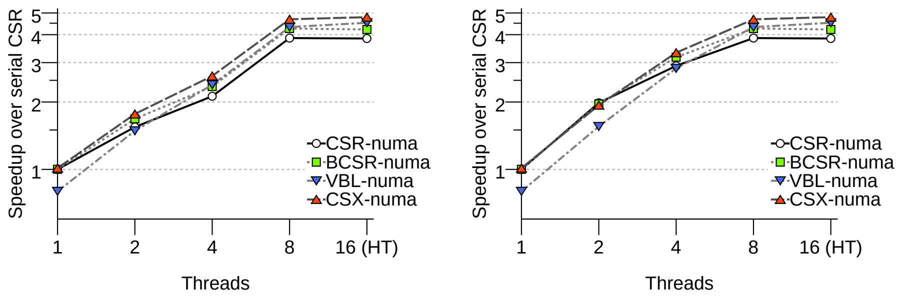


(a) Harpertown. (b) Dunnington.

Figure 5.11: CSX speedup in SMP systems using a 'share-nothing' core-filling policy for utilizing the maximum available system's bandwidth as early as possible. Average speedups are depicted.

uration, where the contention for shared resources by the hardware threads becomes visible. The respective numbers for VBL are 8.4% and 13.9% for the two aforementioned configurations. The case of BCSR is interesting, since it manages to achieve a performance comparable to VBL, thanks chiefly to the ample memory bandwidth that better exposes its computational advantage. While VBL's performance falls behind CSR's up to the two-threaded configuration, reaching a 23.5% performance degradation for the single-threaded one, CSX experiences a slight 2.8% performance degradation only for the single-threaded configuration. From the two-threaded configuration, however, CSX is already ahead of every other format having achieved an 11.4% performance improvement over CSR and reaching 20.7% when the full system is utilized.

In order to examine the behavior of CSX in configurations where the memory bandwidth is not an issue, we have conducted experiments in Gainestown using a 'share-nothing' core-filling policy (see Figure 5.12b). In these configurations, the increased memory bandwidth offered up to the four-threaded configuration favors the simpler CSR and BCSR formats, which achieve comparable performance. VBL falls victim of the increased overhead incurred by its additional data structures and it is not until the eight-threaded configuration that compensates this cost. On the other hand, CSX manages to stay very close to CSR and BCSR for the first two configurations, despite its increased bookkeeping and decompression cost, experiencing only a slight 3.6% performance degradation in the two-threaded configuration. From the four-threaded configuration onward, however, CSX builds a significant 11.3% performance gap over CSR, which is further expanded to 17.5% and 20.7% in the eight- and 16-



(a) Share-all policy. (b) Share-nothing policy.

Figure 5.12: CSX speedup in NUMA architectures. Average results for the Gainestown system using the two different core-filling policies are shown. Even with 'share-nothing' policy, CSX manages to stay very close to BCSR and CSR up to the two-threaded configuration, before taking the lead in more threads.

threaded configurations, respectively.

The speedup results in NUMA architectures revealed a significant performance stability advantage of CSX, since even in configurations where the memory bandwidth is not an issue, it manages to match performance of CSR and BCSR. This behavior is further illuminated by examining the per-matrix performance results for eight threads in Gainestown presented in Figure 5.13. In this configuration, the demands in memory bandwidth are quite significant, but the memory subsystem is not yet saturated. Observing the per-matrix performance results, we can first distinguish two matrix categories:

(a) low-performing matrices (≈3 Gflop/s in CSR) and

(b) high-performing matrices (≈5 Gflop/s in CSR).

The first category is formed by matrices with an irregular non-zero element structure and very short rows; according to the discussion in Chapter 3, SpMV's performance in these cases 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. The second category, which is the most typical in SpMV applications, consists of matrices with a more regular structure arising mostly from the discretization of PDEs; the key problem here is the contention for memory bandwidth. CSX manages to achieve considerable performance improvements in matrices of the second category, approaching the upper bound of dense-matrix vector multiplication performance in this configuration (8.5 Gflop/s). In the low-performing matrices, CSX exhibits a rather stable behavior, matching or even surpassing the performance of CSR.

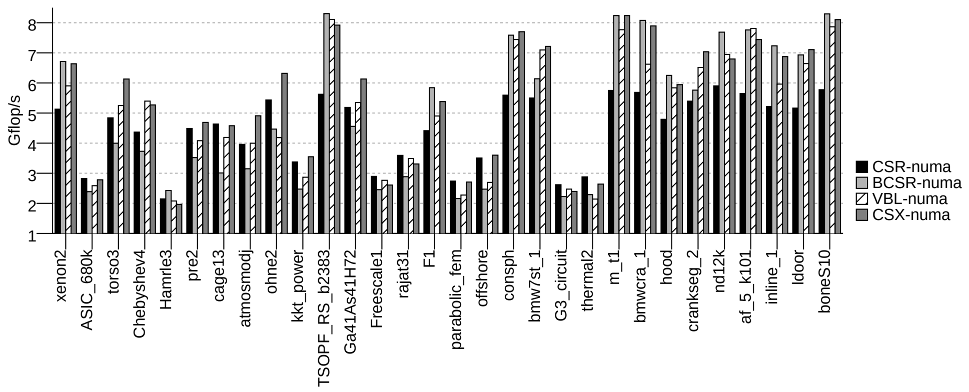


Figure 5.13: Per-matrix performance in Gainestown (eight threads, NUMA-aware). CSX provides consistently high performance from irregular to regular matrices, while other CSR alternatives take an 'all-or-nothing' approach.

Although VBL and BCSR exhibit similarly high performance in more regular matrices, with BCSR matching the performance of CSX in block-dominated matrices (xenon2, consph, m_t1, bmwcra_1, inline_1), the situation changes for irregular matrices, where both BCSR and VBL tend to exhibit a significant performance degradation, due to their increased overhead in matrix size and computations. The advantage of CSX is also clear in matrices dominated by diagonal substructures (e.g., torso3, cage13, atmosmodj), which cannot be efficiently exploited by the alternative formats considered.

Performance stability

The performance stability of CSX is quantitatively exhibited in Table 5.3, where we provide some statistics for eight-threaded, NUMA-aware configuration in Gainestown, an achitecture that favors more computational-friendly formats[6]. The competitiveness of CSR and BCSR is clear from this table, gaining seven and nine matrices, respectively. Nonetheless, even in this not so favorable configuration, CSX manages to achieve the absolute best performance in the majority of matrices (12), while VBL contents itself to just two matrices. The most important information of this table, however, are the performance differences of each format from the best overall performance. Specifically, for each considered format, we present its average performance difference from the absolute best for all the matrices that it did not gained; we also present a 95% confidence interval (C.I.). These metrics confirm the predominance of CSX both in terms

[6] In Harpertown and Dunnington, the predominance of CSX is almost total, providing the best performance in 26 out of the 30 matrices of our suite.

	CSR	BCSR	VBL	CSX
Best perf.	7	9	2	12
Differences from best				
Average	21.15%	20.13%	12.03%	5.79%
95% C.I.	±3.53%	±4.99%	±2.94%	±2.16%
Minimum	2.60%	0.01%	1.62%	1.13%
Maximum	32.27%	35.86%	33.79%	19.04%

Table 5.3: Performance stability of the different storage formats examined compared to CSX (Gainestown, eight-threaded, NUMA-aware configuration).

of overall SpMV performance and in terms of adaptivity to the different characteristics of each matrix. Despite obtaining the absolutely best performance in less than half of the matrices, CSX manages to be as close as 5.8% to the overall best performance. VBL follows further back with an average difference of 12% from the best performance, while BCSR and CSR come last with 20.1% and 21.2% differences, respectively. The stability of CSX is also superior to the other considered formats as it is depicted by the computed confidence intervals, while BCSR, as expected, is the least stable format with a performance variation of 5%. CSX can be therefore considered as a high performance storage format for sparse matrices, since not only achieves the highest performance in the majority of matrices in both SMP and NUMA architectures, but also manages to stay very close to the overall best performance in the rest of the matrices, including corner cases, such as very irregular matrices, exhibiting significant performance stability compared to other alternatives.

Performance on special-structured matrices

Despite being a matrix-agnostic format, CSX is able to detect the structure of the input matrix and provide performance comparable to specialized formats for matrices with a special structure, e.g., diagonal, banded matrices etc. As a proof-of-concept example, we compare the performance of CSX with the performance of the Dense Matrix-Vector multiplication kernel (DMV) on a 2000×2000 dense matrix. Table 5.4 shows the performance of both kernels for the Dunnington and Gainestown systems. Despite being a generic format, it is clear that CSX, without any prior knowledge, can match the performance of a special-purpose format (here dense matrix format). This was possible because CSX has detected and encoded a single substructure (horizontal substructures, no delta units) and our code generator has completely optimized out the switch statement by using a single inline call to the substructure-specific code. The lit-

	1 core	*1 socket*	*4 sockets*
DMV	558 Mflop/s	755 Mflop/s	10095 Mflop/s
CSX	547 Mflop/s	751 Mflop/s	9922 Mflop/s
CSR	404 Mflop/s	473 Mflop/s	5509 Mflop/s

(a) Dunnington.

	1 core	*1 socket*	*2 sockets*
DMV	1827 Mflop/s	4950 Mflop/s	8562 Mflop/s
CSX	1766 Mflop/s	4872 Mflop/s	8442 Mflop/s
CSR	1526 Mflop/s	3117 Mflop/s	5658 Mflop/s

(b) Gainestown.

Table 5.4: DMV vs. CSX performance. Despite being matrix-agnostic, CSX matches the performance of dense matrix-vector multiplication.

tle overhead encountered is incurred by the 255-element unit limit and the rest of CSX's bookkeeping. The behavior will be similar for a banded matrix, since CSX will detect diag(1) units only.

The ability of CSX to adapt automatically to the structure of the matrix offers great flexibility, since it can provide near optimal performance for a variety of different matrices without any prior knowledge of the non-zero element structure. These results emphasize the adaptability of CSX, which also matches or surpasses the performance of CSR in irregular matrices.

The effect of matrix reordering

Some of the matrices in our matrix suite depart from the typical, memory bandwidth-bound behavior, and experience significant performance overheads, incurred chiefly by their irregular sparsity structure (see Figure 5.13). As discussed in Chapter 2 (Section 2.4), a common technique for 'homogenizing' the sparsity structure of a sparse matrix is matrix bandwidth minimization [Cuthill and McKee, 1969; Karypis and Kumar, 1995; Çatalyürek and Aykanat, 1999]. Such techniques operate on structurally symmetric matrices and use row and column permutations, in order to bring the non-zero elements as close as possible to the main diagonal. A successful matrix reordering will render the access pattern in the input vector more regular and will also homogenize the flop:byte ratio across the whole matrix, therefore leading to higher performance levels. We have applied the Reverse Cuthill-McKee (RCM) [Cuthill and McKee, 1969] reordering algorithm in low-performing and structurally symmetric matrices from our suite, namely parabolic_fem, offshore, G3_circuit and thermal2.

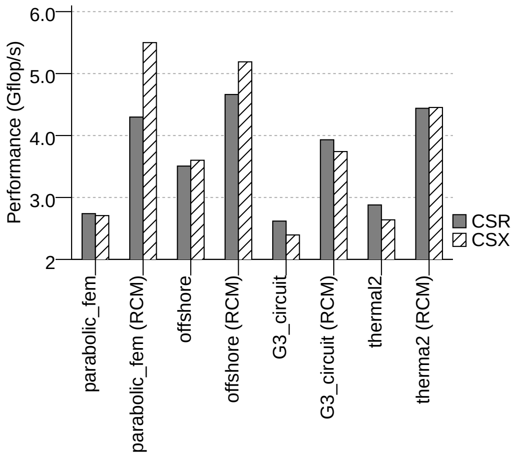
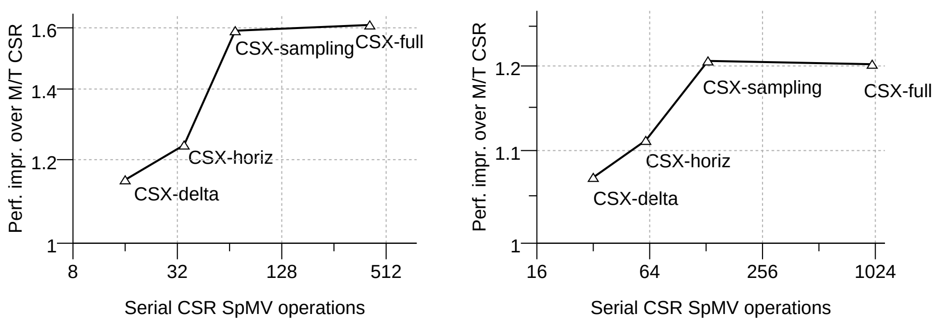


Figure 5.14: Performance of CSX and CSR in RCM reordered matrices. Matrix reordering techniques benefit more the CSX format, since the redistribution of the non-zero elements allows the detection of more substructures by CSX. Results shown are from the eight-threaded NUMA-aware configuration in Gainestown.

Figure 5.14 shows the performance of their original and reordered versions for CSR and CSX in the Gainestown platform using a NUMA-aware eight-threaded configuration. The performance gain is significant for both formats, but CSX is benefited more in all matrices considered. In parabolic_fem and offshore, CSX achieves a 28% and 11% performance improvement over CSR, respectively, while before reordering both formats achieved similar performance. The same behavior is observed for G3_circuit and thermal2, where CSX has managed to close the performance gap from CSR. A significant observation from this figure is that matrix reordering and similar input vector access optimization techniques have an orthogonal effect to the memory bandwidth minimization performed by CSX. Additionally, these techniques seem to favor CSX more, since they provide the grounds for detecting more substructures and therefore achieving higher compression levels.

5.7.3 CSX preprocessing cost

As discussed in Section 5.5, the preprocessing of CSX is bounded from the detection and encoding of non-horizontal substructures, which require a lexicographic sort of the matrix coordinates. An easy way to reduce to the prepro-



(a) Dunnington (24 threads). (b) Gainestown (16 threads, NUMA-aware).

Figure 5.15: The preprocessing cost of CSX. Statistical sampling of the matrix leads to an order of magnitude drop in the preprocessing cost, rendering CSX a viable alternative even for online preprocessing.

cessing cost is to program CSX[7] to detect only delta units or horizontal substructures. Nonetheless, this kind of preprocessing reduction comes at a cost of the overall CSX performance as it is depicted in Figure 5.15. To exploit its full potential, CSX must be programmed to detect as many substructure types as possible. In this case, however, the preprocessing cost climbs to several hundreds of serial CSR SpMV operations. Though not irrational even for on-line preprocessing, this cost is large and can eradicate the performance benefit of CSX in the midterm. For this reason, we employ uniform statistical sampling on the input matrix as described in detail in Section 5.5. The use of sampling can drop the preprocessing cost nearly an order of magnitude with a minimal impact in CSX's overall performance. A key aspect for sampling a sparse matrix with CSX is to use a lot of sampling windows scattered all over the matrix, in order to obtain a 'good look' of the whole matrix and avoid any over- or under-estimation of the presence of certain substructures. In our case, we have used 48 sampling windows for sampling only 1% of the total non-zero elements of the matrix.

Figure 5.15 shows the preprocessing cost of CSX measured in serial CSR SpMV operations in relation to the achieved performance improvement over the multithreaded CSR in Dunnington using 24 threads and in Gainestown using 16 threads. The cost when detecting only delta units (CSX-delta) or horizontal substructures (CSX-horiz) in Dunnington is very small ranging from 16 to 35 CSR SpMV operations, but only a 14.8% to 23.9% performance improvement is achieved. The use of sampling in detecting all the available substructure

[7] CSX can be configured to detect specific types of substructures or even to encode specific substructure instantiations.

types (CSX-sampling) increases the cost up to 69 CSR SpMV operations—an affordable number for on-line preprocessing of the matrix—providing a 59% performance improvement. The preprocessing cost increases significantly with the activation of full preprocessing (no windows, no sampling) surpassing the 400 serial CSR SpMV operations, only to offer a mere 1.3% additional performance improvement over CSR.

Similar is the picture in Gainestown, where sampling leads even to a slightly higher average performance. SpMV performance in Gainestown is not as directly related to matrix size representation as in SMP systems, therefore, some encoding decisions using sampling may lead to better computational characteristics, in spite of a larger representation size. Another difference between SMP and NUMA architectures, as revealed by Figure 5.15, is that the preprocessing cost is more exposed in NUMA architectures ranging from 32 to 985 serial SpMV operations, while in Dunnington this cost ranges from 16 to 492 operations.

5.8 Integrating CSX into multiphysics simulation software

As a further evaluation of the potential of CSX in optimizing SpMV in the context of a multiphysics application, we have integrated CSX into the Elmer [Lyly et al., 1999–2000] multiphysics simulation software. Elmer employs iterative Krylov subspace methods for treating large problems using the preconditioned Bi-Conjugate Gradient Stabilized (BiCGStab) method [van der Vorst, 1992] for the solution of the resulting linear systems. Elmer supports parallelism across multiple nodes using MPI, but uses only a single thread inside every node. For this reason, we have also implemented a multithreaded CSR version for Elmer, in order to achieve a fair comparison with CSX.

The integration process was quite straightforward, since Elmer already supported the use of external SpMV libraries through a well-defined multiplication interface; we did, however, extend the interface slightly, in order to support the notion of matrix representation tuning (see Figure 5.16). Elmer passes to the external library the CSR data structures and the input and output vectors along with a generic handle to be filled with the tuned matrix, either automatically (e.g., first call) or upon request (`reinit` flag). When a matrix-vector multiplication call is redirected to the CSX library, if the `*tuned` handle is valid or the `reinit` flag is set, we construct the CSX matrix, assign it to the `*tuned` handle and proceed with the multiplication. Otherwise, we simply perform the multiplication with the provided handle, ignoring all CSR parameters. The process of constructing the CSX matrix from a CSR input consists of two steps:

(a) *Conversion of CSR into CSX's internal representation.* This representation

```
void matvec_(void **tuned, void *n,
             void *rowptr, void *colind, void *values,
             void *u, void *v,
             int reinit);
```

Figure 5.16: The updated matrix-vector multiplication interface of Elmer. New arguments are typeset in italics, while u and v are the input and output vectors, respectively. If `*tuned` is `NULL` or `reinit` is set, the CSX matrix is constructed from the CSR parameters and assigned to the `*tuned` handle. If the `reinit` flag is not set during a computation, the CSX matrix will be constructed once upon the first call.

Problem name	**Equations involved**	**SpMV exec. time (%)**
fluxsolver	Heat + Flux	57.4
HeatControl	Heat	57.5
PoissonDG	Poisson + Discontinuous Galerkin	62.0
shell	Reissner-Mindlin	83.0
vortex3d	Navier-Stokes + Vorticity	92.3

Table 5.5: The benchmark problems used for the evaluation of the CSX integration into the Elmer multiphysics software. The problems are selected from Elmer's testing suite and significantly enlarged, in order to lead to quite large matrix representations (larger than 576 MiB). The execution time percentages are reported over the total solver's time, including a diagonal preconditioner.

is used for the preprocessing of matrix (see Section 5.3.1).

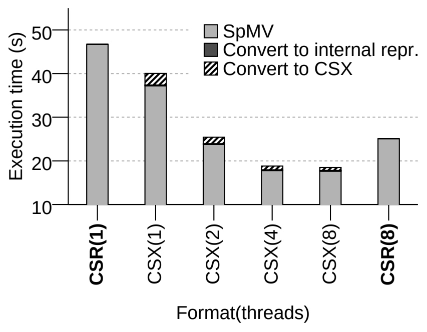
(b) *Detection and encoding of substructures.* This is the phase of the actual CSX preprocessing, where the matrix is mined for substructures and the final CSX format is constructed.

For the evaluation of the CSX integration, we have selected five benchmark problems from the Elmer testing suite. We have properly augmented the problem size (increasing the grid density), so that the resulting matrices are quite large, exceeding the 576 MiB boundary[8]. Table 5.5 presents details on the selected benchmarks; the percent of the total solver's execution time[9] spent in the SpMV routine is also reported.

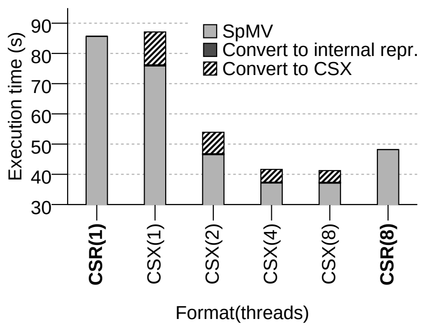
Figure 5.17 shows an execution time breakdown of the total time Elmer spent inside the CSX library after 1000 Bi-CG iterations. The first observation

[8] This is the size of the aggregate cache of the cluster nodes participating in the computation.

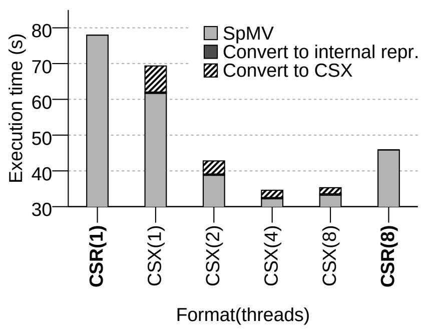
[9] Assembly times are not included.



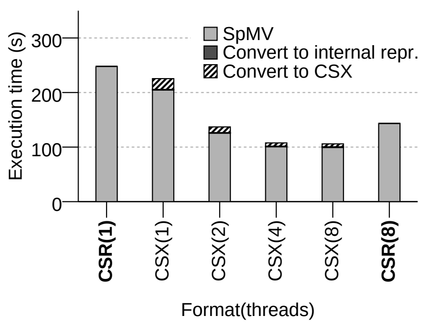
(a) HeatControl.



(b) PoissonDG.



(c) shell.

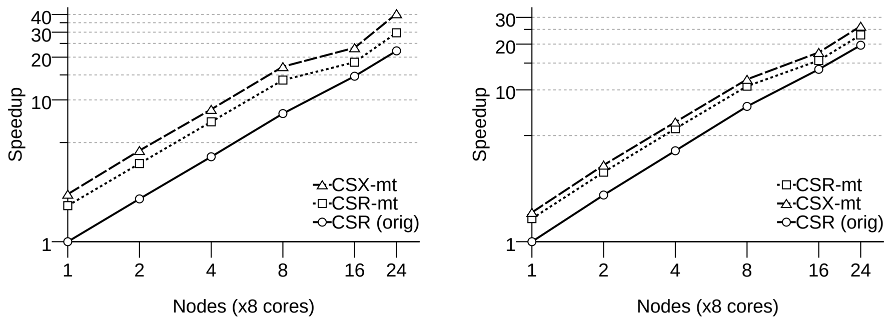


(d) vortex3d.

Figure 5.17: Execution time breakdown of the CSX library after 1000 Bi-CG iterations. The conversion cost is negligible, while CSX preprocessing is easily hidden by increasing the computation threads. CSX's bootstrap cost is completely amortized after less than 300 iterations.

is that the cost of the conversion to the CSX's internal representation is negligible. In fact, this step is a single large allocation of the internal representation's data structures and a sweep of the input CSR matrix. The rest of the preprocessing cost is hidden as we add more threads to the SpMV computation or if the computation itself is quite significant, e.g., in the vortex3d benchmark. The total preprocessing cost of CSX is completely amortized after 224–300 Bi-CG iterations with the multithreaded CSR kernel. While this cost is reasonable for large and complex simulations that may take thousands of iterations, CSX's preprocessing could also be performed offline, in order to further minimize this bootstrap cost.

Despite the preprocessing cost, however, CSX was able to considerably accelerate not only the SpMV component of the Elmer software, but also the over-



(a) SpMV component. (b) Overall solver.

Figure 5.18: Speedup of the Elmer multiphysics simulation software by employing CSX (preprocessing time is included) after 1000 linear system iterations. The knee in the multithreaded speedup diagrams at the 16 nodes is due to the lower memory bandwidth of the Clovertown cluster nodes, which start to participate in the computation from this point on.

all execution time of the solver (Figure 5.18). More specifically, CSX was able to provide a 37% average improvement of the SpMV component in a 24-node, two-way SMP, quad core Intel Xeon E5405/E5335 (Harpertown/Clovertown) mixed cluster after 1000 Bi-CG iterations. The overall solver's performance was improved by a noticeable 14.8%, despite the large preconditioning cost in three of the benchmarks used (see Table 5.5). We believe this improvement will be much more significant, if other parts of the solver (e.g., the preconditioner) also exploit parallelism.

5.9 Summary

In this chapter, we presented, discussed and evaluated in detail the Compressed Sparse eXtended (CSX) storage format, a new approach for storing sparse matrices, in order to achieve high performance in modern multicore architectures. CSX is able to detect a variety of substructures in the non-zero element structure of the matrix, e.g., horizontal, diagonal, two-dimensional blocks, which then encodes them in a single, very condensed, integrated matrix representation. This characteristic allows CSX to adapt to the specificities of every input matrix and achieve considerable compression levels, providing stable high performance from irregular to special-structured matrices. CSX takes special care for NUMA architectures by favoring the computational part of the kernel, which is more significant in these cases. In practice, CSX overwhelms in terms of absolute performance all other CSR alternative storage formats, providing performance gains more than 60% and 20% in SMP and NUMA architectures, respectively. Moreover, it is rather important that these improvements do no not come at an excessive preprocessing cost, allowing CSX to provide noticeable performance improvements also in the context of a production multiphysics simulation software, despite its initial bootstrap cost.

6

Exploiting symmetry in sparse matrices

Symmetric sparse matrices arise quite often in the solution of sparse linear systems and specific iterative methods exist for treating such symmetric problems. Since the key performance problem of the SpMV kernel is the saturation of the system's memory bandwidth, it is quite tempting to reduce almost to the half the representation size of a symmetric matrix by do not storing the non-zero elements of the upper or lower triangular part. While beneficial in a single-threaded context, such an approach proves to be problematic in multithreaded configurations. The typical row-wise distribution of the sparse matrix among the threads introduces an undesirable race condition on the elements of the output vector. The preferred way to solve this dependency, in order to avoid the prohibitive cost of locking, is to use per-thread local vectors for performing the local SpMV computation; this method, however, introduces an additional reduction step in the symmetric SpMV computation, which further stresses the memory subsystem and eliminates the benefit from the reduction in the matrix size.

In this chapter, we propose an indexing scheme for the local vectors of the symmetric SpMV kernel, which minimizes the memory traffic during the reduction phase of the kernel, allowing symmetric SpMV to scale. We also extend the CSX storage format to support efficiently symmetric matrices. In conjunction with the local vectors indexing scheme, symmetric CSX is capable of providing a more than 2$\times$ performance improvement over the standard CSR format and accelerating the CG iterative method by nearly 80%.

6.1 The symmetric SpMV kernel

The most widely used format for storing symmetric sparse matrices is the Symmetric Sparse Skyline (SSS) format, a variation of CSR that stores only the main diagonal and the lower triangular elements. More specifically, SSS stores the diagonal elements separately and the lower triangular elements using the typical CSR format (see Chapter 2, Section 2.4 for more details). Assuming four-byte

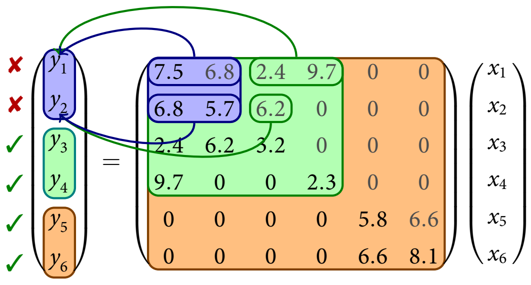


Figure 6.1: The RAW dependency on the output vector in a parallel execution of the symmetric SpMV kernel (figure replicated from Chapter 2).

indices and eight-byte double-precision floating point values for the non-zero elements, the matrix size in the SSS format can be calculated as follows:

$$S_{SSS} = \underbrace{4\frac{NNZ-N}{2}}_{\text{colind}} + \underbrace{8\frac{NNZ-N}{2}}_{\text{lower values}} + \underbrace{4(N+1)}_{\text{rowptr}} + \underbrace{8N}_{\text{diagonal}} \tag{6.1}$$

$$= 6(NNZ+N)+4 \tag{6.2}$$

For the typical case of sparse matrices, where $NNZ \gg N$, the matrix size in the SSS format is almost half that of CSR's (approximately $12NNZ$). Given the streaming nature of the SpMV kernel, this considerable reduction in the matrix size is expected to provide a significant performance improvement in a single threaded implementation of the kernel. In a multithreaded context, however, the execution of the kernel becomes problematic. Splitting the matrix row-wise introduces a RAW dependency in specific output vector elements due to the vector updates generated by the SpMV operations in the upper triangular matrix (see Figure 6.1). The typical way of avoiding this dependency without the use of locking, whose cost can be prohibitive, is to use local, per-thread vectors for storing the partial results of the multiplication (Algorithm 6.1). The SpMV execution is split into two parallel phases: the *multiplication* phase and the *reduction* phase. The multiplication phase (Algorithm 6.1, lines 2–11) performs the actual SpMV operation writing the partial results to the local vectors, while the reduction phase (Algorithm 6.1, lines 12–16) reduces the local vectors into the final output vector.

The reduction operation required by the local vectors method is streaming and is expected to be bound from the available memory bandwidth, despite its high parallelism at the order of $\Theta(N)$. This phase introduces a significant workload overhead to the SpMV computation that depends on the total number of threads used. Assuming eight-byte double precision values for the local

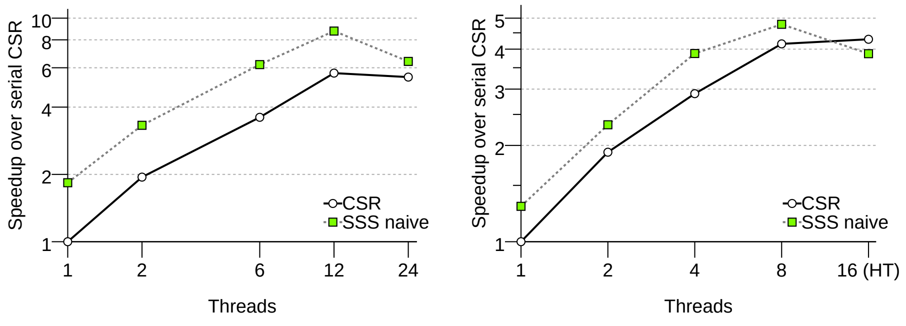
```
1: procedure MatVecSssMt($A$::in, $x$::in, $y$::out, $p$::in, $start$::in, $end$::in)
   $A$: matrix in SSS format
   $x$: input vector
   $y$: output vector
   $p$: participating threads
   $start, end$: start/end of thread partitions
2:     for $i \leftarrow 0$ to $p$ do in parallel
3:         for $r \leftarrow start[i]$ to $end[i]$ do
4:             $y_i[r] \leftarrow dvalues[r] \cdot x[r]$
5:             for $j \leftarrow rowptr[i]$ to $rowptr[i+1]$ do
6:                 $c \leftarrow colind[j]$
7:                 $y_i[r] \leftarrow y_i[r] + values[j] \cdot x[c]$
8:                 $y_i[c] \leftarrow y_i[c] + values[j] \cdot x[r]$
9:             end for
10:        end for
11:    end for
12:    for $r \leftarrow 0$ to $N$ do in parallel
13:        for $i \leftarrow 0$ to $p$ do
14:            $y[r] \leftarrow y[r] + y_i[r]$
15:        end for
16:    end for
```

Algorithm 6.1: Parallel implementation of the symmetric SpMV kernel using the SSS storage format.

vectors and p participating threads, the working set of the reduction phase is

$$ws_{\text{red}} = 8pN \tag{6.3}$$

The key characteristic of this overhead is that it increases proportionally to the number of threads used and is expected to lead to a saturation of the available memory bandwidth beyond a certain number of threads whatsoever, limiting the scalability of the symmetric SpMV kernel. This behavior is depicted clearly in the speedup diagrams of Figure 6.2. While the performance benefit of the symmetric SSS kernel ranges between 30% and 80% in the single-threaded configuration, this gain is continuously shrinking as the number of threads increases until it is completely eliminated (Dunnington, 24 threads, Gainestown, 8 threads) or even reversed (Gainestown, 16 threads), leading to considerably lower performance than the baseline CSR.



(a) Dunnington. (b) Gainestown.

Figure 6.2: Speedup of the naive symmetric SpMV implementation. The scalability of the symmetric SpMV kernel using local vectors is limited by the final reduction step, whose workload increases linearly to the number of computation threads, leading to a complete saturation of the available memory bandwidth resources.

6.2 Minimizing the reduction cost

The previous discussion on the symmetric SpMV kernel has revealed the key performance problem of the increased workload overhead of the reduction phase. Due to its streaming nature, it is crucial to reduce the data computations during the reduction phase, in order to alleviate the memory subsystem and, if possible, decouple this overhead from the thread count, allowing SpMV to scale. We present here two approaches toward this direction based on the key observation that only specific parts of the output vector need to be kept locally, since not all accesses conflict. The memory traffic incurred by the reduction phase can be therefore considerably reduced.

6.2.1 Effective ranges of local vectors

The method of effective ranges, proposed by Batista et al. [2010], updates only a specific region of each local vector and redirects the rest of updates directly to the output vector. According to the symmetric SpMV algorithm (Algorithm 6.1), the i-th thread is assigned the calculations for the SSS submatrix between the $start[i]$ and $end[i]$ rows. Since we store the lower triangular matrix, there is no way for the thread i to access elements below the $end[i]$ row boundary. Therefore, it is safe to exclude this region from the reduction phase. Furthermore, thread i can obtain access to the output vector between the $start[i]$ and $end[i]$ elements directly without ruining SpMV consistency, exactly as in the regular unsymmetric SpMV implementation. The SpMV operations at the remaining region, from the first row up to $start[i]$, may conflict with the output vector updates and, therefore, it must write to the local vector; this region is the *effective region* of the local vector (Figure 6.3c) and should be updated during the reduction phase. Assuming, without loss of generality, that each thread obtains almost the same number of rows, the working set overhead of the reduction phase for this method using p threads can be calculated as follows:

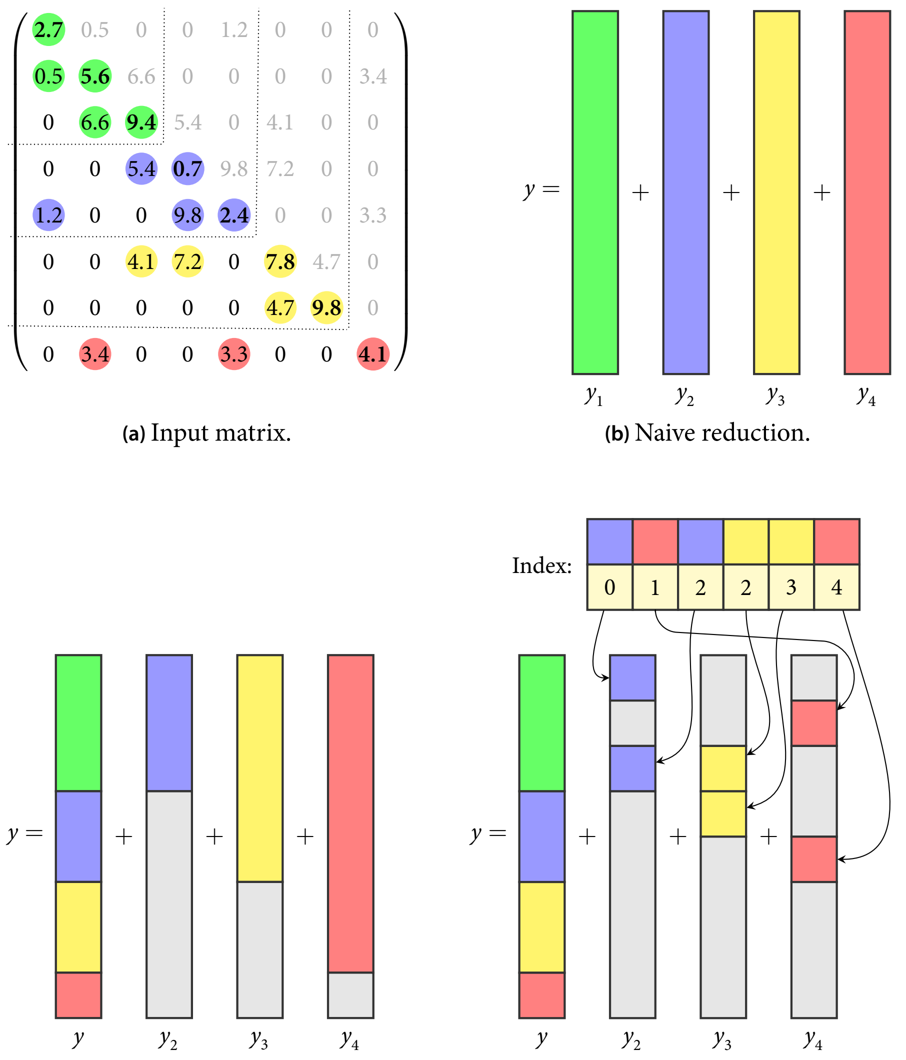
$$ws_{\text{eff}} \approx 8\frac{p(p-1)}{2} \cdot \frac{N}{p} = 4(p-1)N \qquad (6.4)$$

The reduction overhead is now halved compared to the naive version, but the key problem of the symmetric SpMV execution remains: the overhead of the reduction phase still grows linearly with the number of participating threads.

Geus and Röllin [2001] present a similar approach for non-banded matrices for minimizing the communication data among the processes of a distributed SpMV computation. They specialize their approach for banded matrices by reducing the effective range of the communicated data (i.e., local vectors) to the half of matrix bandwidth. This approach can be extended straightforwardly to regular matrices and collapses to the method described here for matrices with a very large bandwidth. Nonetheless, even this approach does not decouple the reduction overhead from the thread count, especially for matrices with a relatively large bandwidth.

6.2.2 Local vectors indexing

Our approach on minimizing the reduction phase overhead is based on the observation that the effective regions of the local vectors are quite sparse, i.e., very few elements of the effective regions are actually updated during the SpMV computation. Another interesting trait of the effective regions, as depicted in Figure 6.4, is that their density is continuously decreasing as more threads are added to the SpMV computation, reaching an average of 10.7% at the 24 threads of Dunnington and 2.6% at 256 threads. Motivated by this behavior, we introduce an indexing scheme for the non-zero elements of the local vectors, in order to update only the conflicting elements during the reduction phase, minimizing the workload overhead. For each non-zero element in the effective regions of the local vectors (Figure 6.3d), we keep a pair (*vid, idx*), where *vid* is a unique local vector ID and *idx* is the index of the non-zero element inside the local vector. The size of *idx* equals the matrix index size, i.e., four-bytes, while *vid* can vary depending on the maximum expected thread count. In our implementation, we use generously four bytes for the *vid* field, but two or even a single byte is enough for current multicore architectures.



(a) Input matrix.

(b) Naive reduction.

(c) Reduction with effective ranges.

(d) Indexed reduction.

Figure 6.3: Local vector methods for the reduction phase of the symmetric SpMV kernel. An example implementation of symmetric SpMV with $p = 4$ threads. The naive method (6.3b) uses simply four local buffers that reduces later to the final output vector. The effective ranges method (6.3c) uses $p - 1$ local vectors writing only the possibly conflicting regions. The indexing scheme proposed (6.3d) uses $p - 1$ local vectors and an indexing structure that points only to the really conflicting elements.

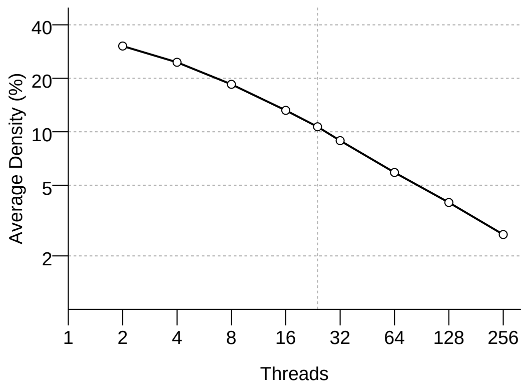


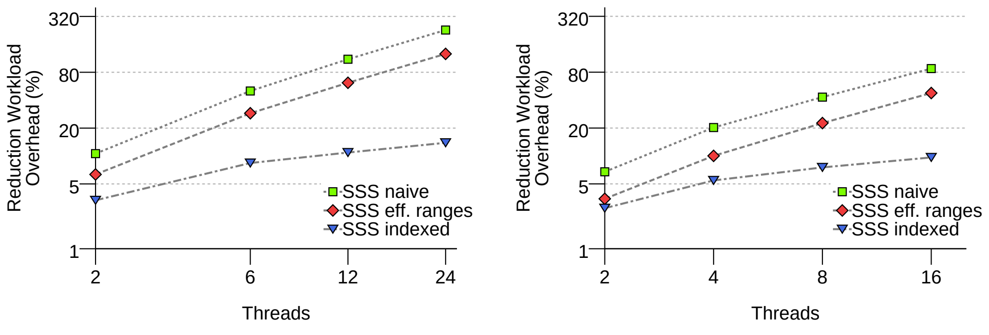
Figure 6.4: The density of the effective regions of local vectors. Local vectors become more sparse as the thread count increases, reaching a 2.7% density at 256 threads. The vertical line marks the density at the 24 threads of Dunnington.

The workload overhead of the reduction phase using our indexing scheme is now dependent on the density d of the effective regions of the local vectors. More specifically, assuming, without loss of generality, that the N matrix rows are equally partitioned among p threads, the working set overhead is

$$ws_{\text{idx}} = ws_{\text{eff}}d + \underbrace{8\frac{(p-1)Nd}{2}}_{\text{index size}} \tag{6.5}$$

$$\approx 8(p-1)Nd \tag{6.6}$$

Although, theoretically, the local vectors indexing does not decouple the reduction phase overhead from the thread count, in practice, the effect of the thread count increase is considerably attenuated. This is due to the inverse relation between the thread count and the density of the effective regions, which eventually tends to stabilize the workload overhead as the thread count increases and is visually depicted in Figure 6.5. The reduction overhead of both the naive and the effective ranges methods increases linearly with the thread count, exceeding significantly the multiplication phase at the 24 threads in Dunnington. The overhead of the indexing scheme proposed, on the other hand, increases at a much slower pace and tends to stabilize around 15% at the 24 threads. Indeed, with an average density of almost 11% at this thread count, the indexing scheme's working set overhead is more than four times lower than the case of the effective ranges. In Gainestown, the reduction overhead is lower for all methods, since the memory bandwidth contention is not very pronounced, but the same behavior is observed also in this architecture.



(a) Dunnington. **(b)** Gainestown.

Figure 6.5: The workload overhead of the reduction phase (over the serial SSS implementation) for the three local vectors methods considered. The reduction overhead increases linearly with the thread count for the naive and the effective ranges methods, surpassing the execution time of the actual multiplication. Conversely, the overhead of the indexing scheme proposed tends to stabilize as the thread count increases, leading to a rather small, 15% overhead.

Parallelization The parallelization of the reduction phase in our index-based scheme is based on the local vectors index, since this specifies the actual reduction operations. More specifically, we first sort the index in an ascending order of the *idx* field and then split it equally among the participating threads. The only restriction in the splitting process is that an *idx* value must not be shared among any pair of threads, in order to guarantee the independence of the updates to the final output vector.

6.2.3 Alternative methods

The reduction phase of the symmetric SpMV cannot be avoided, unless with the use of atomic operations on the output vector elements. Buluç et al. [2011] take an interesting approach by introducing a hybrid method that uses a constant number of reduction operations and some strain atomic operations on specific output vector elements. The key idea of the method resides on organizing the matrix into large block diagonals (three in the proposed implementation), stored with the efficient CSB blocking storage format (see Chapter 2, Section 2.2.3). The SpMV computation proceeds independently between the three block diagonals[1] and writes the results to dedicated local vectors, which

[1] The computation inside each diagonal is also parallel proceeding with two distinct phases.

will be reduced to the output at a final step. The SpMV computations on non-zero elements beyond the block diagonals proceed independently and write directly to the output vector using atomic operations. The key advantage of this method is that it uses a constant number of local vectors, decoupling the reduction overhead from the thread count. However, the overhead of reducing even three local vectors for very large and sparse matrices, where the local vectors might not fit in the system's aggregate cache, can be quite important. Additionally, the cost of atomic operations for strain non-zero elements can be quite important for matrices with a large bandwidth.

6.3 CSX for symmetric matrices

Having optimized the reduction phase of the symmetric SpMV kernel, we take a step further and optimize the multiplication phase by extending the highly compressed CSX format (see Chapter 5) to support symmetric matrices. This optimization is orthogonal to the reduction phase optimization discussed in the previous section. Similarly to SSS, the symmetric version of CSX, henceforth called CSX-Sym, stores separately the main diagonal in a `dvalues` array and the lower triangular submatrix in normal CSX. All the substructures detected in the lower submatrix have a symmetric counterpart in the upper triangular submatrix; for example, every horizontal substructure in the lower half starting at (i, j) element denotes also the existence of a vertical one in the upper half starting at position (j, i) and so forth. The generated code for CSX-Sym takes care for computing also the symmetric substructure. The only restriction we impose on the detection of symmetric substructures is that all of its vector updates must be directed either to the original output vector or the local vector; not to both of them. In other words, if a the i-th thread's partition starts at $start[i]$ row, symmetric substructures crossing this boundary are not encoded. Figure 6.6 shows an example of the substructures detected by CSX-Sym; note that the horizontal substructure starting at row 6 is not encoded, since the elements of its symmetric counterpart cross the original/local vector boundary. Despite deteriorating slightly the compression potential of CSX-Sym, this restriction leads to more efficient code, since per-element checks would be needed otherwise, in order to determine where to redirect the vector update.

Compression potential Employing all the sophistication of CSX in detecting and encoding substructures inside a sparse matrix, CSX-Sym is able to considerably compress the symmetric sparse matrix representation, reaching very close to the maximum possible compression ratio (67%), while the SSS format

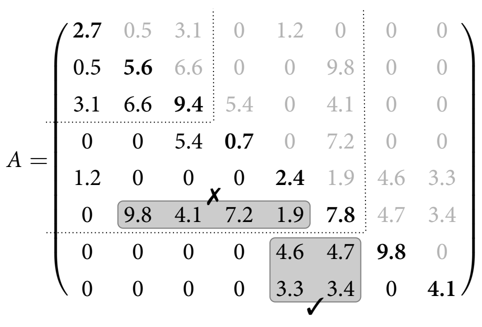


Figure 6.6: Encoded substructures by CSX-Sym in an 8 × 8 sparse matrix. Dotted lines denote thread partitions; the shaded elements of the upper triangular submatrix are not stored. Note that the horizontal substructure at row 6 is not encoded, since its symmetric vertical substructure crosses the thread partition boundary.

Matrix	**CSR Size (MiB)**	**C.R. (SSS)**	**C.R. (CSX-Sym)**	**C.R. (Max.)**
parabolic_fem	44.06	45.4%	49.6%	63.6%
offshore	49.54	48.0%	56.1%	65.3%
consph	69.10	49.5%	63.9%	66.4%
bmw7st_1	84.54	49.3%	64.4%	66.2%
G3_circuit	93.72	43.5%	60.2%	62.4%
thermal2	102.88	45.4%	53.4%	63.6%
m_t1	111.99	49.7%	65.3%	66.4%
bmwcra_1	122.38	49.5%	65.1%	66.4%
hood	124.08	49.3%	64.4%	66.2%
crankseg_2	162.16	49.8%	64.9%	66.6%
nd12k	162.88	49.9%	64.9%	66.6%
af_5_k101	202.77	49.1%	63.9%	66.0%
inline_1	423.25	49.5%	64.7%	66.4%
ldoor	536.04	49.3%	64.5%	66.2%
boneS10	638.28	49.5%	65.1%	66.3%

Table 6.1: Compression ratio (C.R.) of the symmetric CSX. Symmetric CSX is able to provide near optimal compression ratios for symmetric matrices.

usually contends itself to less than 50% in most of the cases. Table 6.1 details the compression ratio of both formats for the 15 symmetric matrices of our suite.

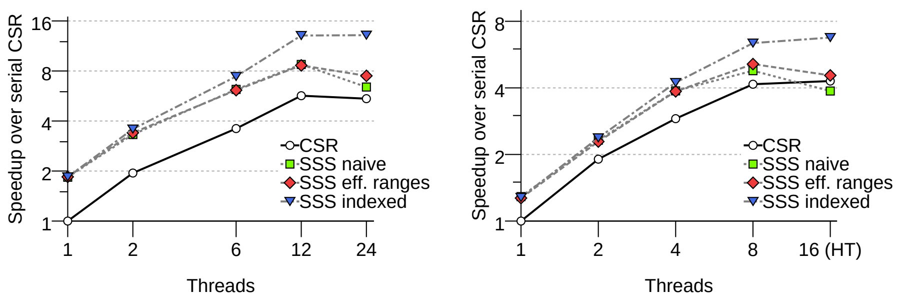
6.4 Performance evaluation

In the following, we evaluate the performance of the three local vector methods discussed, namely the naive, the effective ranges and the index-based local vector methods. We also evaluate the impact of the CSX-Sym variant in the execution of the symmetric SpMV kernel. For the evaluation, we use the 15 symmetric and positive definite sparse matrices of our matrix suite and focus on the Dunnington and Gainestown platforms as representatives of the SMP and NUMA architectures, respectively. We employ a 'share-nothing' core-filling policy for this evaluation, in order to exhibit more clearly the memory bandwidth contention of the reduction phase of the symmetric SpMV kernel. The CSX-Sym variant is setup exactly as the typical CSX (see Chapter 5, Section 5.7). In order to exhibit its full potential, we report performance results with full preprocessing enabled for the comparisons with other symmetric SpMV approaches. In the evaluation of the CG iterative method, however, we employ statistical sampling as described in Chapter 5, Section 5.7.3, and include also this time in the overall reported results. For more information on the matrix suite, the hardware platforms used and the experimental procedures, the reader is referred to Chapter 3, Section 3.2.

6.4.1 Local vectors methods

The performance improvement achieved by the different local vector methods considered is depicted in the speedup diagrams of Figure 6.7. All methods start with a significant improvement in the single-threaded configuration, especially in the Dunnington platform, but the naive and the effective ranges methods scale at a lower rate compared to the baseline CSR. The performance improvement offered by this methods is continuously shrinking as the thread count increases and is completely eliminated when the memory bandwidth is saturated. Due to its reduced working set overhead, the effective ranges method exhibits a slightly better behavior in such cases, but it still does not allow symmetric SpMV to scale. In Gainestown, the performance of the naive and the effective ranges methods is very close to the original CSR at the eight-threaded configuration and deteriorates significantly at the 16-threaded.

The performance benefit of the proposed local vectors indexing method is prominent in both considered architectures. The considerably reduced working set overhead, which remains almost stable with the thread count, allows symmetric SpMV to scale at the same rate as the original CSR implementation, without compromising the performance improvement in the cases of memory bandwidth saturation. More specifically, our indexing scheme achieves an 83.9% performance gain over the best SSS configuration in Dunnington (12



(a) Dunnington. (b) Gainestown.

Figure 6.7: Symmetric SpMV speedup with different local vectors reduction methods. The indexing method is the most efficient allowing SpMV to scale.

threads) and a 44% improvement in Gainestown. Overall, the symmetric SSS kernel using the local vectors indexing scheme was able to provide a more than 2× improvement over the standard CSR implementation in Dunnington and 1.5× in Gainestown in the multithreaded configurations. This is a rather significant achievement, since the proposed technique allows the efficient exploitation of the symmetric structure of certain matrices, which otherwise would remain unexploited.

In order to provide more insight on the importance of the reduction phase in the symmetric SpMV implementation, we present in Figure 6.8 the execution time breakdown of the symmetric SpMV kernel for all the considered reduction methods at the 24-threaded configuration in Dunnington. It is clear that the proposed local vectors indexing scheme reduces considerably the reduction phase overhead, keeping it at a minimal level. This reduction method has a beneficial side-effect: the multiplication phase has also decreased with the proposed indexing scheme. This is mainly due to the lower cache interference introduced by the modest working set overhead of our method. The high working set overhead of the alternative methods in many-threaded configurations is likely to spill out useful data from the cache, incurring an increased overhead to the multiplication phase of the next iteration.

Finally, it is essential to point the four cases (parabolic_fem, offshore, G3_circuit, thermal2) that CSR's performance surpasses or reaches our indexing method. These matrices are high-bandwidth matrices with a lot of their non-zero elements at very long distances from the main diagonal, leading to considerable memory traffic during the reduction phase. However, the local vectors indexing method seems to handle efficiently even such cases, exhibiting a rather low overhead, while the naive and effective ranges methods are overwhelmed by the reduction cost.

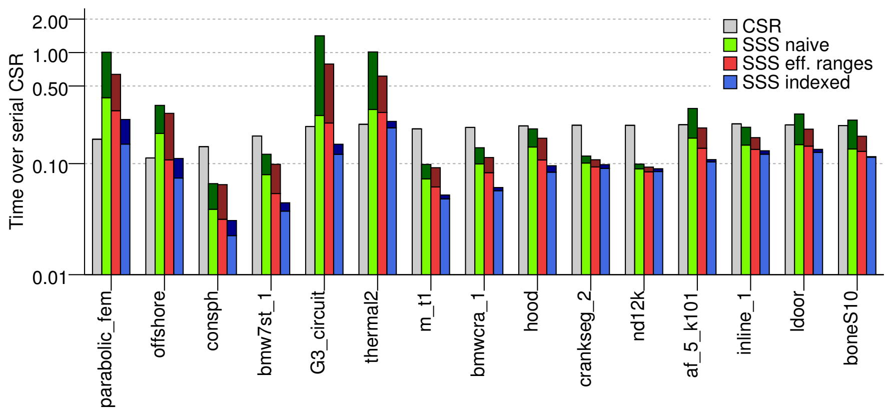
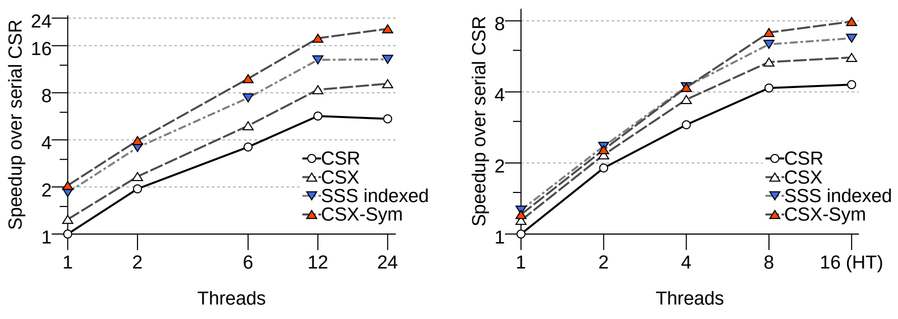


Figure 6.8: Symmetric SpMV execution time breakdown at 24 threads in Dunnington. The reduction overhead (darker regions) is considerably reduced with the use of local vector indexing.

6.4.2 Symmetric CSX

The performance of the CSX-Sym variant and the optimized SSS format (local vectors indexing) compared to the unsymmetric CSR and CSX implementations are depicted in Figure 6.9. Thanks to its highly compressed representation, CSX-Sym provides a 43.4% further performance improvement over the optimized SSS format in Dunnington, an architecture that is mostly affected by the matrix size representation. In Gainestown, where the available memory bandwidth is ample, the performance gap closes to 10% on average, but CSX-Sym is still able to provide a performance gain. The unsymmetric CSX and CSR implementations are well below in performance, especially in Dunnington, where the memory bottleneck is more prominent. CSX-Sym was able to provide an impressive (for an SpMV implementation) 21× speedup in Dunnington at the 24 threads and a nearly 8× in Gainestown at the 16-threaded configuration. This noteworthy behavior is due to the low overhead of the index-based reduction phase, which allows the benefit from the advanced compression of CSX to become visible also in the symmetric kernel.

In order to gain a further insight in the CSX-Sym performance, Figure 6.10 shows the per-matrix performance of the four considered formats (CSR, SSS, CSX and CSX-Sym) at the 16-threaded configuration in Gainestown. CSX-Sym manages to achieve the best performance, surpassing 10 Gflop/s, in 11 out of the 15 matrices. The remaining four matrices are the high-bandwidth corner cases, where no symmetric format did achieve any performance improvement over CSR. High-bandwidth matrices have their non-zeros across the whole matrix, leading to a rather low substructure frequency. However, with the ex-



(a) Dunnington. (b) Gainestown.

Figure 6.9: Symmetric SpMV speedup with the CSX-Sym format. CSX-Sym optimizes further the symmetric SpMV kernel, especially in SMP architectures, where the memory bandwidth contention is more prominent.

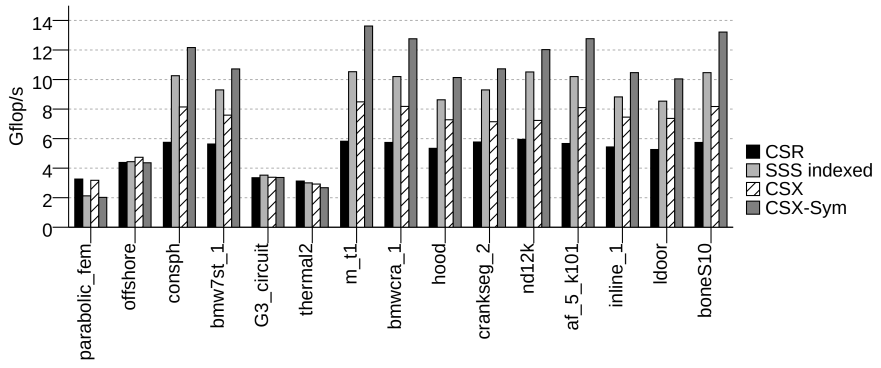


Figure 6.10: Per-matrix performance for the CSX-Sym format at 16 threads in Gainestown. CSX-Sym's performance in more regular matrices surpasses 10 Gflop/s, while staying close to baseline CSR performance in less regular ones.

ception of parabolic_fem, which has a rather irregular structure and very high bandwidth, CSX-Sym was able to achieve near-best performance for almost all these corner-case matrices.

6.4.3 Reduced bandwidth matrices

The execution of the symmetric SpMV kernel is particularly affected by the bandwidth of the input sparse matrix. Apart from the performance problems caused to the typical SpMV kernel, e.g., irregular access in the input vector,

	Dunnington (24 threads)	**Gainestown (16 threads)**
CSR	24.8%	8.9%
CSX	50.2%	11.1%
SSS	73.6%	34.9%
CSX-Sym	85.3%	38.6%

Table 6.2: Symmetric SpMV performance improvement due to matrix reordering (the RCM algorithm was used). The impact of reordering is much more significant for symmetric formats, while CSX tends to be benefited more from this optimization, since it is able to detect and encode more substructures.

increased loop overheads, load imbalances etc. (see Chapter 3, Section 3.3 for more details), high-bandwidth matrices tend to increase the interference among the threads of the symmetric SpMV kernel when writing to the output vector. The reduction phase of the naive and the effective ranges local vectors methods is not affected by this interference, since all updates are directed to the local vectors. The proposed local vectors indexing scheme, however, is affected, since the reduction phase overhead depends exactly on this interference, as we update in local vectors only the conflicting elements. Reducing, therefore, the bandwidth of such matrices can be quite beneficial for the symmetric SpMV kernel.

Originally conceived for reducing the communication overhead in distributed symmetric SpMV implementations, matrix bandwidth reduction techniques try to bring the non-zero elements as close to the main diagonal as possible, by reordering the rows and columns of the matrix. The obvious effect of this non-zero rearrangement in the SpMV execution is the minimization of the inter-process interference. In a distributed SpMV implementation, this is equivalent to reducing the communication overhead, since less data from the input and output vectors must be exchanged among the participating processors. In the context of a multithreaded execution, this is equivalent to the minimization of the reduction phase overhead. Matrix bandwidth minimization techniques have beneficial side-effects also in the typical unsymmetric SpMV implementation, since the homogenization of the non-zero elements distribution leads to a better access pattern and load balance (see Chapter 3). Substructure-detecting formats, like CSX, are benefited further, since the non-zero rearrangement increases the opportunities of detecting more substructures (see Chapter 5). This picture is verified also for the symmetric SpMV kernel, as Table 6.2 depicts, where the average performance improvement of the different SpMV implementation due to matrix reordering is reported. In Dunnington, the standard CSR gains a 22% improvement, while baseline CSX

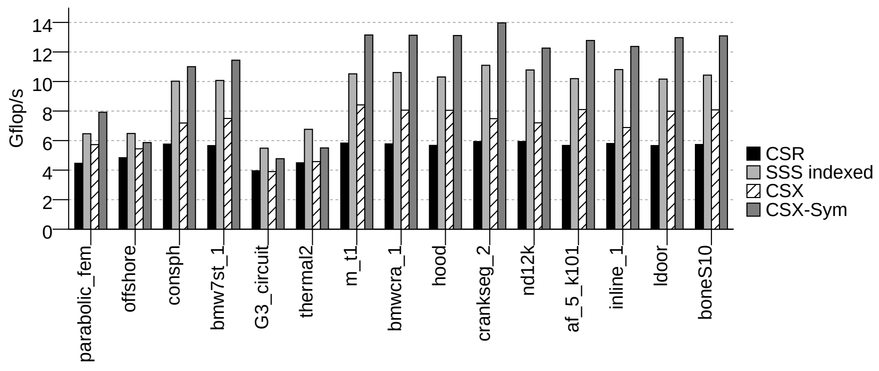


Figure 6.11: Per-matrix performance on reordered symmetric matrices (Gainestown, 16 threads). CSX-Sym is the most performant format surpassing the 12 Gflop/s barrier in the majority of sparse matrices.

is benefited by 63%. As expected, the effect of matrix reordering in the symmetric kernel is much more important, with the improvement of SSS surpassing 90%, while CSX-Sym gets a more than 2× acceleration. Similar is the picture in Gainestown, but both the encountered improvements and the differences among the different formats are attenuated. Such behavior—apparent throughout the performance evaluations in this thesis—is quite typical in NUMA architectures, where the memory bandwidth contention is not so intense as in SMP systems. Figure 6.11, finally, shows the absolute SpMV performance in the reordered matrices of our suite; it is noteworthy that CSX-Sym's performance surpasses 12 Gflop/s in nine matrices.

6.4.4 Impact on the CG iterative method

In order to evaluate the impact of the local vectors indexing optimization and the CSX-Sym format in the context of an iterative solver, we have implemented a non-preconditioned version of the Conjugate Gradient (CG) method. The CG method is a widely used iterative solution method for symmetric positive definite linear systems. It is also part of the NAS parallel benchmark suite, as a typical kernel of unstructured grid computations, for assessing the performance of a parallel system in irregular long distance communication and the matrix-vector product [Bailey et al., 1991].

Figure 6.12 shows the execution time breakdown of the CG method using the unsymmetric CSR and CSX formats and their symmetric counterparts, SSS and CSX-Sym (with local vectors indexing). Results are shown at the 24 threads in Dunnington for the RCM reordered matrices after 2048 iterations. The first generic observation is that the vector operations can be quite signif-

```
1: procedure ConjugateGradient($\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}_0$)
   $\mathbf{A}$: coefficient matrix
   $\mathbf{b}$: target vector
   $\mathbf{x}_0$: initial approximate solution
2:     $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
3:     $\mathbf{p}_0 \leftarrow \mathbf{r}_0$
4:     $i \leftarrow 0$                                    ▷ Iteration count
5:     loop
6:         $a_i \leftarrow \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}$                    ▷ SpMV operation
7:         $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + a_i \cdot \mathbf{p}_i$
8:         $\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - a_i \cdot \mathbf{p}_i$
9:         if $|\mathbf{r}_{i+1}|$ is adequately small then
10:            break
11:        $b_i \leftarrow \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}$
12:        $\mathbf{p}_{i+1} \leftarrow \mathbf{r}_{i+1} - b_i \cdot \mathbf{p}_i$
13:        $i \leftarrow i + 1$
       return $\mathbf{x}_{i+1}$
```

Algorithm 6.2: The non-preconditioned Conjugate Gradient algorithm.

icant in smaller and sparse matrices, such as parabolic_fem, offshore etc., and may exceed 50% of the overall execution time of the multithreaded CG kernel. CG performs several vector operations, including dot products, during an iteration (see Algorithm 6.2), but only a single SpMV operation. For small matrices, therefore, that fit in the aggregate cache, the overhead of vector operations can easily dominate the total execution time of the solver. With the exception of the very sparse parabolic_fem and offshore matrices, where the computation is dominated by the vector operations, CG is greatly benefited by the symmetric storage formats, encountering a more than 50% overall performance improvement in large matrices. CSX-Sym is hindered by its preprocessing cost in smaller matrices and offers similar or lower performance the SSS format with the local vectors indexing optimization technique. In larger matrices, however, CSX-Sym compensates its preprocessing cost and manages to offer a further performance improvement to the CG kernel.

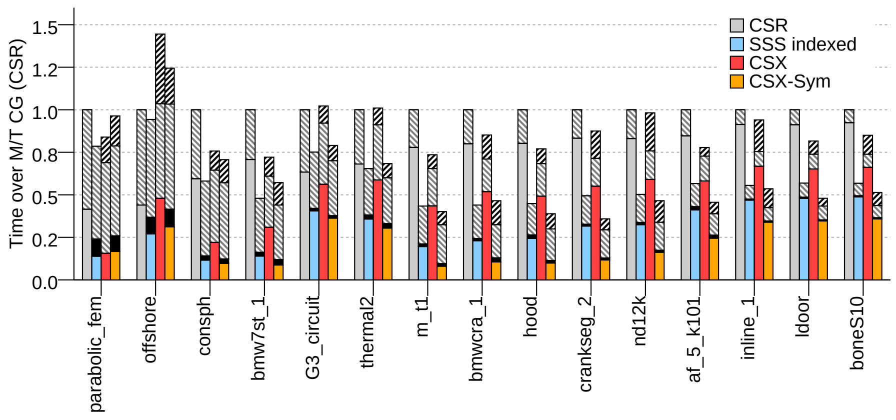


Figure 6.12: CG execution time breakdown using 24 threads in Dunnington at the RCM reordered matrices after 2048 iterations. Symmetric storage formats using the local vectors indexing technique provide a significant performance improvement; CSX-Sym is beneficial in large sparse matrices where it is able to compensate its preprocessing cost. *[Execution time breakdown legend – colored: SpMV, black: SpMV reduction, anti-diagonal: vector operations, diagonal: CSX preprocessing.]*

6.5 Summary

In this chapter, we focused on exploiting the non-zero element symmetry of symmetric sparse matrices, in order to accelerate the performance of the SpMV kernel. Storing almost half of the original matrix's non-zero elements (lower triangular and main diagonal), symmetric sparse matrix storage formats can significantly benefit the SpMV performance, due to the alleviation of the memory subsystem. However, the multithreaded symmetric SpMV kernel has to successfully overcome a RAW dependency in the output vector incurred by the SpMV operations performed in the symmetric (upper triangular) part of the matrix. The most common solution involves the use of local, per thread vectors, where the intermediate results of the SpMV computation are written. At a second step, these vector are reduced in parallel to the final output vector. The key performance problem of this step is that it incurs a significant performance overhead, increasing linearly with the thread count, which not allows SpMV kernel to scale. Based on the observation that local vectors are quite sparse and their sparsity increases with the thread count, we proposed a local vector indexing scheme, which reduces only the conflicting output vector elements, therefore attenuating the memory bandwidth bottleneck of the reduction phase. We further optimize symmetric SpMV by extending the CSX format to support also symmetric matrices. The so called CSX-Sym variant achieves almost optimal compression of the sparse matrix and, in conjunction with the local vectors indexing scheme, manages a more than $2\times$ acceleration of the symmetric SpMV kernel, while the impact of the proposed optimizations in the context of the CG iterative solver accounts for a more than 50% performance improvement for large symmetric sparse matrices.

7 Energy-efficiency considerations

Power-aware computing and energy-efficiency issues has been of particular interest in embedded computing for several years. As the level of integration of CMOS digital circuits heads toward its limits, energy-efficiency is gaining an increasing interest from the hardware up to the software level also in the HPC community. Leakage power is becoming increasingly important as circuit designs scale down to a few nanometers and tend to dominate the overall power dissipation of a chip. Techniques for dynamically switching off idle processor components are now meeting more traditional ones, like Dynamic Voltage and Frequency Scaling (DVFS) techniques, in an effort to minimize the overall processor's power dissipation. Nonetheless, application performance is a critical and definitive factor of HPC and should remain in the foreground. Balancing this tradeoff, therefore, is of key concern in implementing high-performance and energy-efficient systems and applications.

In this chapter, we take a first step toward identifying and exploring the performance-energy tradeoffs of the SpMV kernel in modern multicore architectures by varying the processor's frequency and the placement of threads at the available cores. Employing the notion of Pareto optimality, we characterize the different tradeoffs and propose a methodology for selecting the SpMV execution configurations (processor frequency, thread placement) that lead to the best performance-energy compromises for a specific input matrix. This chapter opens a number of issues that need to be answered more precisely in the future and serves as food-for-thought for evaluating the real impact of software optimizations targeting energy-efficiency in the context of HPC.

7.1 Fundamentals of processor power dissipation

7.1.1 Sources of power dissipation

The power dissipation of a CMOS device at the elementary transistor gate level is formulated as a sum of three major components: switching loss, leakage and short-circuit loss [Brooks et al., 2000; Kaxiras and Martonosi, 2008]. More specifically, the power dissipation P of a CMOS device is approximated by the following formula:

$$P = \frac{1}{2}CV^2\alpha f + P_{\text{leakage}} + P_{\text{short-circuit}} \tag{7.1}$$

The first term, also referred to as *dynamic power*, is the power dissipation due to the switching of the device transistors and is directly related to the circuit capacitance C, the supply voltage V, the clock frequency f of the device and the *activity factor* α. The circuit capacitance is a design parameter that depends largely on the wiring of the on-chip components. The activity factor is a fraction at the range $[0, 1]$, denoting how often the device is clocked, i.e., how often it is actually used; whenever idle, devices are deconnected from the clock source in order to save energy (*clock gating*). The *leakage power* is the power dissipation due to the current that leaks from a transistor when its gate is switched off. Although transistors are viewed as ideal on-off switches, in practice, a very small amount of current is flowing at the 'off' state, inducing a power loss. The *short-circuit* or *glitching power* is the power dissipated by the instantaneous short-circuit, due to imperfect state transition times, as a transistor switches between the on and off states.

Dynamic power has been the dominant source of power dissipation in CMOS devices for several years. Its quadratic relation to the supply voltage has led to considerable savings in power consumption as new technology allowed the safe reduction of this. The clock frequency has a double effect on the dynamic power dissipation: apart from its direct relation to the total dissipated power as specified by equation (7.1), sustaining a higher frequency often requires a proportionally higher supply voltage, leading to a cubic relation between frequency and dynamic power [Brooks et al., 2000; Kaxiras and Martonosi, 2008]. This significant impact of frequency on the total dissipated power has led to the wider adoption of a dynamic management of voltage and frequency, in order to reduce the overall power dissipation without compromising performance. *Dynamic Voltage and Frequency Scaling (DVFS)* techniques allow the selective scaling of both voltage and frequency in less CPU-intensive phases of an application, e.g., memory-bound or latency-tolerant parts, in order to achieve high performance at a lower power budget [Xie et al., 2003; Choi et al., 2004]. More advanced techniques separate the processor in multiple independent domains of voltage control, allowing the selective scaling of voltage in different processor components depending on the needs of the running application [Semeraro et al., 2002; Herbert and Marculescu, 2007; Mattson et al., 2010].

As the integration scale is shrinking to a few nanometers, leakage power is becoming a significant factor of the power dissipation of modern processors. There are two types of leakage power: *sub-threshold leakage* and *gate leakage*. Sub-threshold leakage is due to the electric current that flows through a transistor at the off state, i.e., when its voltage is below a threshold denoting the off state. Sub-threshold leakage increases exponentially as this threshold voltage is lowered [Kaxiras and Martonosi, 2008]. The reduction of the threshold voltage comes as a result of an analogous reduction of the supply voltage V in equation (7.1), since a minimum voltage difference is required between the on and off states of the transistor. Gate leakage, on the other hand, occurs due to direct tunelling of electrons through the insulator separating the gate and the transistor channel. As the insulator layer becomes too thin, due to the shrinking of the integration scale, the gate leakage increases exponentially and so does leakage power [Borkar, 1999; Kim et al., 2003; Agarwal et al., 2006].

It is clear that computer architecture is hitting a 'power wall', built chiefly by the tremendous increase in leakage power [Ramirez, 2011; Ahmed and Schuegraf, 2011], and drastic technology shifts are needed in order to keep Moore's law alive beyond the 22 nm barrier [Hisamoto et al., 2000; Yang et al., 2004; Skotnicki et al., 2005; Hu, 2012]. Modern microarchitectures tackle the leakage power problem by incorporating advanced power management. Apart from dynamic voltage and frequency scaling techniques, which have been around for some years [Semeraro et al., 2002], modern processors allow also the dynamic 'shut down' of idle cores. For example, IBM's latest eight-core four-way SMT Power7 processor can bring a single core into one of two 'sleep' modes, in order to save as much power as possible from idling cores [Kalla et al., 2010]. Similarly, Intel's Sandy Bridge microarchitecture alters dynamically the power state of different processor components, in order to optimize the total power dissipation, and also offers a power monitoring interface to the user level, so that software can take advantage and reduce the overall energy consumption [Rotem et al., 2012].

Closing the discussion on the sources of power dissipation in modern microarchitectures, we should make a specific note on caches. The integration of multiple fast cores into the same processor socket has placed a significant burden to the memory subsystem. Both in terms of latency and bandwidth, main memory cannot sustain the very high rates that a modern processor can consume data; and this difference tends to grow exponentially with every new processor generation [Wulf and McKee, 1995]. Therefore, large caches come to fill in and hide this performance gap, and it is not extraordinary to encounter today caches as large as 24 MiB per socket. However, such large caches take up a significant portion of the total die area and, as a result, are chief contributors to the processor's total leakage power[1] [Kaxiras and Martonosi, 2008]. Several techniques have been proposed for reducing the leakage power loss from caches, ranging from specific memory cell technologies [Powell et al., 2000] to techniques for shutting down whole unused portions of the cache [Yang et al., 2001] or specific unused cache lines [Kaxiras et al., 2001; Flautner et al., 2001]. Such techniques are soon becoming mainstream and allow energy-efficient designs for caches even exceeding 20 MiB [Chang et al., 2009; Huang et al., 2012].

7.1.2 Energy-Delay products

From a software perspective, the minimization of the power dissipation of the underlying architecture is just one side of the coin: performance is the other and is quite critical, as well. While there is a clear tradeoff between the power dissipation of a computer system and its performance, a high performance system is not necessarily inefficient in terms of energy consumption. Power dissipation is nothing more than the rate a computer system consumes energy. Therefore, the faster the computation, the lower is the total amount of energy consumed; after all, energy is what we pay for. For this reason, the $\frac{\text{flop/s}}{\text{W}}$ metric is being widely adopted for assessing the energy efficiency of a computer system. As of this writing, the fastest supercomputer in the world, achieving the mighty performance of 17.6 Pflop/s, is also the third[2] most energy-efficient with a performance of 2.1 Gflop/s per Watt of dissipated power [Top500, 2012; Green500, 2012].

A closer look at the performance per Watt metric reveals that it is nothing more than the inverse of the energy consumption. A common way for assessing the performance-energy tradeoffs of a system's design are the *energy-delay products* or $ED^n, n \geq 0$. For $n = 0$, the energy-delay product corresponds to the system's energy consumption, while larger values of n bias the tradeoff toward higher performance. As we shall see in Section 7.2, energy-delay products are optimal design tradeoffs under the Pareto optimality criterion, i.e., there exist no other design without compromising at least one of the energy or the performance objectives.

7.1.3 Energy-efficiency from a software perspective

Software applications can play a significant role in reducing the overall power dissipation of a computer system. The dynamic power of a processor is directly

[1] The dynamic power dissipation of very large caches is not so significant, since they usually reside in a different voltage and frequency domain than the rest of the processor, running at much lower frequencies.

[2] The most energy-efficient achieves 2.5 Gflop/s per Watt.

affected by the utilization of its units during the different phases of an application. In power terms, the utilization of a hardware unit can be viewed as an indication of its activity factor. Based on this assumption, Isci and Martonosi [2003] build a quite accurate model for predicting the power dissipation of processors based on the Intel Netburst microarchitecture [Koufaty and Marr, 2003]. The authors infer the activity factor of 22 hardware components (caches, TLBs, execution units etc.), by calculating their utilization using hardware performance counters, and use a weighted sum of the per-component power predictions, in order to calculate the overall dynamic power dissipation of the processor. A more detailed and generic approach in predicting the power dissipation of a modern processor is the McPAT power model framework [Sheng et al., 2009]. McPAT also predicts power dissipation through the use of performance monitoring events, but, conversely to the model of Isci and Martonosi [2003], it is not bound to a specific architecture technology. Instead, it uses low-level power models for modeling the behavior of fundamental components of a chip multiprocessor, allowing the exploration of design tradeoffs for new architectures.

Collecting a multitude of performance monitoring events, in order to obtain an accurate power estimate, might require multiple runs of the profiled application, since the number of performance monitoring registers is rather restricted in current microarchitectures. As a result, such methods are not suitable for online power estimation of a running application. Curtis-Maury et al. [2008] take a hybrid offline/online approach for predicting the execution configuration (thread count and core frequency) that provides an optimal balance between performance and energy consumption. The runtime system proposed collects performance monitoring information from a running application and feeds a regression model trained offline, which predicts the optimal configuration for the next phase of the application. In effect, this technique intends to allocate the minimum of resources (frequency, thread count) to an application, so that it proceeds at a low power budget without compromising performance. A similar hybrid approach is adopted also by Singh et al. [2009] in a power-aware thread scheduling scheme that tries to keep the overall system's power dissipation within a user-specified envelope.

While software control over dynamic power consumption is feasible, thanks to smart use of frequency scaling and thread placement, software control of leakage power is not yet widely adopted. In order to achieve this, the software must be able to 'switch off' specific functional units of the processor (ALU, FPU etc.) or deactivate unused parts of the cache, depending on its own needs. Assuming a hardware support for such fine-grained leakage power control, Zhang et al. [2003] and You et al. [2006] explore compiler techniques for identifying basic blocks where a specific functional unit is unused and instrument the resulting code with explicit activation/deactivation instructions for controlling the state of a functional unit. Similarly, Zhang et al. [2002] assume a fine-grained control of the instruction cache lines and propose compiler optimization techniques for selectively putting cache lines into a low leakage mode.

7.2 Performance-Energy tradeoffs in the SpMV kernel

Sparse Matrix-Vector Multiplication is a memory bandwidth bound kernel. This has been made clear from the analysis in the previous chapters, where we have proposed original techniques for attacking this key performance problem. We have shown that as far as the memory path is completely saturated, there is almost no worth at increasing the concurrency of the kernel, since the performance gain is minimized. This observation is important from a power perspective, since adding more cores to the computation increases the power dissipation for only a marginal gain in performance.

Thread placement plays also a significant role in the execution of the SpMV kernel. A sane core-filling policy can allow SpMV to scale, by delaying the saturation of the memory bandwidth through a better balance of the memory traffic. For example, the 'share-nothing' core-filling policy, presented in Chapter 3, distributes the threads in all the available sockets, so that a minimum resource sharing is achieved. The advantage of this policy is the minimization of the contention in shared resources of the memory data path (e.g., caches, bus interface, memory controller etc.). Specifically for the shared caches, such a placement increases the size of the system's aggregate cache available to the computation, therefore relieving further the main memory subsystem. As a result, SpMV scales at a steady pace up to the 12-threaded configuration in Dunnington, before the memory bus is eventually saturated (see Chapter 3, Section 3.3.2). Unfortunately, such a performance improvement is not a free lunch. Not only the increased processor utilization augments its power dissipation, but, more importantly, the use of multiple caches can lead to a considerable increase in the total power dissipation of a system that supports dynamic leakage control and is able to put in 'sleep mode' unused caches or even whole unused processors. This drift in power dissipation, however, may be compensated by the faster execution time and a possible frequency scaling through the DVFS mechanism.

Finally, the performance of the SpMV kernel is directly related to the structure of the input matrix. Apart from the implications in the SpMV performance that a sparse matrix may have (memory bottleneck, irregular accesses, load imbalances etc.), the energy footprint of SpMV is likely to vary from matrix to matrix. For example, it might be a waste of energy to assign multiple sockets to the computation of an irregular matrix with load imbalances, while such a placement sounds reasonable for a more regular one. It is therefore important to quantify and consider such information in the effort to predict the optimal performance-energy tradeoffs for the SpMV kernel.

7.2.1 Experimental setup

For the evaluation of the performance-energy tradeoffs of the SpMV kernel, we have used the Dunnington system for the performance measurements (see Chapter 3, Section 3.2) and the McPAT framework [Sheng et al., 2009] for detailed power estimations. The Dunnington system comprises four Intel X7460 packages built at 45 nm technology. Each X7460 package is essentially a set of three dual-core modules with a 3 MiB shared cache and the whole package hosts a 16 MiB unified L3 cache. The processor supports DVFS and offers two frequency steps, namely at 2.14 GHz and at 2.67 GHz (full speed). Each frequency step is associated with a specific supply voltage, which is regulated accordingly upon transition to the specified frequency [Int, 2008]. We use the cpufrequtils userspace tools in Linux for controlling the processor's frequency.

The alternative thread placements we consider fall in three categories depending on the sharing of common resources:

(a) *Full sharing*, in which threads are placed as close as possible, sharing all the levels of the cache hierarchy (same as the 'share-all' policy described in Chapter 3).

(b) *Semi sharing*, in which threads are placed so that they share only the L3 cache, but not the L2 cache.

(c) *No sharing*, in which threads are placed as sparsely as possible, so as to minimize the sharing of the cache hierarchy (same as the 'share-nothing' policy described in Chapter 3).

We use the notation XpYsZc to describe a placement using Z cores in total, Y sub-packages (shared L2 components) and X physical packages or sockets (shared L3). For example, the possible thread placements for four threads are 1p2s4c, 2p4s4c and 4p4s4c. We consider in total 26 different thread placements for the following thread counts: 1, 2, 3, 4, 6, 8, 9, 12, 15, 18, 24.

The McPAT framework predicts power dissipation based on specific processor event counts (e.g., core cycles, cache accesses/misses etc.). Since it is intended for use with processor simulators, it supports a very detailed list of events, which are not all available in real processors. For this reason, we included events that are a close match to the ones required by McPAT and contribute to the overall power dissipation of the processor [Isci and Martonosi, 2003; Singh et al., 2009]. In total, we included 14 event counts, which are de-

Event Mnemonic	Description
	CORE EVENTS
UnHalted Core Cycles	Unhalted core cycles
UOPS_RETIRED.ANY	Micro-ops retired
SIMD_UOPS_EXEC	SIMD micro-ops executed
	BRANCH UNIT EVENTS
Branch Instruction Retired	Branch instructions retired
Branch Misses Retired	Mispredicted branch instructions retired
	CACHES EVENTS
L1I_READS	Instruction fetches
L1I_MISSES	Instruction fetch unit misses
L1D_REPL	Cache lines allocated in the L1 data cache
L1D_ALL_REF.ANY	All references to the L1 data cache
L2_RQSTS	L2 cache requests
L2_LINES_IN	L2 cache misses
LLC Reference	Last level cache references
LLC Misses	Last level cache misses
	BUS EVENTS
BUS_TRANS_ANY	All bus transactions

Table 7.1: Performance monitoring events used for the prediction of power dissipation by the McPAT framework. For more information on the performance monitoring events of Intel microarchitectures, please refer to [Int, 2010] using the event mnemonic.

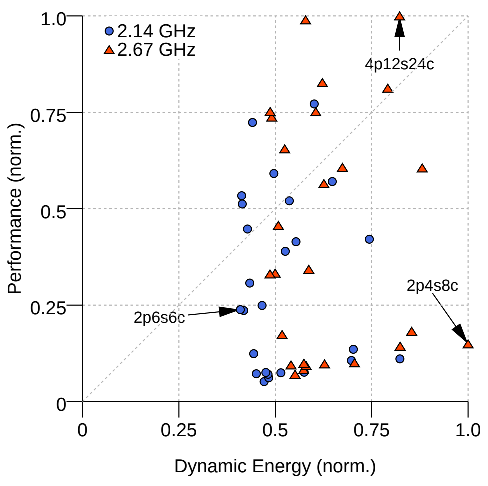
tailed in Table 7.1. The Intel Core microarchitecture of Dunnington includes only two programmable performance monitoring counters, so multiple runs of the SpMV kernel were required to collect all the event counts. Finally, we do not include in our overall power measurements the idle power of unused processor packages, as if we were able to put them in a low-leakage sleep mode.

7.2.2 Characterizing the tradeoffs

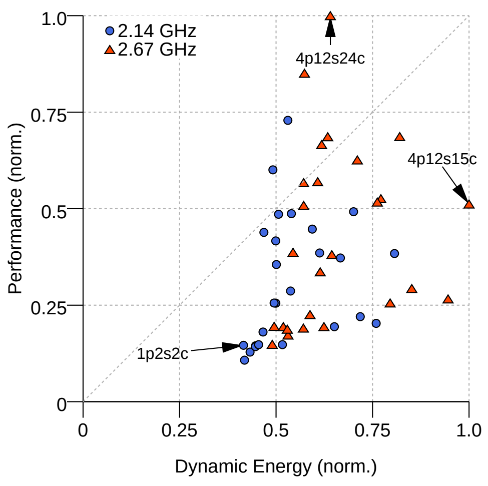
Figure 7.1 presents the performance-energy tradeoffs of 52 execution configurations (thread placement, core frequency) of the SpMV kernel in Dunnington for four sparse matrices with different computational characteristics. More specifically, xenon2 and parabolic_fem are both small matrices, fitting in the system's aggregate cache, but the latter is quite irregular. Conversely, boneS10 and thermal2 are both large enough to fit in the system's aggregate cache, but the second has an irregular structure. All matrices were stored in CSX format. Performance and dynamic energy consumption (dynamic power $\times$ execution time) are normalized over the overall best performance and the maximum energy consumption, respectively. The first key observation is that the performance-energy landscape is rather broad and two execution configurations might be quite apart in terms of energy-efficiency. Additionally, at the same energy level there exist multiple configurations with varying performance and vice versa. This requires a method for characterizing each tradeoff, an issue we discuss later in this section. Another key observation is that the performance-energy landscape differs between small and large regular matrices, while for irregular ones the difference is not so wide.

A closer examination of the landscape for matrix xenon2 (31 MiB in CSX) reveals that the most energy-efficient execution configuration is the one using six threads spread sparsely (no sharing of L2 caches) across two packages and clocked at 2.14 GHz. At the other end, using two packages and eight threads packed together to share the L2 caches at 2.67 GHz is the least energy-efficient configuration, consuming almost 2.5$\times$ more energy for a fraction of the performance. This is a typical example of how beneficial a correct thread placement at a lower frequency can be. Although, the six-threaded configuration uses more shared hardware resources, the lower frequency allows a 35% saving in power dissipation compared to the high-frequency eight-threaded configuration (4.4 W vs. 6.8 W). From a performance perspective, the six-threaded configuration is also more beneficial, since it eliminates the contention in the shared L2 caches, now acting as private caches, and offers a significantly higher performance (+37%), despite the lower frequency. A key observation, common in more regular matrices, is the steep increase in energy-efficiency. For example, the 3p9s9c execution configuration at 2.14 GHz in xenon2 has twice the performance of the most energy-efficient 2p6s6c with almost the same energy consumption[3]. This is due to an almost equal increase in performance and power consumption, which is typical of the linear scaling that SpMV encounters on regular matrices when more sockets are added to the computation. Despite the same steep increase in energy-efficiency, large matrices, exceeding the system's aggregate cache, exhibit a slightly different performance-energy landscape. For example, the most energy-efficient configuration for boneS10 is the 1p2s2c two-threaded configuration at 2.14 GHz, while the least energy-efficient is the 15-threaded at 2.67 GHz. It is worth noting the memory bottleneck and its effect on energy consumption. The most performant configurations for boneS10 are the 12-threaded and the eight-threaded configuration using all four sockets at 2.67 GHz. The memory bus is already saturated from

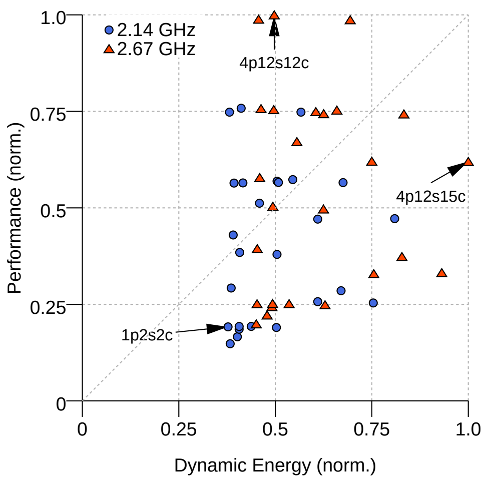
[3] The doubling of performance when moving from six to nine threads is due to the comfortable fitting of the working set in the aggregate cache of the three packages.



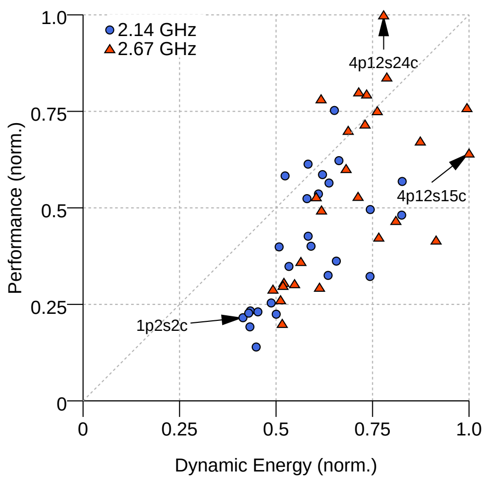
(a) Small, regular (xenon2).



(b) Small, irregular (parabolic_fem).



(c) Large, regular (boneS10).



(d) Large, irregular (thermal2).

Figure 7.1: Performance-energy tradeoffs of the SpMV kernel for a set of matrices with different performance characteristics. Execution configurations are in the form XpYsZc denoting that *X* sockets, *Y* sub-packages (shared L2 components) and *Z* cores are used in total.

the eight-threaded configuration and adding four more threads offers only a marginal performance improvement; trying to add even more threads is a pure waste of energy, which is eventually doubled at the 15-threaded configuration[4].

The landscape of the performance-energy tradeoffs in irregular matrices is similar to that of large regular matrices, but with a less steep increase in energy-efficiency. Matrix size seem not to play a significant role, since both the small parabolic_fem and the larger thermal2 have the same most and least energy-efficient configurations, namely the 1p2s2c at 2.14 GHz and the 15-threaded at 2.67 GHz, respectively. This should be expected, since the key performance problem of these matrices is not the memory bandwidth bottleneck (see Chapter 3), therefore, matrix size is not very critical. For the same reason, there is no such a steep increase in energy-efficiency; since SpMV's scaling is hindered, the gain in performance from the use of more cores is less important than the resulting increase in power dissipation.

Optimal tradeoffs

Having examined in more detail the performance-energy landscape of the SpMV kernel, the question that arises is what consists a good tradeoff. Apparently, points in the upper left corner of the performance-energy landscape are good tradeoffs, since they maximize performance, while keeping energy requirements low. However, the question that poses now is how we could compare tradeoffs against each other and if there exist a single best tradeoff. Suppose two points, let (e, p) and (e', p'), in the performance-energy landscape that correspond to execution configurations E and E'. If $e < e'$ and $p \geq p'$, then E is definitely a better tradeoff than E', since with a lower energy consumption, it achieves at least the same performance; the same is true for two configurations where $e \leq e'$ and $p > p'$. In this case, configuration *E strictly dominates* configuration E'. On the other hand, if $e \leq e'$ and $p \leq p'$, we cannot assess whether E is a better tradeoff than E'. The set of points (and the corresponding execution configurations) that are not dominated by any other point in the performance-energy landscape form the *non-dominated Pareto front* [Luke, 2011]. The configurations on the Pareto front are the set of optimal tradeoffs, since they are strictly better than any other configuration in at least one objective (performance or energy). An important property of the Pareto front is that when moving from one point to another, we experience the least possible loss in energy consumption and the greatest gain in performance and vice versa. In that sense, all points of the Pareto front are formally equivalent tradeoffs.

[4] The 15-threaded configuration suffers also from load imbalance, since six threads must share the L2 cache, while the rest use it exclusively.

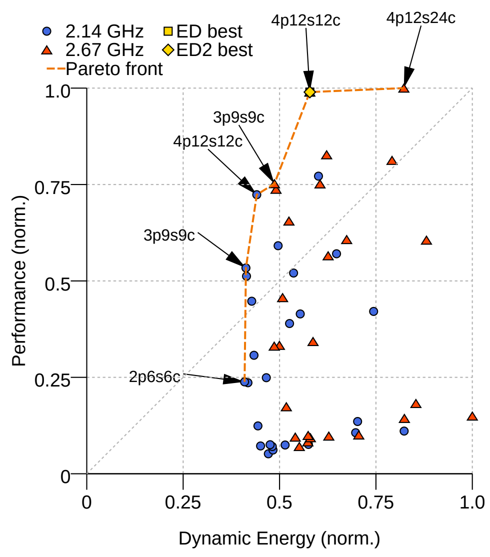
Therefore, there is not a single best tradeoff of performance and energy (unless the Pareto front collapses to a single point), but rather a set of best tradeoffs.

Optimality of energy-delay products Each tradeoff corresponding to a minimal energy-delay product for some $n \geq 0$ is an optimal tradeoff in the Pareto sense. This can be easily proved: assume that the execution configuration M minimizes the ED^n product for some $n \geq 0$, i.e., $e_M d_M^n < e_{M'} d_{M'}^n$ for each $M' \neq M$. If we suppose that (e_M, d_M) does not lie on the Pareto front, then there must be a configuration C, such that either $e_C < e_M$ and $d_C \leq d_M$ or $e_C \leq e_M$ and $d_C < d_M$. Therefore, the energy-delay product of this configuration will be $e_C d_C^n \leq e_M d_M^n$ for $n = 0$ or $e_C d_C^n < e_M d_M^n$ for $n > 0$, which contradicts with our initial assumption that the configuration M minimizes the energy-delay product. Therefore, M is a non-dominated solution, i.e., a point on the Pareto front. As a corollary, every point of the Pareto front is a minimal energy-delay product for some $n \geq 0$.

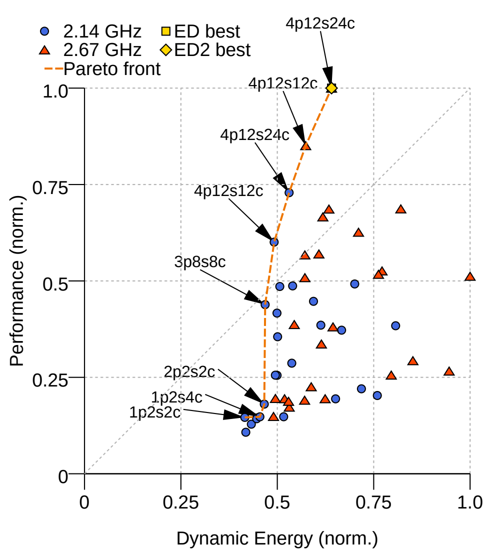
The most typical energy-delay products used in practice are the energy (E), energy-delay (ED) and energy-delay squared (ED^2) products, in an increasing bias toward higher performance. Figure 7.2 shows the Pareto front for the xenon2 and parabolic_fem matrices along with the ED and ED^2 products.

7.3 Predicting the optimal execution configurations

As SpMV kernel's performance depends heavily on the input matrix structure, it is desirable, given an unknown matrix, to be able to predict those multicore execution configurations that lead to an ideal performance-energy tradeoff, in order to spare hardware resources and achieve possible energy savings. According to the discussion and experimental evaluation in Chapter 3, two of the most definitive parameters for the SpMV kernel performance are the matrix size and the flop:byte ratio of the matrix, which is directly related the average non-zero elements per row. If the matrix is small enough to fit in the aggregate cache of the underlying architecture, SpMV will experience a significant performance improvement, since the contention for main memory bandwidth is eliminated. However, there exist matrices where the memory bandwidth contention is not the key performance problem, as these suffer from irregular accesses, loop overheads, load imbalances etc. We showed in Chapter 3 that there exist a significant correlation between the flop:byte ratio of a matrix and its performance, while in the last section, we identified also differences in the performance-energy landscape of SpMV, depending on the size and the sparsity of the matrix. This information can be therefore useful in the prediction of the optimal execution configurations for the SpMV kernel.



(a) xenon2 matrix.



(b) parabolic_fem matrix.

Figure 7.2: The optimal performance-energy tradeoffs for two sparse matrices. The points on the Pareto front are annotated with the corresponding execution configurations. The configurations minimizing the energy-delay products are also shown.

Our goal is to predict a broad set of optimal execution configurations; for this reason we rely on a machine learning approach based on clustering, in order to approximate the Pareto front of the optimal configurations. The key concept behind this idea is that we expect matrices with similar structural characteristics, e.g., large regular matrices, to have similar performance-energy tradeoffs. The procedure of learning is based on an initial clustering of matrices according to their size and their average non-zero elements per row. For each cluster, we construct a representative Pareto front from the execution configurations that lie on the Pareto fronts of the matrices in the cluster. Each cluster is then represented by its geometric center (in the space of the matrix attributes) and its Pareto front.

7.3.1 Clustering the matrices

In order to obtain a representative set of matrices and more samples for training of our model, we have expanded our initial matrix suite (see Chapter 3, Table 3.1) to contain 50 sparse matrices, all selected from the University of Florida Sparse Matrix collection [Davis and Hu, 2011]. Each matrix is assigned a vector of two attributes reflecting its sparsity and size. Since clustering depends on the distances of the matrices in the attribute space, it is crucial that the attribute values represent a valid similarity measure. For example, considering as the size attribute the matrix size relative to the total aggregate cache of the underlying system can lead to misleading results, since very large matrices, such as boneS10, are likely to become falsely *outliers*, i.e., not belonging to any cluster. In practice, we do not care how large is a matrix, if it significantly exceeds the system's aggregate cache; therefore, such a metric is not a valid similarity measure. The exact is true for a pure absolute or relative average-nonzeros-per-row metric. For this reason, we assign values in a stepwise fashion, trying to reflect better the similarity of matrices. More specifically, assuming C is the system's aggregate cache and S the matrix size, we calculate the size attribute as following, identifying different cases for matrices fitting in the aggregate cache of two, three or four sockets (full system)[5]:

$$a_m = \begin{cases} 1, & S/C \in (0, 1/2) \\ 1.5, & S/C \in [1/2, 3/4) \\ 3, & S/C \in [3/4, 1) \\ 3.5, & S/C \in [1, \infty) \end{cases} \tag{7.2}$$

Similarly, we set the barrier for very sparse matrices at 15 non-zero elements per row and calculate the sparsity attribute as following:

$$a_s = \begin{cases} \frac{1}{15} \cdot \frac{NNZ}{N}, & \frac{NNZ}{N} \leq 15 \\ 2, & \frac{NNZ}{N} > 15 \end{cases} \tag{7.3}$$

Figure 7.3 shows the formed matrix clusters using a hierarchical clustering technique. Our attribute value assignment allowed the clustering algorithm to correctly identify the four basic matrix categories, namely 'small and sparse', 'small and regular', 'large and sparse' and 'large and regular', without leaving irrelevant outliers.

7.3.2 Constructing the cluster Pareto front

Assuming that the clustering of the matrices reflects correctly their common performance characteristics, the cluster Pareto front will consist of the most common execution configurations present in the Pareto fronts of every matrix in the cluster. In order to construct the cluster Pareto front, we start iterating

[5] We do not include a case for matrices fitting in the L3 cache of a single socket, since our suite does not have so small matrices.

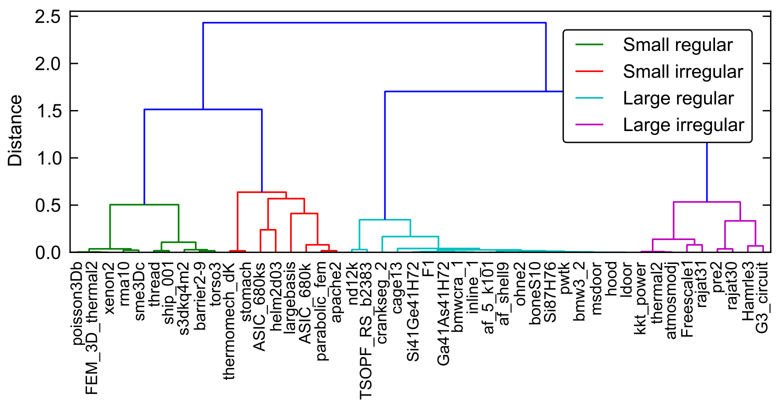


Figure 7.3: The four major clusters in our extended matrix suite.

the Pareto fronts of the matrices in the cluster from the most performant configuration to the most energy-efficient, i.e., from the upper-rightmost Pareto point down to the leftmost. During the iteration, we record the execution configurations that correspond to the current point for every matrix in the cluster and select the most frequent. If two or more configurations have the same frequency, we select both of them if their frequency is more than 25% of the cluster size, otherwise we select the one with the least number of threads, in order to cover better the low-energy configurations. We also use this technique to provide predictions about the configurations minimizing the *ED* and ED^2 products.

Avoiding overfitting The Pareto fronts of the different matrices in a cluster do not necessarily have the same amount of points. The proposed method for constructing the cluster Pareto entails the risk of creating a Pareto front with so many points that will not be representative of the cluster, covering even corner cases. To avoid this situation of *overfitting*, we stop the construction of the Pareto front as soon as we 'run out' of points for more than half of the matrices in the cluster. Figure 7.4 shows an example of the construction of the cluster Pareto front.

7.3.3 Classification and testing

After the training procedure is over, we store the cluster centers and the associated cluster Pareto fronts for future use. The cluster center is the geometric median of its comprising matrices in the attribute space, while the cluster Pareto is constructed as described before. These two features characterize every discovered matrix category. When a new unseen matrix appears, we calculate its size

Matrix #1	*Matrix #2*	*Matrix #3*	*Cluster Pareto*
4p12s24c@2.67	**4p12s24c@2.67**	**4p12s24c@2.67**	*4p12s24c@2.67*
4p12s12c@2.67	**4p12s12c@2.67**	**4p12s12c@2.67**	*4p12s12c@2.67*
4p8s8c@2.67	**4p8s8c@2.67**	**4p8s8c@2.67**	*4p8s8c@2.67*
4p12s24c@2.14	**4p12s24c@2.14**	**4p12s24c@2.14**	*4p12s24c@2.14*
4p12s12c@2.14	**4p12s12c@2.14**	**4p12s12c@2.14**	*4p12s12c@2.14*
4p8s8c@2.14	**4p8s8c@2.14**	**4p8s8c@2.14**	*4p8s8c@2.14*
2p6s6c@2.14	1p2s2c@2.14	**2p6s6c@2.14**	*2p6s6c@2.14*
1p3s3c@2.14	n/a	1p3s6c@2.14	*1p3s3c@2.14*
1p2s2c@2.14	n/a	1p2s4c@2.14	*1p2s2c@2.14*
n/a	n/a	1p2s2c@2.14	n/a

Figure 7.4: Construction of the cluster Pareto front for a cluster of three matrices. Bold typeface shows the configurations that are selected to be part of the constructed Pareto front.

and sparsity attributes, according to equations (7.2) and (7.3) and classify it to the closest matrix cluster in the attribute space. We retrieve then the execution configurations of the Pareto front of the matching cluster and assign it to the new matrix.

In order to assess the quality of our predictions, we assign a rank to every execution configuration of the predicted Pareto front reflecting its optimality. To achieve this, we first compute iteratively a set of Pareto fronts for all the performance-energy tradeoffs of the target matrix. The first Pareto front is the set of optimal configurations as described in Section 7.2.2 (optimality rank zero). In every iteration i, we omit the points of the previously computed fronts and recompute a new Pareto front; the points of this front are the i-th best execution configurations (optimality rank i). Figure 7.5 shows an example of the three first Pareto fronts of matrix thermal2. Each point of the predicted Pareto front, therefore, is assigned the optimality rank of the real Pareto front it lies on. The average rank of all predicted configurations is the overall quality measure of our prediction; we call this measure the *rank of the predicted Pareto*.

We test our prediction method through a *five-cross-validation (5-CV)* technique. Cross-validation is a common statistics technique for assessing the accuracy of a prediction model and consists of the following steps:

(a) Shuffle uniformly the initial data set and split it into a fixed number of equal partitions (also known as *folds*). In our case, the data set is the matrix suite of the 50 matrices, which is split into five folds.

(b) Keep one fold for testing and use the rest for training.

(c) Repeat previous step until all folds are tested.

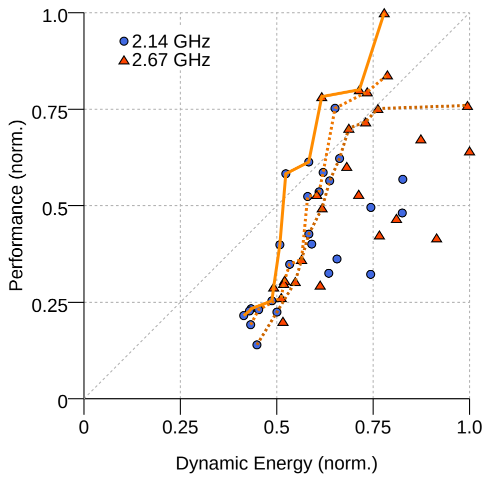
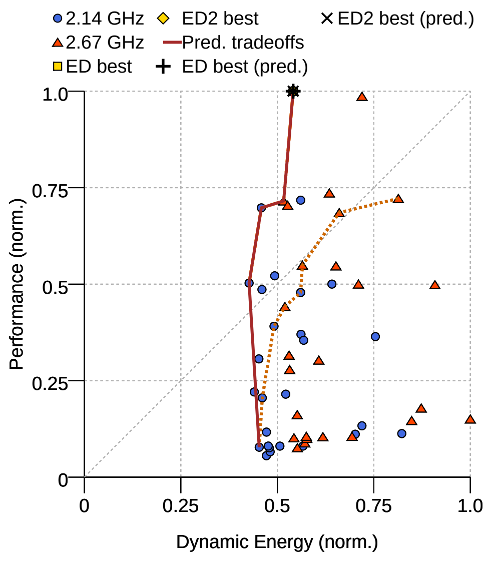


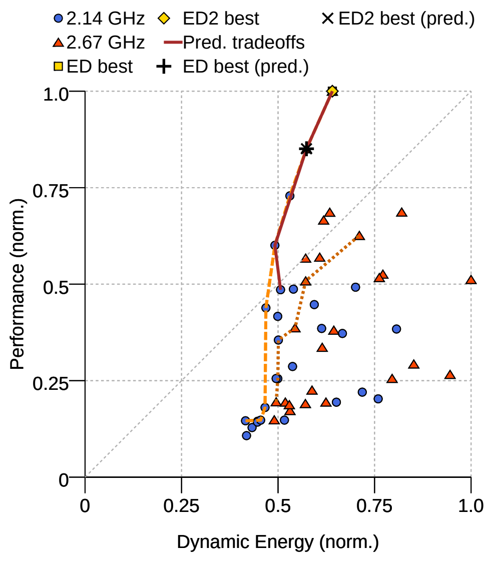
Figure 7.5: The three first Pareto fronts of the matrix thermal2. The execution configurations on the first front are optimal, while those lying on the *i*-th front constitute the *i*-th best solutions.

Table 7.2 summarizes the results of the 5-CV test. For each fold, it is depicted the rank of the predicted Pareto front and the percentage of predictions with a rank below a certain threshold. Our prediction methodology manages to provide rather accurate predictions with an average prediction rank close to zero. In fact, 90% of the predicted Paretos have a rank below one on average, while all have a rank below two. Figure 7.6 shows typical predictions for matrices from every cluster. The predictions for regular matrices are quite accurate and tend to become a perfect match for larger ones. On the other hand, predictions for very sparse matrices, e.g., thermal2, tend to be slightly less accurate. This should be expected, since our sparsity attribute (average non-zero elements per row) reflects accurately only the loop overheads of very sparse matrices. Although very sparse matrices are likely to suffer also from irregular accesses and load imbalances, this rule has exceptions; for example, a reordered version of an irregular matrix does not have such performance limitations (see Chapter 3, Section 3.3.2). However, we have chosen not to fragment further the category of very sparse matrices, since our initial data set is small enough to allow an accurate training of such sub-categories.

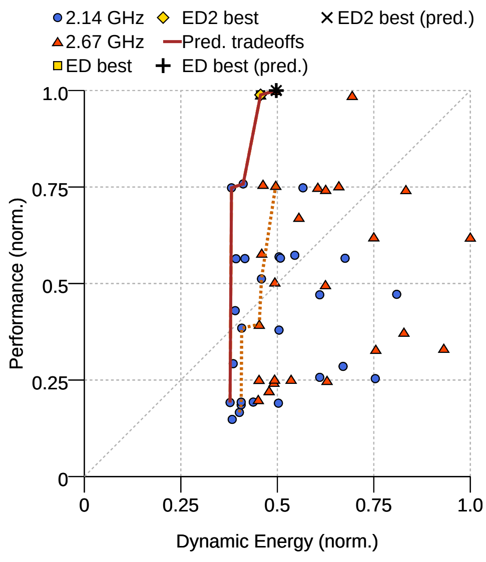
Finally, Table 7.3 summarizes the accuracy of the energy-delay predictions. More specifically, for each of the three products selected, namely energy, energy-delay and energy-delay squared, it is depicted how far our predictions lie from the original minimal products. For example, our predictions for the most energy-efficient configurations lead to an average 7.6% more energy consumption than



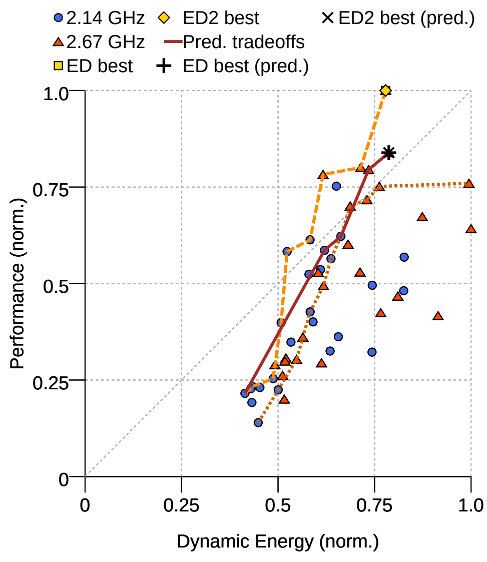
(a) Small regular (torso3).



(b) Small irregular (parabolic_fem).



(c) Large regular (boneS10).



(d) Large irregular (thermal2).

Figure 7.6: Optimal execution configurations predictions and predictions for the minimal *ED* and ED^2 products. The rank-zero and rank-two Pareto fronts are also shown.

Fold No.	Prediction Rank	Rank < 1	Rank < 2
#1	0.45	100.0%	100.0%
#2	0.27	77.8%	100.0%
#3	0.37	88.9%	100.0%
#4	0.19	100.0%	100.0%
#5	0.27	88.9%	100.0%
Average	0.31	91.1%	100.0%

Table 7.2: Cross-validation results for the proposed prediction methodology. It is depicted the rank of the predicted Pareto front and the percentage of predictions with a rank below a certain threshold.

Fold No.	Energy	*ED*	ED^2
#1	6.9%	1.6%	6.6%
#2	12.5%	10.5%	17.7%
#3	13.2%	4.1%	8.0%
#4	2.1%	0.9%	0.0%
#5	3.5%	8.5%	14.2%
Average	7.6%	5.1%	9.3%

Table 7.3: Energy-delay prediction accuracy. The average per-fold distances from the original minimal products are depicted.

the optimal. Of similar accuracy are the predictions for the rest of the considered products, with *ED* products being approximated with a 5.1% difference and ED^2 with 9.3% on average.

7.3.4 Limitations

The proposed methodology for predicting the SpMV execution configurations that lead to optimal performance-energy tradeoffs is able to provide rather accurate predictions in a diverse set of sparse matrices. However, some limitations still exist. First, despite the accurate execution configurations predictions, our method cannot currently predict how far these configurations lie in the performance-energy landscape. For example, the configurations 1p2s2c and 4p8s8c at 2.14 GHz for boneS10 (Figure 7.1), although equivalent in terms of Pareto optimality, consume almost the same amount of dynamic energy for a more than 3× performance difference; it is clear that the 1p2s2c could have been omitted as a 'not-so-ideal' tradeoff in practice. This limitation can be raised with a more 'loose' construction of the Pareto front, which would include only points with a variation of the objectives above a certain threshold.

A second limitation involves the matrix clustering. While the matrix size-related attribute is well formed, the sparsity-related attribute does not account for specific intricacies of very sparse matrices that affect SpMV performance, e.g., irregular non-zero element structure and load imbalances. As a result, a deterioration of accuracy is observed in such cases. Including this information for clustering requires a full matrix scan and a reasonable quantification of these characteristics, which will reflect correctly the similarity or dissimilarity of the matrices in terms of performance.

7.4 Open issues

In this chapter, we have taken a first approach on identifying the performance-energy tradeoffs of the SpMV kernel and predicting the execution configurations that lead to optimal performance-energy compromises, depending on the input matrix. A number of issues, though, still remain open and should be addressed in the future. Our approach focused solely on the dynamic power and energy consumption of the processor ignoring leakage power dissipation. According to McPAT's predictions, the leakage power of the four sockets of the Dunnington system amounts to 88 W, while the maximum dynamic power recorded was slightly over 50 W. On average, matrices fitting in the system's aggregate cache contributed a 37% increase in the combined power dissipation of all processors in the system, while the total processor power dissipation for large matrices was only 12.5% higher. This difference is quite expected, since SpMV under-utilizes the processor for large matrices (IPC < 1) due to the memory bottleneck. Nonetheless, even these numbers are half-truth, since main memory power dissipation is not considered. In fact, main memory operations are quite power-hungry consuming almost $50\times$ more energy than processor's arithmetic operations [Dongara, 2012]. Indeed, Kamil et al. [2008] show that memory-intensive benchmarks, such as STREAM and CG, are more power-hungry than CPU-intensive ones in a full-system scale. And the SpMV kernel is not an exception. It is therefore important to consider the performance-energy tradeoffs of the SpMV kernel from a full-system utilization context. Despite the several questions that remain to be answered in this direction, we believe that the preliminary results presented in this chapter and the proposed methodology for identifying and predicting the optimal execution configurations will remain relevant also in a full-system performance-energy analysis, where the main memory power dissipation will play a definitive role.

8 Conclusions

This thesis focused on the optimization of the Sparse Matrix-Vector Multiplication kernel (SpMV) for modern multicore architectures. We performed an in-depth performance analysis of the kernel and identified its major performance bottlenecks. This allowed us to propose an advanced storage format for sparse matrices, the Compressed Sparse eXtended (CSX) format, which targets specifically the minimization of the memory footprint of the sparse matrix. This format provides significant improvements in the performance of the SpMV kernel in a variety of matrices and multicore architectures, maintaining a considerable performance stability. Finally, we investigated the performance of the SpMV kernel from an energy-efficiency perspective, in order to identify the execution configurations that lead to optimal performance-energy tradeoffs. This chapter summarizes the basic conclusions and achievements of this work, providing also the author's vision for the future research prospects and directions.

8.1 SpMV: Victim of the memory wall

The SpMV kernel lies at the heart of iterative solution methods for sparse linear systems, which arise in a variety of scientific domains. The key performance problem of SpMV stems primarily from its streaming nature, incurring a very low flop:byte ratio, which eventually poses a considerable pressure to the memory hierarchy. Performing an in-depth performance analysis of the SpMV kernel in this thesis, we highlighted this characteristic—somehow overlooked in the past—as the major performance bottleneck of the kernel in modern multicore architectures. In SMP architectures, this problem defines completely the SpMV performance for large matrices in a highly multithreaded context, as the common front-end bus is quickly saturated. In NUMA architectures, on the other hand, the integrated memory controllers leave more headroom and the memory path is not completely saturated until the full system is utilized; this allows some computational optimizations, which in SMP systems have no impact at all. Nonetheless, SpMV performance in NUMA systems is quite sensitive to the correct placement of the threads' data at the participating memory nodes, since the interprocessor links can be easily saturated. Respecting the correct data placement for shared data structures may require considerable porting effort and lead to an important code refactoring of an existing SMP-based code. To facilitate this process, we have introduced a simple NUMA-aware interleaved allocator that transparently places parts of a shared data structure on the correct memory nodes without affecting the SMP-based logic of the user code; simply replacing the memory allocation calls suffices for a successful porting.

In this thesis, we took a closer look at blocking storage formats for sparse matrices. Blocking has been successfully used in the past as an alternative storage method for sparse matrices, since it not only allows the reduction of the matrix size but also offers good computational characteristics. We provided a detailed performance analysis of the most typical blocking methods, identifying the merits and weaknesses of each one. The key conclusion of this analysis was the compression vs. multithreaded performance tradeoff between the variable and fixed size blocking methods. While leading to considerable gains in the matrix size representation, especially for matrices with an irregular structure, variable size blocking methods should pay the cost of the additional bookkeeping needed to store the different block size values. The increased compression starts to pay off as the number of threads increases and the pressure to the memory subsystem becomes of crucial importance for the performance of the SpMV kernel; in a highly multithreaded context, therefore, variable size blocking formats can significantly outperform their fixed size counterparts. Fixed size blocking methods, on the other hand, lead to a simpler kernel implementation, allowing not only a better code generation from the compiler, but also advanced computational optimizations, such as vectorization. Focusing on BCSR, the most representative fixed size blocking storage format, we investigated the implications of the block shape on the performance of the SpMV kernel. These can be quite important in cases where the memory bandwidth is not completely saturated and computational optimizations can offer a significant performance benefit. Toward this direction, we have developed a performance model that considers both the memory and the computational part of the SpMV kernel and is able to accurately predict the optimal block for BCSR.

8.2 CSX: A viable approach to a high performance SpMV

Two are the main drawbacks of previous approaches in optimizing the performance of the SpMV kernel. First, these approaches are not very focused on the minimization of the memory footprint of the sparse matrix, since their origins go back to different backgrounds, e.g., register optimizations, and, second, they take an 'all-or-nothing' approach. For example, the BCSR format will provide a significant performance improvement in matrices dominated by dense two-dimensional regions, but its performance will plummet in other cases. Motivated by these shortcomings of alternative storage formats for SpMV, we have proposed in this thesis the Compressed Sparse eXtended format (CSX), an explicit compression-based and integrated format. CSX is able to detect and encode in the same representation a multitude of substructures present in sparse matrices, namely horizontal, vertical, diagonal, anti-diagonal and two-dimensional blocks. Combined with a highly compressed representation based on run-length encoding of the column indices, CSX is able to reduce the matrix representation size close to the theoretical bounds. As a result, CSX provides considerable performance improvements (exceeding 50% on average) in SMP systems, where the memory bottleneck is more pronounced. In NUMA architectures, we take particular care in balancing the decompression overhead. More specifically, we relax the compression scheme and use an advanced substructure selection heuristic that considers also the computational overhead of decompression. These optimizations allow CSX to provide a more than 20% average performance improvement in NUMA systems, outperforming other alternative storage formats.

An important trait of CSX is its performance stability. While it is able to provide significant performance improvements in matrices with a lot of substructures, its performance is at least comparable to the baseline CSR format in very sparse and irregular matrices, proving a high adaptability to the matrix structure. Additionally, we placed considerable effort in minimizing the matrix preprocessing cost (detection and encoding of substructures), in order to render CSX a viable alternative in the context of 'real-life' solvers. Indeed, CSX was a able to accelerate the performance of the Elmer solver nearly 15%, despite its preprocessing cost and the increased preconditioning cost for the considered benchmark problems.

A significant extension of CSX, proposed in this thesis, is the support for symmetric sparse matrices. Symmetric sparse matrices may arise in the discretization of PDEs and it is quite tempting—in the context of the SpMV kernel—to store only the lower triangular part and the main diagonal. Despite the obvious benefit of this layout in matrix size reduction, the multithreaded execution of the symmetric SpMV kernel inserts a RAW dependency on the output vector, rendering the efficient implementation of the kernel problematic. Most approaches eliminate this dependency either using local per-thread vectors or a combination of per-thread vectors and selective locking. Using a local vector per thread requires an additional final reduction step, whose overhead is growing linearly with the thread count, limiting therefore the scalability significantly, while the cost of excessive locking, on the other hand, can be prohibitive. Our approach in optimizing the symmetric SpMV kernel exploited the fact that local vectors become quite sparse as the thread count increases. For this reason, we employ an indexing scheme for the local vectors to reduce only the conflicting elements; all other vector updates are pushed directly to the output vector. This technique decouples in practice the reduction overhead from the thread count and allows the symmetric SpMV kernel to scale. In combination with the CSX variant for symmetric matrices, which leads to nearly 67% compression ratios, we were able to accelerate more than 2$\times$ the performance of the SpMV kernel, an improvement that is visible also from the context of the CG iterative solution method.

8.3 Toward an energy-efficient SpMV

In this thesis, we took a first step toward investigating the performance-energy tradeoffs of the SpMV kernel. Motivated by the memory-intensive nature of SpMV, we investigated alternative core frequencies and thread placements, in order to achieve high performance at a low energy budget. Modern processors support dynamic voltage and frequency scaling, allowing considerable energy savings without sacrificing performance in non-CPU-intensive workloads. As such, SpMV's performance depends greatly on the placement of threads on the available cores, since a sane placement can alleviate the pressure to the memory subsystem. As a result, a combination of a low processor frequency and an efficient thread placement may lead to considerable energy savings, sacrificing only a small fraction of the overall performance. Based on the assumption that matrices with similar performance characteristics (mainly due to matrix structure) will exhibit a similar behavior in energy consumption, we proposed a machine learning approach for predicting the execution configurations leading to the optimal performance-energy tradeoffs. Our preliminary results are promising enough and pave the path for a further investigation of the energy-efficient execution of the SpMV kernel.

8.4 Future research directions

Memory intensity is an inherent problem of the SpMV kernel. In fact, it is a scientific kernel with one the smallest flop:byte ratios and it is expected to be bound from memory bandwidth in the foreseeable future, unless we experience a considerable breakthrough in main memory technology. As a result, we expect the notion of data compression, employed also by CSX in this thesis, to be of actual interest in the next years. However, CSX has reached the limits of matrix metadata compression and it is therefore essential to extend compression to the matrix data itself, i.e., its non-zero values. The performance gains of symmetric storage formats provide a clear indication on the benefits of an efficient non-zero value compression. The key challenges in this approach, however, is that the compression must respect the double precision accuracy of the values and a good balance of the decompression cost is needed, in order not to hog the overall computation.

Data compression techniques are gaining recently an increasing interest not only as a means of tackling the ever increasing memory-processor speed gap in memory-intensive applications, but also as a means of reducing the energy consumption of the memory subsystem. This adds another dimension in the optimization of memory-intensive applications and opens interesting research directions. In the context of a scientific application, a combination of data compression and advanced power management techniques in the processor could provide the grounds for minimizing its energy footprint and maximizing performance.

Finally, from a more technical point of view, it is important to enrich CSX with a functional library interface so that it will be easily exploited by the HPC community. Toward the same direction, a further improvement of matrix preprocessing through advanced caching techniques will make CSX more appealing and help its dissemination.

List of publications

The work in this thesis produced a number of publications in a series of well-known, high-quality journals and conferences of the HPC community.

Journal publications

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)*. To Appear.

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.

Conference publications

V. Karakasis, G. Goumas, K. Nikas, N. Koziris, J. Ruokolainen, and P. Råback. Using state-of-the-art sparse matrix optimizations for accelerating the performance of multiphysics simulations. In *PARA 2012: Workshop on State-of-the-Art in Scientific and Parallel Computing*, Helsinki, Finland, 2012. Springer.

V. Karakasis, G. Goumas, and N. Koziris. Exploring the performance-energy tradeoffs in sparse matrix-vector multiplication. In *Workshop on Emerging Supercomputing Technologies (WEST), ICS'11*, Tucson, AZ, USA, 2011.

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.

V. Karakasis, G. Goumas, and N. Koziris. Perfomance models for blocked sparse matrix-vector multiplication kernels. In *38th International Conference on Parallel Processing (ICPP'09)*, pages 356–364, Vienna, Austria, 2009. IEEE Computer Society.

V. Karakasis, G. Goumas, and N. Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *12th IEEE International Conference on Computational Science and Engineering (CSE-09)*, Vancouver, Canada, 2009. IEEE Computer Society.

V. Karakasis, G. Goumas, and N. Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–8, Rome, Italy, 2009. IEEE Computer Society.

G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'08)*, Toulouse, France, 2008. IEEE Computer Society.

G. Goumas, N. Drosinos, V. Karakasis, and N. Koziris. Coarse-grain parallel execution for 2-dimensional PDE problems. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–8, Long Beach, CA, USA, 2007. IEEE.

Bibliography

A. Agarwal, S. Mukhopadhyay, A. Raychowdhury, K. Roy, and C. H. Kim. Leakage power analysis and reduction for nanoscale circuits. *IEEE Micro*, 26(2):68–80, 2006.

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 Supercomputing'92*, pages 32–41, Minneapolis, MN, USA, 1992. IEEE Computer Society.

K. Ahmed and K. Schuegraf. Transistor wars. *IEEE Spectrum*, 48(11):50–66, 2011.

K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

D. H. Bailey, E. Barszcz, J. T. Barton., D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinksi, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, Albuquerque, NM, USA, 1991. ACM.

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.

M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd international conference on Supercomputing (ICS'09)*, pages 100–109, Yorktown Heights, NY, USA, 2009. ACM.

N. Bell and M. Garland. Implementing sparse matri-vector multiplication on throughput-oriented processors. In *SC'09 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, OR, USA, 2009. ACM.

S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual Symposium on Parallelism in Algorithms and Architectures*, pages 233–244, Calgary, Canada, 2009. ACM.

A. Buluç, 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.

A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. *International Journal of High Performance Computing Applications*, 21(4):467–484, 2007.

A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software*, 34 (4):17–39, 2008.

Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

J. Chang, S.-L. Chen, W. Chen, S. Chiu, R. Faber, R. Ganesan, M. Grgek, V. Lukka, W.-W. Mar, J. Vash, S. Rusu, and K. Zhang. A 45nm 24MB on-die

L3 cache for the 8-core multi-threaded Xeon®processor. In *Symposium on VLSI Circuits*, pages 152–153, Kyoto, Japan, 2009. IEEE.

J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, 2010. ACM.

K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 174–179, Newport, CA, USA, 2004. ACM.

P. Colella. Defining software requirements for scientific computing, 2004. DARPA's High Productivity Computer Systems (pres.).

M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 250–259, Toronto, Ontario, Canada, 2008. ACM.

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.

J. Dongara. Algorithmic and software challenges when moving towards exascale. In *PARA 2012: Workshop on the State-of-the-Art in Scientific and Parallel Computing*, Helsinki, Finland, 2012. Springer. (keynote pres.).

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.

L. F. Escudero. Solving systems of sparse linear equations. *Advances in Engineering Software*, 6(3):141–147, 1984.

J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, 3rd edition, 2001. ISBN 3-54042-074-6.

K. Flautner, K. Nam Sung, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *29th Annual International Symposium on Computer Architecture*, pages 148–157, Anchorage, AL, USA, 2001. IEEE.

A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Copenhagen University College of Engineering, February 2012.

J. A. George. Computer implementation of the finite element method. Technical Report STAN-CS-71-208, Stanford University, Computer Science Department, 1971.

R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. *Parallel Computing*, 27:883–896, 2001.

N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Toulouse, France, 2008. IEEE Computer Society.

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.

Green500. The Green500 List, November 2012. URL http://www.green500.org.

W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In *Proceedings of Parallel CFD*, pages 233–240, Williamsburg, VA, USA, 1999. Elsevier.

D. Guo and W. Gropp. Optimizing sparse data structures for matrix-vector multiply. *High Performance Computing Applications*, 25(1):115–131, 2011.

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

S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 38–43, Portland, OR, USA, 2007. IEEE.

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.

D. Hisamoto, W.-C. Lee, J. Kedzierski, H. Takeuchi, K. Asano, C. Kuo, E. Anderson, T.-J. King, J. Bokor, and C. Hu. FinFET – A self-aligned double-gate MOSFET scalable to 20 nm. *IEEE Transactions on Electron Devices*, 47(12): 2320–2325, 2000.

M. Hoemmen. *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, University of California, Berkeley, 2010.

C. Hu. Thin-body FinFET as scalable low voltage transistor. In *International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA)*, pages 1–4, Hsinchu, Taiwan, 2012. IEEE.

M. Huang, M. Mehalel, R. Arvapalli, and He. S. An energy efficient 32nm 20MB L3 cache for Intel®Xeon®processor E5 family. In *IEEE Custom Integrated Circuits Conference*, pages 1–4, San Jose, CA, USA, 2012. IEEE.

E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, 2000.

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.

E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18:135–158, 2004.

Intel®Xeon®Processor 7400 Series – Datasheet. Intel Corp., October 2008.

Intel®64 and IA-32 Architectures Optimization Reference Manual. Intel Corp., March 2009.

Intel®64 and IA-32 Architectures Software Developer's Manual – Volume 3B: System Programming Guide, Part 2. Intel Corp., September 2010.

C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, USA, 2003. IEEE Computer Society.

ITRS. International Technology Roadmap for Semiconductors, 2001. URL http://www.itrs.net.

K. R. James and W. Riha. Convergence criteria for successive overrelaxation. *SIAM Journal on Numerical Analysis*, 12(2):137–143, 1975.

G. Juckenland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch. BenchIT – performance measurement and comparison for scientific applications. In *Parallel Computing Software Technology, Algorithms, Architectures and Applications*, volume 13 of *Advances in Parallel Computing*, pages 501–508. Elsevier, 2004.

R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30(2):7–15, 2010.

S. Kamil, J. Shalf, and E. Strohmaier. Power efficiency in high performance computing. In *International Symposium on Parallel and Distributed Processing*, pages 1–8, Miami, FL, USA, 2008. IEEE.

V. Karakasis, G. Goumas, and N. Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *12th IEEE International Conference on Computational Science and Engineering*, Vancouver, Canada, 2009a. IEEE Computer Society.

V. Karakasis, G. Goumas, and N. Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, Rome, Italy, 2009b. IEEE Computer Society.

V. Karakasis, G. Goumas, and N. Koziris. Performance models for blocked sparse matrix-vector multiplication kernels. In *International Conference on Parallel Processing*, Vienna, Austria, 2009c. IEEE.

V. Karakasis, G. Goumas, and N. Koziris. Exploring the performance-energy tradeoffs in sparse matrix-vector multiplication. In *Workshop on Emerging Supercomputing Technologies (WEST), ICS'11*, Tucson, AZ, USA, 2011.

G. Karypis and V. Kumar. METIS – Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minessota, Department of Computer Science, 1995.

S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*, volume 3 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool, 2008.

S. Kaxiras, H. Zhigang, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Annual International Symposium on Computer Architecture*, pages 240–251, Göteborg, Sweden, 2001. IEEE.

N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

D. Koufaty and D. T. Marr. Hyperthreading technology in the Netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

K. Kourtis. *Data Compression Techniques for Performance Improvement of Memory-Intensive Applications on Shared Memory Architectures*. PhD thesis, National Technical University of Athens, 2010.

K. Kourtis, G. Goumas, and N. Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, Oregon, USA, 2008a. IEEE Computer Society.

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, G. Goumas, and N. Koziris. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Transactions on Architecture and Code Optimization*, 7(3), 2010.

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. URL http://clang.llvm.org/.

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & 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.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

S. Luke. *Essentials of Metaheuristics*. Lulu, first edition, 2011.

M. Lyly, J. Ruokolainen, and E. Järvinen. ELMER – a finite element solver for multiphysics. In *CSC Report on Scientific Computing*, 1999–2000. URL http://www.csc.fi/english/pages/elmer.

T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Hass, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processsor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, New Orleans, LA, USA, 2010. ACM.

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

J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *International Journal of High Performance Computing Applications*, 18(2):225–236, 2004.

T. C. Oppe and D. R. Kincaid. The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reservoir simulation problems. *Communications in Applied Numerical Methods*, 3(1): 23–29, 1987.

Ch. H. Papadimitriou. The NP-Completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network Based Processing*, pages 66–71. IEEE Computer Society, 2004.

A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR, USA, 1999. ACM.

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

M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 90–95, Rapallo, Italy, 2000. ACM.

S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, 2000.

A. Ramirez. Supercomputing: Past, present, and a possible future. In *International Conference on Embedded Computer Systems (SAMOS)*, page ii, Samos, Greece, 2011. IEEE.

E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21 (1):63–72, 1978.

Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37(155):105–126, 1981.

Y. Saad. *Numerical methods for large eigenvalue problems.* Manchester University Press ND, 1992.

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 Scientific and Statistical Computing*, 7(3):856–869, 1986.

G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 29–40, Cambridge, MA, USA, 2002. IEEE.

L. Sheng, H. A. Jung, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, NY, USA, 2009. IEEE.

K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

T. Skotnicki, J. A. Hutchby, T.-J. King, H.-S. P. Wonk, and F. Boeuf. The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance. *IEEE Circuits and Devices Magazine*, 21(1):16–26, 2005.

V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13 (4):354–356, 1969.

O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing'92*, pages 578–587, Minneapolis, MN, USA, 1992. IEEE Computer Society.

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.

S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. *IBM Journal of Research and Development*, 41:711–725, 1997.

Top500. TOP500 Supercomputer sites, November 2012. URL http://www.top500.org.

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.

H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 1–35, Baltimore, MD, USA, 2002. IEEE Computer Society.

R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(521), 2005.

R. W. Vuduc and H.-J. 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 *Proceedings of the Fourth International Conference on High-Performance Computing*, pages 66–71. IEEE Computer Society, 1997.

J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual International conference on Supercomputing*, pages 307–316, Cairns, QLD, Australia, 2006. ACM.

S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, USA, 2007. ACM.

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.

A. Wm. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architectures News*, 23(1), 1995.

F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 49–62, San Diego, CA, USA, 2003. ACM.

F.-L. Yang, D.-H. Lee, H.-Y. Chen, C.-Y. Chang, S.-D. Liu, C.-C. Huang, T.-X. Chunk, H.-W. Chen, C.-C. Huang, Y.-H. Liu, C.-C. Wu, C.-C. Chen, S.-C. Chen, Y.-Ts. Chen, Y.-H. Chen, C.-J. Chen, B.-W. Chan, P.-F. Hsu, J.-H. Shieh, H.-J. Tao, Y.-C. Yeo, Y. Li, J.-W. Lee, P. Chen, M.-S. Lian, and C. Hu. 5nm-gate nanowire FinFET. In *Symposium on VLSI Technology*, pages 196–197, Honolulu, HI, USA, 2004. IEEE.

S. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 147–157, Monterey, Mexico, 2001. IEEE.

Y.-P. You, C. Lee, and J.-K. Lee. Compilers for leakage power reduction. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):147–164, 2006.

W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction cache leakage optimization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–218, Istanbul, Turkey, 2002. IEEE.

W. Zhang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *Conference and Exhibition of Design, Automation and Test in Europe*, pages 1146–1147, Munich, Germany, 2003. IEEE.

Index

Short biography

Vasileios Karakasis was born on April 5, 1982, in Athens, Greece. In 2000, he entered the National Technical University of Athens (School of Electrical and Computer Engineering), where he obtained his diploma as an Electrical and Computer Engineer in 2005. On December 2006, he entered the post-graduate curriculum of the School of ECE, NTUA, as a Ph.D. student in the field of High Performance Computing at the Computing Systems Laboratory, under the supervision of the Associate Professor, Nectarios Koziris. His thesis focused on the optimization of the sparse matrix-vector multiplication kernel on modern multicore computer architectures and received his Ph.D. degree on December 2012.