ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Data compression techniques for performance improvement of memory-intensive applications on shared memory architectures

Antonios - Kornilios Kourtis

Athens, April 2010

Contents

Abstract

This dissertation deals with data volume reduction techniques that aim to improve the performance of memory-bandwidth starving applications. Although our methods can be successfully applied to serial execution [KGK08], we concentrate on parallel shared-memory architectures where memory bandwidth is scarce, but (pure) computation power plentiful. Our proposed techniques aim at decreasing the non-scalable part of execution time (memory access), at the cost of additional computation overhead. We argue that, since the additional cost is scalable, it will be mitigated as core count increases.

We focus on the application domain of sparse computations. We show that the performance of the sparse matrix-vector multiplication (SpMxV) — an important and ubiquitous scientific kernel — on shared memory systems is restrained by the severe lack of available memory bandwidth.

To decrease memory contention and improve kernel performance we propose two compression schemes: CSR-DU, that targets the reduction of the matrix structural data by applying coarse-grained delta-encoding, and CSR-VI, that targets the reduction of the values using indirect indexing, applicable to matrices with a small number of unique values. Thorough experimental evaluation of the proposed methods and their combination, on two modern shared memory systems, demonstrated that they can significantly improve multithreaded SpMxV performance upon standard and state-of-the-art approaches.

Motivated by the design of CSR-DU, we generalize our approach and propose a storage format, called CSX, that aims at sparse matrix structure exploitation by supporting arbitrary compression schemes. We describe a first implementation, based on delta run-length encoding, that focuses on generality and neatness. Although our work is still under way, initial experimentation shows promising results — especially for matrices that are unable to benefit from CSR-DU.

Ευχαριστίες

Η παρούσα διατριβή εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου. Στο συγκεκριμένο χώρο είχα την ευκαιρία να έρθω σε επαφή με ανθρώπους και ιδέες που είχαν καθοριστικό ρόλο στην εξέλιξή μου ως ερευνητή, αλλά και γενικότερα ως άνθρωπο. Θα ήθελα, επομένως, να εκφράσω τις ευχαριστίες μου σε όλους αυτούς που συνέβαλαν στη δημιουργία ενός γόνιμου κλίματος στο εν λόγω εργαστήριο.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διατριβής, Ν. Κοζύρη για την πολυεπίπεδη στήριξη που μου παρείχε, αλλά και για την εμπιστοσύνη και την πίστη που έδειξε στο πρόσωπό μου. Επιπρόσθετα, θα ήθελα να ευχαριστήσω τα υπόλοιπα μέλη της συμβουλευτικής μου επιτροπής, τους καθηγητές Γ. Παπακωνσταντίνου και Π. Τσανάκα, καθώς και τον καθηγητή Κ. Σαγώνα, η αφοσίωση του οποίου στα μεταπτυχιακά του μαθήματα με βοήθησε να έρθω σε επαφή με νέες ερευνητικές περιοχές.

Θα ήθελα, επίσης, να ευχαριστήσω εμφατικά τον Γ. Γκούμα για την καθοδήγηση, την υπομονή και την προθυμία του κατά τη συνεργασία μας. Η συμβολή του στη διατριβή ήταν καθοριστική.

Ευχαριστώ ιδιαιτέρως τον Γ. Τσουκαλά, κυρίως για την επιμονή του να αναζητά την ουσία, τους Α. Σωτηρόπουλο, Β. Κούκη, Α. Χαζάπη, Γ. Βεριγάκη, Α. Ζίσιμο, Γ. Αναστόπουλο και Β. Καρακάση, καθώς και όλα τα — νέα και παλιά — μέλη του εργαστηρίου για τη δημιουργική συνεργασία και το φιλικό καθημερινό περιβάλλον που μου προσέφεραν.

1 Introduction

In the beginning the Universe was created.
This has made a lot of people very angry
and been widely regarded as a bad move.

Douglas Adams

Moore's law describes a historical trend in processor technology, where the number of transistors that can be placed inexpensively on an integrated circuit doubles every two years. Until recently, microprocessor designers have used the extra available transistors to improve serial performance via frequency scaling and exploitation of instruction-level parallelism (ILP) using techniques such as out-of-order execution, deep pipelines and sophisticated branch prediction. In recent years, however, it has become difficult for this approach to achieve a desirable level of performance improvement due to reasons such as heat and power budget constrains, design complexity, and the reduced inherent ILP in user applications.

Although there were some recent research efforts aimed at boosting serial performance (e.g., [CSC+05]), the industry seems to have decided that this approach is a dead end. Instead, architects turned to processors that incorporate multiple, usually simpler, cores in a single die. The resulting processors are called chip multiprocessors (CMPs) or multicores [ONH+96] and are becoming the norm in microprocessor design [PDG06, Gee05]. Multicore processor are able to remain within power constrains and keep benefiting from Moore's law by using the extra transistors to add more cores. Essentially, instead of trying to exploit ILP, multicore processors aim at the exploitation of a higher level of parallelism: thread-level parallelism (TLP).

This change in processor design has created a noticeable stir in the software world. Until now, application performance was able to benefit from advances in processor design without the need for programmers to modify their software. As multicore designs become the standard, programmers need to adapt by abandoning single-thread programming and incorporate concurrency into their programs [Sut05, OH05, ABD+09]. The expected impact of this microprocessor technology shift in software is illustrated by Olukotun and Hammond in the conclusion of their article [OH05]:

> " [...] *the transition to CMPs is inevitable because past efforts to speed up processor architectures with techniques that do not modify the basic von Neumann computing model, such as pipelining and superscalar issue, are encountering hard limits. As a result, the microprocessor industry is leading the way to multicore architectures; however, the full benefit of these architectures will not be harnessed until the software industry fully embraces parallel programming. The art of multiprocessor programming, currently mastered by only a small minority of programmers, is more complex than programming uniprocessor machines and requires an understanding of new computational principles, algorithms, and programming tools.*"

There are two major aspects of the transition to the multiprocessor programming paradigm that need to be considered: programmability and performance. Parallel programming is generally considered a hard and counter-intuitive task [MGM$^+$09]. Hence, since parallel platforms are becoming ubiquitous, the need for new software practices and tools that make the programmer's life easier emerges [SL05]. For example the transactional memory approach [Gro07, ATKS07] aims at simplifying parallel programming by replacing explicit locking with transactions. Nevertheless, programmability alone is not enough. It is important to ensure that application performance can scale as core count increases.

In this thesis, we tackle performance issues. Our work aims at improving the performance of *memory-intensive* applications — applications whose performance bottleneck is (main) memory bandwidth.* Memory-intensive applications usually have a low ratio of computation operations to memory accesses and they are characterized by poor temporal locality. On multicore systems these applications will frequently perform poorly, even if their parallelization does not create significant overhead. The reason for this is the inability of most systems to deliver the required data transfer rate when all cores simultaneously access main memory. The resulting delays will hurt performance, especially if memory accesses are loads that subsequent instructions depend upon.

We direct our efforts towards sparse computations, an important application domain of scientific computing. Sparse computations are used in several applications (e.g., partial differential equation solvers) and are usually concerned with sparse matrices, i.e. matrices that contain a large number of zeroes. Specifically, we target the performance improvement of the sparse matrix-vector multiplication kernel (SpMxV). This computational kernel, although very simple in its essence, is difficult to optimize and has attracted much attention from researchers due to its importance [AGZ92, TJ92, CA96, Tol97, WS97, PH99, IY99, Im00, GR99, IY01, VDY$^+$02, Vud03, MCG04, PHCR04, PHCR05, BELF07, VM05, KHK$^+$05, WL06, WOV$^+$09, Wil08, BBR09, KGK09b, KGK09a].

1.1 Contribution

This work explores the use of compression, i.e. data volume reduction techniques, to improve the execution time of memory-intensive applications. The main challenge in this endeavor, as well

*We will use the term "*memory-intensive*" throughout this text to refer to applications whose main performance impediment is memory bandwidth. Note, however, that in different contexts, this term may have a different meaning.

as what differentiates our approach from typical compression schemes, is that size reduction is not adequate to ensure success; we aim to improve performance, i.e., reduce execution time. In other words, any possible computational overhead (e.g., decompression cost) should be amortized before a method is deemed successful.

The contribution of this dissertation is summarized below:

- We investigate compression as a means to improve performance of memory-intensive applications by alleviating contention on the memory subsystem. We outline the requirements for such an optimization to be successful and we examine its relevancy on multicore systems.
- We study the performance behavior of the SpMxV kernel arguing that: (a) its performance is restrained by limited memory bandwidth and that (b) it is a good candidate for applying compression schemes to improve its scalability.
- We propose the CSR-DU sparse matrix storage format. CSR-DU compresses matrix structural data by applying coarse-grained delta encoding and exploiting contiguous elements.
- Based on the observation that several sparse matrices contain a significant number of common numerical values we propose CSR-VI: a specialized storage format that employs indirect indexing to compress matrix values. Since CSR-DU operates on structural data, while CSR-VI operates on numerical values, we also consider a combination of these two formats called CSR-DUVI.
- Finally, we attempt a first step towards a storage format that can support arbitrary compression schemes. We call this format CSX and discuss an initial implementation for compression schemes based on delta run-length encoding on multiple directions.

1.2 Outline

The thesis is organized as follows:

Chapter 2 builds the case for using compression to improve the performance of multithreaded memory-intensive applications. First, we provide a brief overview of shared memory systems and show the scalability problems of applications with large memory bandwidth requirements. Next, we describe the proposed compression approach and discuss necessary conditions for its success. The chapter ends with a case study of our method using bitwise operations on bitmap indices.

Chapter 3 provides an introduction to sparse computations. We present several well-established sparse matrix storage formats, as well as their corresponding SpMxV implementations.

Chapter 4 is concerned with the performance of the SpMxV kernel on multicore systems. First, we discuss the kernel's implementation and analyze its performance characteristics. Next, we introduce our experimental setup and present the results of a performance evaluation which shows that SpMxV's performance is restrained by limited memory bandwidth.

Chapter 5 presents the CSR-DU storage format — our approach for compressing the structural data of a sparse matrix using delta encoding. It starts with a discussion of our motivation and continues with the definition of the format. Several aspects of the format are shown, including an extension for exploiting sequential elements. An experimental evaluation is also preformed where it is established that CSR-DU can provide significant performance benefits for multithreaded SpMxV.

Chapter 6 presents the CSR-VI storage format which applies compression to the numerical values of matrices. Initially, we discuss the motivation behind our approach and, since not all matrices are suitable for value compression, the conditions under which CSR-VI can be beneficial. An experimental evaluation shows significant performance benefits for both CSR-VI and its combination with CSR-DU (CSR-DUVI).

Chapter 7 discusses our initial approach towards a unified storage format called CSX. CSX utilizes matrix-specific SpMxV routines and aims to support arbitrary compression schemes. We discusses several classes of structural patterns and we present a general, yet relatively expensive, approach for substructure detection. The chapter is concluded with a performance evaluation.

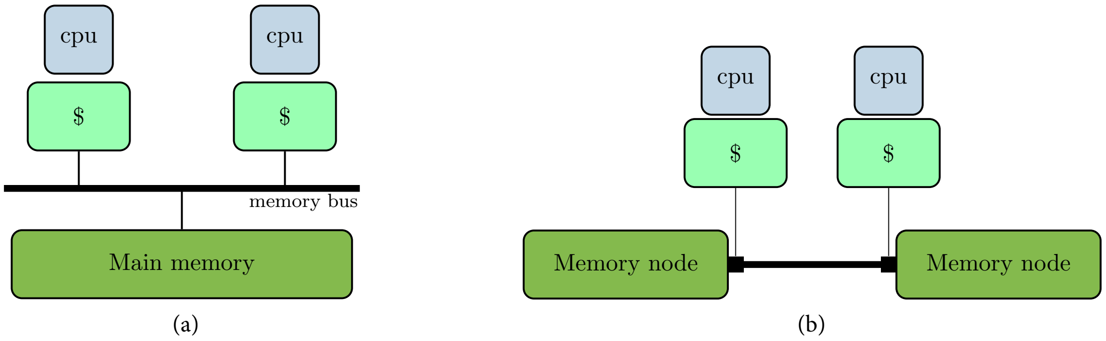
Finally, Chapter 8 summarizes conclusions and briefly discusses future work directions.

Background and key ideas

2

2.1 Shared memory systems

Shared memory systems [CS99, HP07] are a family of parallel architectures where multiple processors operate on the same main memory. Until a few years ago, shared memory systems were implemented, almost exclusively, via symmetric multiprocessing (SMP). SMP systems comprise of two or more identical processors that connect to a single main memory, usually via a bus or similar interconnect (see Figure 2.1a). As in all shared memory systems, a cache coherence protocol is responsible of maintaining data integrity between processor caches. The centralized memory and the memory bus constitute the main performance bottleneck of SMP systems because requests from different processors need to be serialized.



(a)

(b)

Figure 2.1: Typical shared memory systems. (a) an SMP system, (b) a NUMA system.

Hence, as the number of processing elements that share the memory increases, CPU designers turn to more scalable designs like Non-Uniform Memory Access (NUMA) architectures (see Figure 2.1b), where the memory is distributed among different nodes that are connected via a scalable interconnect (e.g., interconnects based on point to point links). In principle, each NUMA node is *local* to a set of CPUs, and access to memory in this node is faster than access to *remote* nodes. Obviously, this architecture mitigates the memory bandwidth bottleneck, since it allows different CPUs to operate on different NUMA nodes. In general, these systems fall in the category of shared memory systems since they provide a coherent unified view of memory for the programs, and is up to the programmer or the operating system to distribute data in different nodes to maximize performance.

The previous paragraph discussed shared memory configurations for separate processor chips. These chips however can, and probably will given the recent multicore trend, implement CMP, i.e., contain multiple processing cores that share a part of the cache hierarchy. For example, Figure 2.2 illustrates a quad-core processor. Each core in this example has its own private L1 cache, and two pairs of cores share two L2 caches. The chip may connect directly to the main memory interconnection network, or via an off-chip cache that is shared between all processor cores. Cache sharing is an important factor of the system's performance and can be either constructive or destructive, depending on the application and on whether threads scheduled on the cores that share a cache operate on common data or not. Evidently, multicore processors intensify the performance problem of concurrent accesses to main memory.

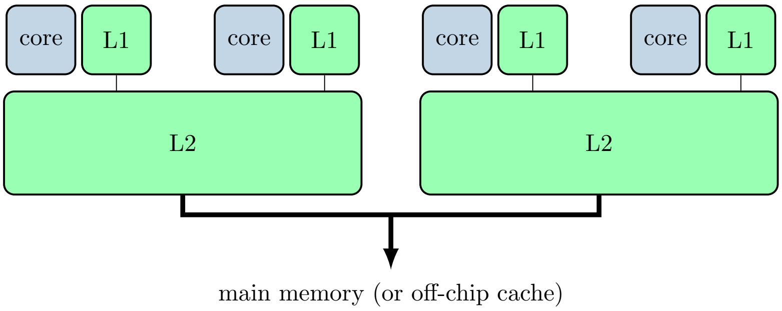


Figure 2.2: An example of a multicore processor with 4 cores. Each core has it's own L1 cache and there are two L2 caches shared by two cores

2.2 Application scaling on shared memory systems

As it is illustrated by Amdhal's law the possible speedup of an application in a parallel architecture is limited by the sequential fractions of the program. It is possible, however, that a shared memory architecture results in serialization of program fractions that are, from the perspective of the programmer, parallel. For example, in an SMP system that interconnects the various processors via a bus, requests to main memory will be serialized. Hence, even if a program does not contain any explicit serial fractions, it is possible that it would not scale linearly — as expected — in a shared memory system. In the following paragraphs we discuss the implicit scalability problem that arises in shared memory architectures, even for fully parallelizable applications without data dependencies.

The scalability behavior of a parallel application in a shared memory environment depends on its data access pattern. Applications with no data dependencies and good temporal locality scale well, since each core can work independently using local data residing in its cache, without interfering with the operation of other cores. An example of a computational kernel with these characteristics is matrix multiplication (MxM). The straightforward implementation of matrix

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Listing 2.1: Matrix multiplication for $N \times N$ matrices

multiplication ($C = AB$) for square matrices $N \times N$ matrices is presented in Listing 2.1.

Matrix multiplication performs $\mathcal{O}(N^3)$ operations on $\mathcal{O}(N^2)$ data and is generally considered a cache-friendly algorithm due to its spatial locality. A blocking transformation is usually applied (see A.1) to ensure that the data will be reused before they are evicted from the cache. Moreover, matrix multiplication can be performed in a completely parallel fashion without data dependencies between execution threads. As a result, MxM generally exhibits very good scalability. Figure 2.3, presents scalability results for MxM on two different multicore systems. The first system consists of two quad-core processors (8 cores total), while the second consists of four 6-core processors (24 cores total). A more detailed description of the systems is provided in 7.5.1.

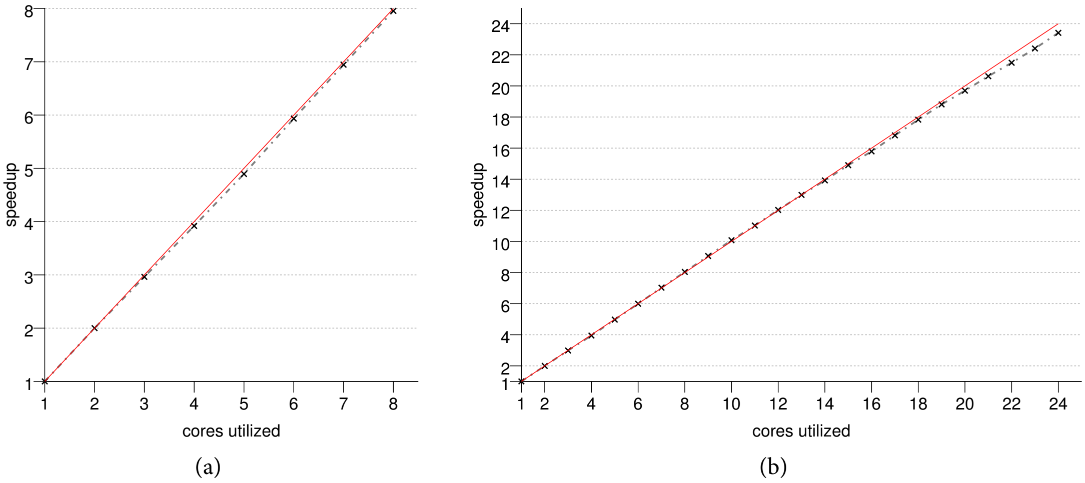


Figure 2.3: Matrix multiplication speedup for two systems. (a) a two-way quad-core system (8 cores in total) and (b) a four-way 6-core system (24 cores in total).The size of the matrices is 2048×2048 single-precision floating-point elements. The optimal (linear) speedup is illustrated with a red line.

Nevertheless, not all parallel applications can scale as well as the matrix multiplication kernel in a shared memory system. Applications with frequent accesses on main memory tend to exhibit poor scaling due to contention on the memory subsystem (e.g., applications with streaming access patterns and limited spatial locality). To illustrate the congestion on the memory subsystem and quantify the resulting performance bottleneck, we developed a benchmark to measure maximum throughput when different threads read data from main memory. The benchmark allocates and initializes large memory areas and subsequently performs read operations using streaming instructions (see A.2 for a detailed description of the benchmark and 4.5.2 for additional experimental results).

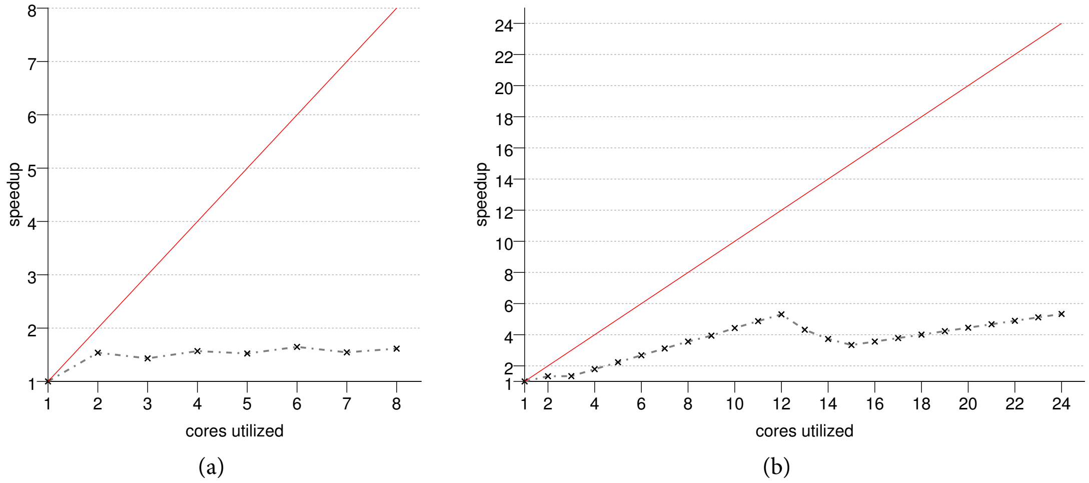


Figure 2.4: Speedup for aggregate read memory throughput on two systems. (a) a two-way quad-core system (8 cores in total) and (b) a four-way 6-core system (24 cores in total).The optimal (linear) speedup is illustrated with a red line.

Figure 2.4 shows the speedup for aggregate read memory throughput on two multicore systems. Evidently, the available memory bandwidth on both machines is not adequate to allow this benchmark to scale. Even though this workload is artificial, it serves as an illustration for the scaling limitations of memory-intensive applications.

In a typical shared memory architecture we can assume that the execution time of each parallel part of the program can be split in two parts: a scalable (computation and access to private caches) and a non-scalable (access to shared memory). Under these assumptions we can express the execution time with the following equation:

$$t = \frac{r_c \cdot d}{n} + bw_n \cdot d \tag{2.1}$$

Where:

n is the number of utilized threads.

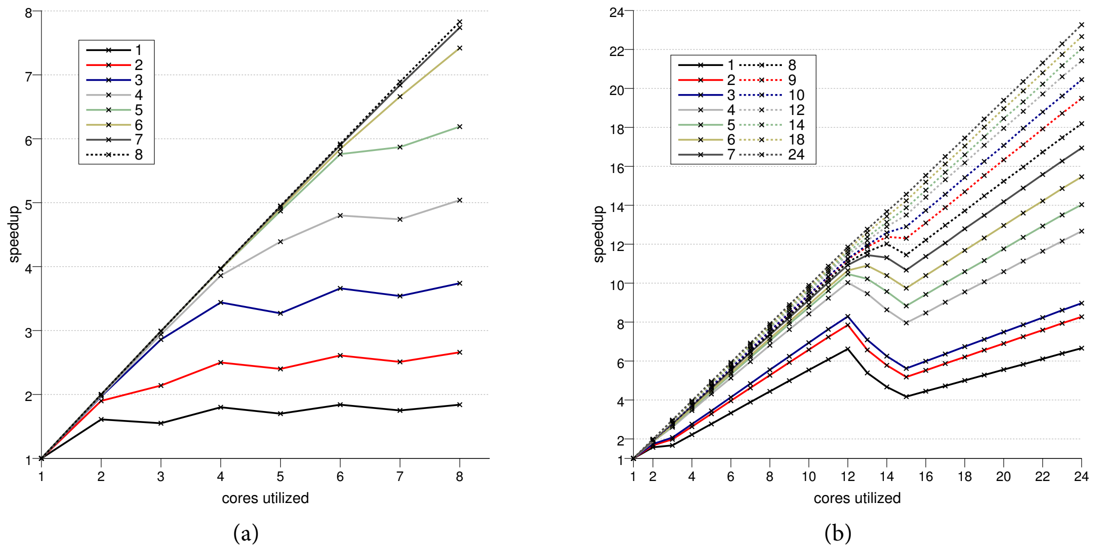
d is the size of the data that need to be fetched from main memory. It depends on the program accesses (e.g., temporal locality) and parameters of the cache hierarchy (cache size, cache-line size, associativity, etc.)

r_c is the time cost of the scalable computations per byte of data fetched from main memory. It depends on the program's operations and CPU speed.

bw_n is the available memory bandwidth when n threads are utilized. It depends on the program accesses (e.g., its spatial locality) and on the capabilities of the hardware, given the topology of the utilized cores (hardware prefetching, interconnection network, etc.).

Modern processors and memory hierarchies are very complex systems. It is, therefore, difficult to model the parameters of Equation 2.1 to predict the actual performance of real-world machines, especially since modern processors can execute instructions out of order and in a speculative way. Hence, to investigate the effect of these two parts on program execution for real-world systems, we developed a micro-benchmark called *memcomp* (see also A.3). Memcomp executes loops that perform a memory load and a variable number of computational operations. The program loads an element from an array stored in main memory and c additions of this element to a register.

We perform experiments using double-precision floating-point elements and an unrolled loop that performs 64 loads (and $64 \cdot c$ additions) at each iteration. The results are presented in Figure 2.5. Different graph lines represent different values of c. As expected, the scalability when performing a single addition ($c = 1$) resembles that of the memory throughput benchmark. As the number of additions are increased, computation becomes dominant in the execution time and the benchmark gradually achieves scalable performance. The point where this change occurs is highly dependent on the architecture.



(a)

(b)

Figure 2.5: Memcomp benchmark speedup. Different lines represent different values of the c parameter. We consider two systems: (a) A two-way quad-core system (8 cores in total) and (b) A four-way 6-core system (24 cores in total).

There are systems, however, that are incompatible with our previous assumptions according to which computations scale and memory accesses do not. For example, processors that implement

TLP in a single core via technologies such as fine-grain multi-threading or simultaneous multi-threading (SMT) [TEL95] are, generally, unable to scale for highly-optimized codes [AAKK06, AAKK08].

Additionally, it is possible to construct a shared memory system with large enough available bandwidth, such that all available threads can operate without contention on the memory subsystem. An example is the Niagara 2 processor [SBB$^+$07] which consists of 8 cores, each of which supports 8 threads, for a total of 64 threads per CPU. Niagara 2 deviates from mainstream multicore chip designs, since it provides a large number of available threads and good memory performance, at the expense of single core computing power. Figure 2.6 presents scalability results from the memcomp benchmark, where it is illustrated that as the computation ratio increases, the scalability of the benchmark is reduced. An additional comment to be made, is that the absolute single-threaded performance is vastly inferior to that of the mainstream chips examined previously.

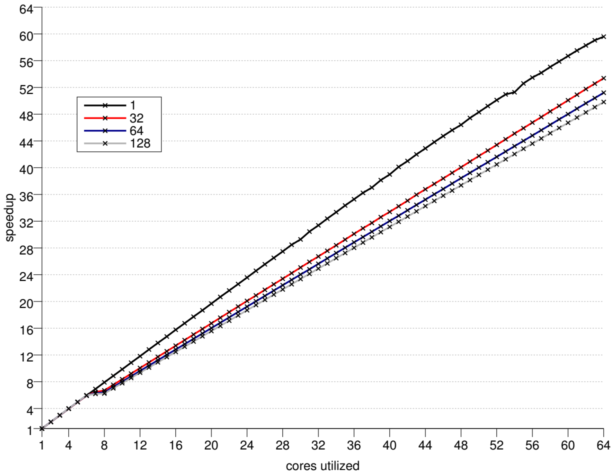


Figure 2.6: Memcomp benchmark speedup on the Niagara 2 processor. The benchmark performs a load operation of a double-precision floating-point value and c additions of this value to another register. The different lines represent different values of c.

In the next paragraphs we focus on typical multicore architectures and investigate the idea of improving the execution time of memory-intensive applications by applying data compression. With regard to Equation 2.1, we aim at decreasing the non-scalable part of the execution time (memory access), at the cost of inducing additional computational overhead. Since the additional cost is scalable, we argue that, it will be mitigated as the number of cores increases.

2.3 Compression for optimizing multithreaded applications

Compression can be viewed as a trade of data volume for computation: it results in reduced data volume at the cost of additional computational overhead. Traditionally compression is used for reducing the time of data network transfers and provide efficient persistent storage in terms of space requirements. We investigate the use of compression for improving the execution time of multithreaded applications in shared memory architectures. Hence, based on Equation 2.1, if we apply a compression scheme that reduces the data by a factor of a and results in an additional computational cost by a factor of b, then the expression for the execution time t' will become:

$$t' = \left(\frac{b \cdot r_c}{n} + \frac{bw_n}{a} \right) \cdot d, \quad a, b \geq 1 \tag{2.2}$$

Not all multithreaded applications are suitable for applying compression as a means for improving execution time. In fact, this technique can be applied only to certain types of applications. Next, we discuss the conditions that need to be met by an application, so that it can qualify for the proposed techniques

(a) *Memory bandwidth bottleneck*. First, the application should be memory intensive — i.e., its performance should be dominated by frequent transfers from (main) memory. If an application is compute-bound, then its execution time is dominated by the r_c term in Equation 2.1. Compression will increase this term by a factor of b, which will lead to a performance hit. The latter is especially true for modern processors which are able to overlap computations with memory transfers and hide the memory access latency. Applications that normally adhere to this requirement are applications that exhibit poor temporal locality.

(b) *Compressible data*. An important requirement is the need for application data to be compressible. If data are random, then factor a of Equation 2.2 will be close to 1, and compression will probably result in performance slowdown. It is expected, however, that fulfilment of this condition would not be a problem for most real-world applications where data usually express specific semantics and contain redundancies that can be exploited towards compression. Nevertheless, since an application can be used in different domains and thus operate on different types of data, it is not always straightforward to determine a suitable compression technique.

(c) *Decompression cost mitigation*. Additionally, it is required that the decompression run-time cost is mitigated by the benefits of data volume reduction. This requirement is an important differentiation from typical compression schemes, where the most important consideration is the compression ratio. With regard to Equation 2.2, the benefits of factor a (compression ratio) should outweigh the losses of factor b (decompression overhead).

    As the number of cores increases, the effect of the decompression overhead on execution time decreases (assuming that it falls into the scalable execution part). This can be demonstrated in Equation 2.2 where $n \to \infty$ leads to an execution time of $(bw_n/a) \cdot d$. Nevertheless, real-world machines contain a finite number of processing cores and, as a result, the decompression overhead cannot be ignored. This condition is strongly associated with the specific hardware implementation, since the actual decompression overhead and the benefit from data reduction depend largely on the costs of the various hardware operations.

(d) *Compression cost mitigation*. Besides the decompression cost, however, we need to additionally consider the compression overhead, which also needs to be mitigated. The compression overhead can be characterized as "hidden", since we excluded it from the execution time expression because we assume that it can be performed off-line. For a number of applications this assumption is correct. A typical example are applications that perform a large number of computations using the same data (e.g., by performing a large number of iterations). On the other hand, applications that use the data only once do not satisfy the above condition and will not benefit from compression in terms of execution time. An example for an application that does not satisfy this requirement is data compression itself.

The latter condition seems contradictory with the characterization of applications with no temporal locality as a good example for this technique. However, because caches are not infinite, applications that operate on large amounts of data retain their streaming nature, even if they are executed iteratively.

2.4 Case study: bitmap indices

Up until now we have limited our discussion in synthetic benchmarks (e.g., memcomp). The issue that naturally arises is whether there exist real-world applications that satisfy the conditions mentioned above and how they can benefit from compression. To elaborate on this issue, we briefly investigate the applicability of our method using a real-world application: bitmap indices. Although this case study is by no means comprehensive, it shows potential execution time benefits for compression techniques when the number of utilized cores increases.

Indices are data structures used extensively in database systems [UGMW01] and aim at improving the speed of information retrieval operations. Although typically these structures are implemented using B-trees, the alternative of bitmap indexing is gaining popularity in modern database systems, especially for read-mostly environments (e.g., data warehouses). The simplest form of a bitmap index on an attribute is a number of bit vectors —one per attribute value— each of which represent the set of records that adhere to the specific attribute value. This is called a *Value-List index* [OQ97], and an example is given in Table 2.1.

RID	X	bitmap index			
(record id)	(attribute)	X = 0	X = 1	X = 2	X = 3
1	1	0	1	0	0
2	0	1	0	0	0
3	2	0	0	1	0
4	1	0	1	0	0
5	3	0	0	0	1
6	3	0	0	0	1
7	0	1	0	0	0
8	1	0	1	0	0
		b_0	b_1	b_2	b_3

Table 2.1: Example of bitmap indices for an attribute (X) that assumes four values (0–3). The bitmap index for each of these values (b_{0-3}) appears as a column.

Information retrieval queries using bitmap indices are implemented using bitwise logical operations. For example, for the data in Table 2.1 a query for $X > 0$, would be answered by performing $b_1 \, OR \, b_2 \, OR \, b_3$. We argue that this application is well-suited for the optimization described previously: (a) it exhibits a low computation to memory accesses ratio and it usually involves large data sets that do not fit into the cache, (b) the bit vectors are typically compressible, and (c) the compression cost can be mitigated when operating on read-mostly environments. The remaining requirement for the compression approach to be gainful is the decompression cost mitigation, which is strongly dependent on the compression method used.

The compression of bitmap indices has been extensively studied in related literature [Joh99, WOS06]. Although not directly motivated by the need to reduce the memory bottleneck, these compression methods can be used in our evaluation. To investigate the potential benefits of the compression tradeoff in terms of multithreaded performance we performed a number of experiments using different compression approaches.

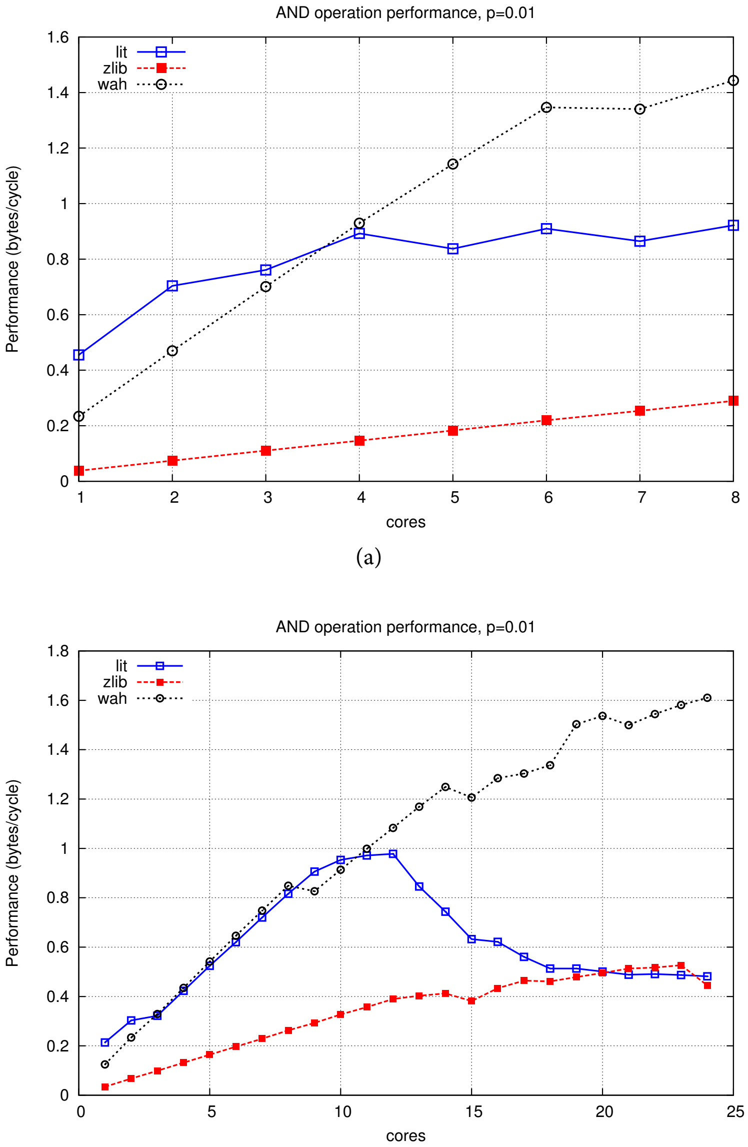
Figure 2.7 demonstrates the performance of an AND operation on two multicore machines (8 and 24 cores in total) for three different compression schemes: *lit*, which uses uncompressed (literal) bitmaps, *zlib*, which uses the zlib compression scheme [DG96, Deu96] and *WAH*, which uses the WAH (Word-Aligned Hybrid) compression scheme [WOS06]. The results were obtained using random-generated bitmaps with a bit density (probability of a bit being '1') of 0.01*. The performance of the uncompressed bitmaps (*lit*) achieves the best serial performance on both machines, but it is not able to scale as more processing cores are utilized. Moreover, a performance degradation is observed after 12 cores in the 24-core system, which can be attributed to contention on the main memory. The *zlib* compression scheme has significant decompression overhead and although it is able to scale better than *lit*, its performance remains considerably low even for a large number of cores. On the other hand, the *wah* scheme has low decompression overhead and is able to achieve better performance than *lit* when a sufficient number of cores is used.

In conclusion our evaluation indicates that — in the context of multicore systems — use of compression can act in benefit for performance of real-world applications, even if it degrades performance in the serial case.

2.5 Conclusions

In this chapter we presented the key idea of our work — using compression for improving the performance of memory-intensive applications on shared memory systems. In the remaining of this thesis we apply our ideas to the domain of sparse-matrix computations and specifically to the SpMxV kernel. The next chapter provides an introduction to these concepts.

*Smaller values would lead to insignificant size reduction (or even increase), while larger values would lead to very fast optimized AND operations for the WAH compression scheme



(a)

(b)

Figure 2.7: bitwise AND operation performance. (a) a two-way quad-core system (8 cores in total) and (b) a four-way 6-core system (24 cores in total).

3 Sparse Matrices and Sparse Matrix-Vector Multiplication

3.1 Sparse Matrices

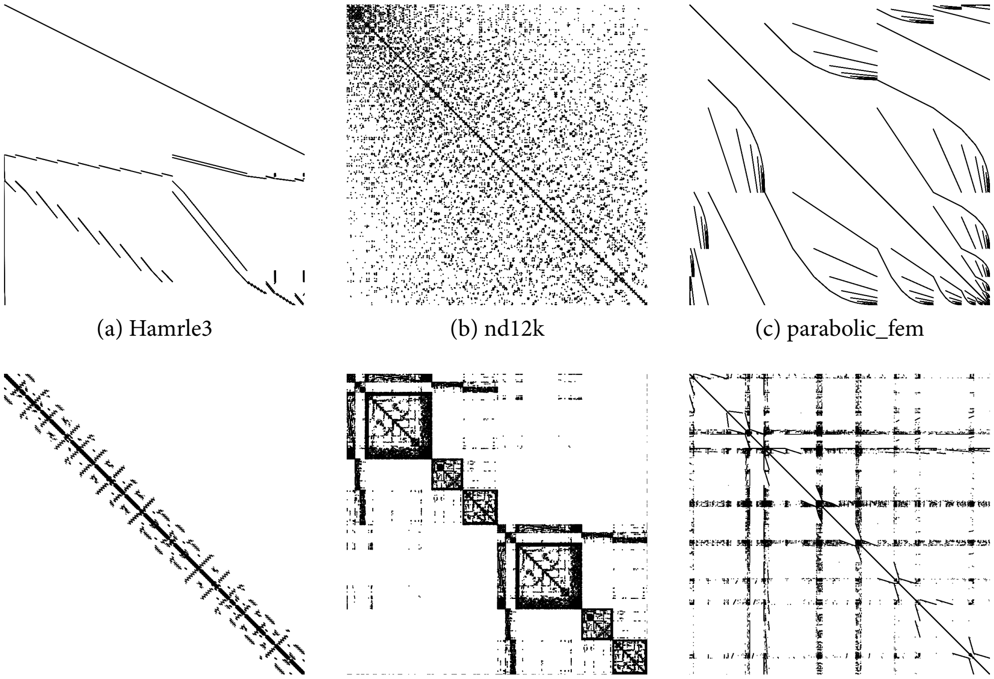
Sparse matrices are typically defined as matrices whose values are dominated by zeroes. However, the characterization of a matrix as sparse is usually not performed on the basis of a qualitative criterion (e.g. the percentage of zero elements). Instead, a matrix is treated as sparse based on the potential benefits that arise from such a treatment: "[...]*a matrix can be termed sparse whenever special techniques can be utilized to take advantage of the large number of zero elements and their locations*" [Saa03]. Based on the above definition we can derive a sufficient requirement regarding the sparsity of a matrix: an $N \times M$ matrix is sparse if the number of its non-zero elements is orders of magnitude smaller than $N \cdot M$. Examples of sparse matrices that correspond to real applications (taken from [Dav97]) are illustrated in Figure 3.1.

Sparse matrices are met in various scientific and engineering fields, and they generally arise when studying systems that are loosely coupled. Large sparse matrices typically appear during the discretization process when solving partial differential equations (PDE) [Saa03]. More specifically, the typical way of solving PDEs is to perform discretization employing techniques such as the Finite Element Method (FEM), which usually results in problems with large sparse matrices.

Additionally, sparse matrices are used in the representation of large graphs using an adjacency matrices. An example of such a graph is the World-Wide Web [KKR$^+$99], where each vertex is a page, and a directed edge from vertex A to vertex B ($A \to B$) represents the existence of an URL link in page A linking to page B. Generally, matrix sparsity and graph theory are subjects that are closely linked: graph algorithms are employed for sparse matrices (e.g., for partitioning [HK99, VB05]), while graph algorithms can be expressed via sparse matrix computations [KCA09].

3.2 The sparse matrix-vector multiplication operation

An important and ubiquitous operation for sparse matrices is the sparse matrix-vector multiplication (SpMxV), where a $N \times M$ sparse matrix is multiplied with a dense vector (of size M) resulting in another dense vector (of size N): $y = A \cdot x$. We refer to y as the *destination vector* and to x as the *source vector*. The general expression for the elements of the y vector is:


(a) Hamrle3

(b) nd12k

(c) parabolic_fem

(d) stomach

(e) inline_1

(f) ASIC_680k

Figure 3.1: Examples of sparse matrices from real-world applications. (source: [Dav97])

$$y_i = \sum_{j=1}^{M} A_{ij} \cdot x_j \quad 1 \leq i \leq N$$

It is evident that zero elements do not contribute on the result and can be omitted. Hence, the sparsity of a matrix can be exploited when performing the SpMxV operation by considering only non-zero elements ($A_{ij} \neq 0$) during the computation.

The SpMxV operation is used in a large variety of applications in scientific computing and engineering. It is the basic operation of iterative solvers, such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES), extensively used to solve sparse linear systems resulting from the simulation of physical processes described by PDEs [Saa03]. Moreover, a number of graph algorithms can be expressed using adjacency matrix multiplication, and many of them perform several iterations, where the iteration time is dominated by SpMxV [KCA09]. Examples include link analysis algorithms such as PageRank [BP98]. Finally, SpMxV has been reported as a member of one of the "seven dwarfs", which are classes of applications that are believed to be important for at least the next decade [ABC$^+$06].

3.3 Sparse storage formats

The rest of this chapter is concerned with an important aspect of sparse matrices — sparse storage formats. These formats are data structures that enable efficient storage and efficient operations for sparse matrices. We start by discussing dense storage.

Typical schemes of dense matrices store the elements subsequently into memory. The space used is equal to the total number of elements. In addition, a location function is used to determine the position of a matrix element in the linear memory space. The most frequently used schemes are row-major (C) and column-major (Fortran) order. For an $N \times M$ matrix, a row-major order scheme stores the element a_{ij} into the $j + (i \cdot M)$ location, while for the column-major order scheme the same element is stored into the $i + (j \cdot N)$ location.

Sparse matrices use special storage representations to exploit the large number of zero values. These storage schemes are generally build around the concept of storing only the non-zero values of the matrix. This results in less storage size requirements, as well as more efficient operations (e.g. SpMxV) since zero values usually do not contribute to the computation. Nevertheless, additional information about the position of the non-zero values is required. Hence, we separate the sparse matrix data into two categories: *index data*: data that are used for the representation of the matrix structure and *value data*: data that represent the numerical values of the matrix.

In the following paragraphs we discuss several existing sparse-matrix formats. We focus on matrices suitable for the SpMxV operation, where the matrix remains constant and random access is not required.

3.3.1 Coordinate format

The *coordinate (COO) format* is one of the simplest forms of sparse storage. It stores the non-zero elements along with their corresponding indices — their matrix location. For instance, the COO format for a vector is called *compressed sparse vector* or simply *sparse vector*. In a sparse vector format the non-zeroes are stored contiguously in an array `val` and the indices of these elements are stored in another array `ind`. In other words, `val`[i] stores the element in position `ind`[i]. An example of a sparse vector is illustrated in Figure 3.2a.

Similarly, for a two-dimensional matrix two index arrays are needed: one for the row (`row_ind`), and one for the column (`col_ind`) of each non-zero element. Hence, the i-th non-zero element has a value of `val`[i] and its coordinates are: (`row_ind`[i],`col_ind`[i]). Figure 3.2b presents an example for the storage of a two-dimensional matrix in the COO format. Each of the index arrays (`row_ind`,`col_ind`) have a size equal to the number of the non-zero elements.

An issue that arises is the order in which the non-zero elements are stored. The COO format does not impose a restriction on the storage of the elements. It is, however, common practice to assume a specific order that bestows beneficial properties (e.g., good spatial locality) on algorithm implementations. Typically, a lexicographical order on the coordinates is used. In this case, elements of sparse vectors are stored in an increasing index order. For the ordering of two-dimensional matrices, the row coordinate is considered first.

$$V = \left[\begin{array}{cccccc} 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \end{array}\right]$$

$$\begin{array}{rl} \texttt{val}: & (\ 6.3\ 7.7\ 8.8\ ) \\ \texttt{ind}: & (\ \ 1 \quad 3 \quad 5\ \ ) \end{array}$$

(a)

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}$$

$$\begin{array}{rl} \texttt{row\_ind}: & (\ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 3\ \ 4\ \ 4\ \ 4\ \ 5\ \ 5\ \ 5\ \ 5\ ) \\ \texttt{col\_ind}: & (\ 0\ \ 1\ \ 1\ \ 3\ \ 5\ \ 2\ \ 2\ \ 4\ \ 5\ \ 0\ \ 3\ \ 4\ \ 0\ \ 2\ \ 3\ \ 5\ ) \\ \texttt{values}: & (\ 5.4\ 1.1\ 6.3\ 7.7\ 8.8\ 1.1\ 2.9\ 3.7\ 2.9\ 9.0\ 1.1\ 4.5\ 1.1\ 2.9\ 3.7\ 1.1\ ) \end{array}$$

(b)

Figure 3.2: Examples of the coordinate format (COO). (a) a vector, (b) a two-dimensional matrix. Elements are stored in a lexicographical order

If the majority of the matrix rows do not contain a small number of elements and a lexicographical order is used, the row_ind array will contain redundant information, as can be seen in Figure 3.2b. The CSR format, described in the next paragraph, exploits this redundancy to reduce the storage requirements of the sparse matrix.

3.3.2 CSR storage format

One of the most popular storage representations for sparse matrices is the compressed storage row format (*CSR*) [BBC+94, Saa03]. CSR stores the sparse matrix as a number of sparse vectors (one for each row) and allows random access to entire rows. More specifically, the matrix is stored in three arrays: values, row_ptr and col_ind. The values array stores the non-zero elements of the matrix in row-major order, while the other two arrays store indexing information: row_ptr contains the location of the first (non-zero) element of each row within the values array and col_ind contains the column number for each non-zero element. An example of the CSR format for a 6×6 sparse matrix is presented in Figure 3.3.

The size of the values and col_ind arrays is equal to the number of non-zero elements (*nnz*), while the row_ptr array size is equal to the number of rows (*nrows*) plus one. The CSR format is considered a good default choice for the SpMxV kernel [Vud03]. Its implementation for a matrix with N rows is illustrated in Listing 3.1. The CSR SpMxV kernel consists of two loops: the outer loop iterates over all rows of the matrix using the row_ptr array, while the inner loop computes a single element of the destination vector. To assist the optimization process of the compiler the code can be optimized to write the y[i] value at the end of the inner loop, by keeping the intermediate result in a temporary variable that can be allocated in a register (see Listing 4.1).

```
for (i=0; i<nrows; i++)
        for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
                y[i] += values[j]*x[col_ind[j]];
```

Listing 3.1: CSR SpMxV implementation.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}$$

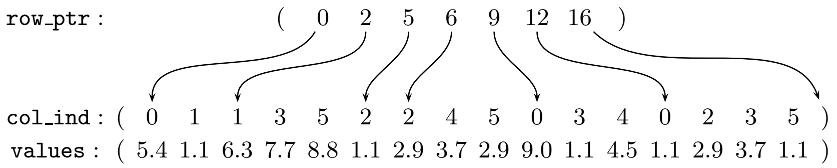

row_ptr : (0 2 5 6 9 12 16)
col_ind : (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5)
values : (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)

Figure 3.3: Example of the CSR storage format

The CSR format can be considered a special case of what is referred as *compressed stripe storage* [Vud03]. Another straightforward case of this class of formats is the compressed storage column (*CSC*) format, which is similar to CSR, except that it uses columns instead of rows, i.e., it allows random access to columns, which stores as sparse vectors. Another possible implementation would be to store the matrix diagonals as sparse vectors.

3.3.3 Blocking formats

Over the years, a number of different storage formats aiming at the exploitation of specific matrix structure for providing an efficient SpMxV operation have been proposed. One of the most successful formats in that respect is the block compressed storage row (*BCSR*) [IY01]. The BCSR format can be viewed as a generalization of CSR, where unit of operation is two-dimensional blocks ($r \times c$), instead of individual elements. As CSR does not store zero elements, BCSR does not store blocks that contain only zeroes. Thus, instead of storing the column index for each non-zero element, BCSR stores per-block column indices. Moreover, BCSR keeps pointers to *block-rows* (i.e. rows of blocks), instead of rows of elements. Obviously, The case of $r = c = 1$ is equivalent to CSR.

Similarly to CSR, BCSR uses three arrays for the representation of a sparse matrix: (a) `bval`, which stores the values for all blocks of the matrix in column- or row-major order — i.e., the j value of block i, is stored in location $(i \cdot r \cdot c) + j$ of the `bval` array, (b) `bcol_ind`, that stores the block-column indices and (c) `brow_ptr`, which stores pointers to the first element of each block row. Hence, assuming that a sparse matrix consists of *nblocks* blocks, the size of the `bval` array is $r{\cdot}c{\cdot}$*nblocks*, the size of the `brow_ptr` array is the number of block rows (*nbrows*) plus 1: $\lceil \frac{nrows}{r} \rceil + 1$, and the size of the `bcol_ind` array is *nblocks*.

An example of the BCSR format is presented in Figure 3.4, where an 8×8 matrix is divided into 2×2 blocks. As it shown in the figure, it is possible for the `bval` array to contain zeroes to account for zeroes contained in blocks. This procedure is known as *padding* and it may result in inefficiencies depending on the block shape ($r \times c$) and the matrix structure.

The BCSR format shown in Figure 3.4 requires that aligned blocks at r row and c column

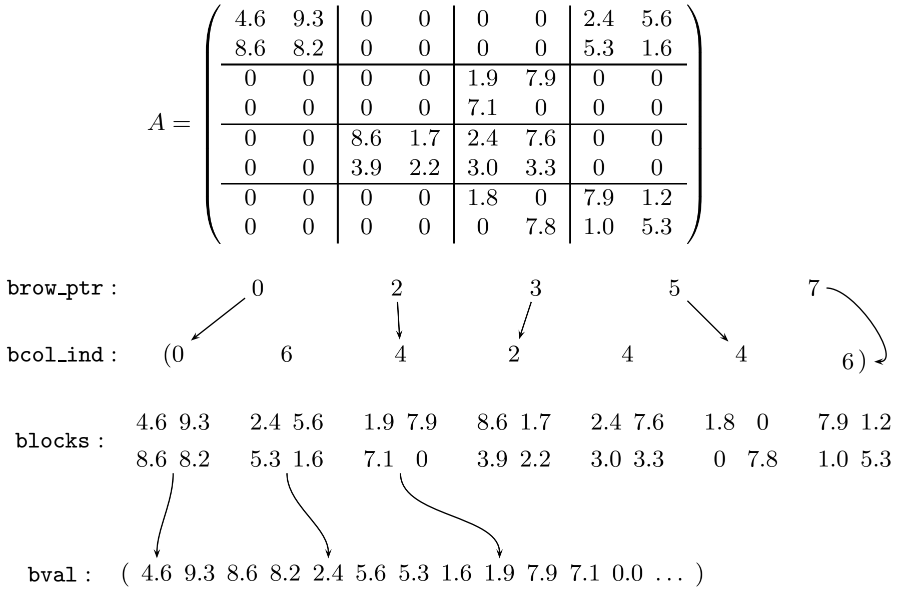


Figure 3.4: Example of the BCSR storage format

boundaries, i.e. that each $r \times c$ blocks starts at a position (i, j) such that: $i \bmod r = 0$ and $j \bmod c = 0$ This approach provides the benefits of simpler construction of the BCSR format and allows for easy vectorization, which can result in a positive performance impact [KGK09b]. A variation of the BCSR format that aims at reducing the necessary padding by relaxing the above constraints is the unaligned BSCR (*UBCSR*) [VM05].

```
for (i=0; i < nbrows; i++)
    for (j=brow_ptr[i]; j < brow_ptr[i+i]; j++)
        for (ir=0; ir < r; ir++) // rxc block multpilication
            for (ic=0; ic < c; ic++){
                y_idx = (i*nrows) + ir;
                v_idx = (j*r*c) + (ir*c) + ic;
                x_idx = bcol_ind[j] + ic;
                y[y_idx] += bval[v_idx]*x[x_idx];
            }
```

Listing 3.2: BCSR SpMxV implementation. The first outer loop iterates over all block-rows. The second outer loop iterates over all blocks of a specific block-row. The last two innermost loops perform an $r \times c$ block matrix-vector multiplication.

A simple implementation of the SpMxV kernel for the BCSR storage format is presented in Listing 3.2, and it generally follows the structure of the CSR version. At the outermost loop, all block-rows are iterated, while the second loop performs an iteration over all blocks of a block-row. The two innermost loop perform appropriate computations for each $r \times c$ block. This version of the kernel is general in the sense that it does not assume a specific block shape. Generating BCSR SpMxV kernels for specific block shapes allows for various optimizations (e.g., register blocking and vectorization), that can significantly reduce the execution time of the kernel. A BCSR SpMxV kernel for 2×3 blocks is presented in Listing 3.3.

```
for (i=0; i < nbrows; i++){
    y0 = y1 = 0.0;
    for (j=brow_ptr[i]; j < brow_ptr[i+i]; j++){
        x_start = bcol_ind[j];
        v_start = j*2*3;

        x0 = x[x_start];
        x1 = x[x_start +1];
        x2 = x[x_start +2];

        y0 += bval[v_start]    * x0;
        y0 += bval[v_start +1] * x1;
        y0 += bval[v_start +2] * x2;

        y1 += bval[v_start +3] * x0;
        y1 += bval[v_start +4] * x1;
        y1 += bval[v_start +5] * x2;
    }
    y_start = i*2;
    y[y_start]    = y0;
    y[y_start +1] = y1;
}
```

Listing 3.3: BCSR 2×3 SpMxV implementation

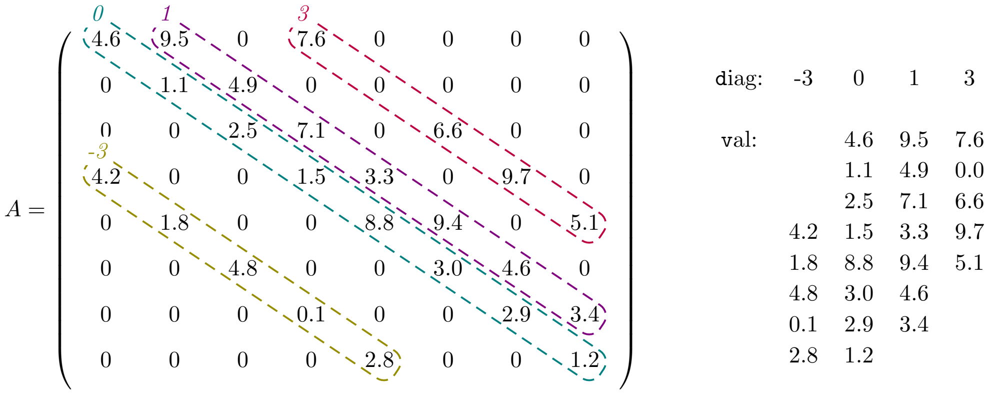
The selection of the block shape for a specific matrix is an important aspect of the BCSR format, one that has been extensively studied in related literature [Vud03, BELF07, KGK09a, KGK09b]. The optimal block shape selection for performing SpMxV is related not only to the specific sparse matrix structure (e.g., to avoid padding), but also to the architectural characteristic of the target processor (e.g., vector size, number of registers).

Aiming at flexibility in block shape selection, the Variable Block Row (*VBR*) format generalizes BCSR by allowing arbitrary block shapes. Nevertheless, this generality makes the VBR implementation complex, without providing any performance benefits [Vud03].

3.3.4 Formats for exploiting diagonal structure

Diagonal patterns arise frequently in sparse matrices, and for this reason several storage formats aim to exploit these patterns. The diagonal (*DIAG*) format is specifically designed for sparse matrices that contain only full — or almost full — diagonals. Diagonals which contain exclusively zeroes are discarded, while non-zero diagonals are stored fully, eliminating the need for indexing information about individual elements.

We consider an enumeration of the matrix diagonals: the main diagonal is numbered 0, diagonals in the upper triangle have positive numbers and diagonals in the lower triangle have negative numbers (see Figure 3.5). The DIAG format maintains two arrays: An array with s elements (`diag`) and an $s{\times}N$ array (`val`), s being the number of diagonals stored and N the number of matrix rows. For each diagonal i, its number is stored in `diag[`i`]` and its values are stored in column i of `val`. Elements storage in `val` ensures that they retain their matrix row number. More specifically, values of upper-diagonals are stored in `val` starting from the first row (0), while values of lower-diagonals are stored so their final element is placed on the last row ($N - 1$). Padding is applied as necessary: upper-diagonals are padded from the top, while lower-diagonals are padded from the bottom. The standard SpMxV implementation for this format is shown in Listing 3.4.



$$A = \begin{pmatrix} 4.6 & 9.5 & 0 & 7.6 & 0 & 0 & 0 & 0 \\ 0 & 1.1 & 4.9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.5 & 7.1 & 0 & 6.6 & 0 & 0 \\ 4.2 & 0 & 0 & 1.5 & 3.3 & 0 & 9.7 & 0 \\ 0 & 1.8 & 0 & 0 & 8.8 & 9.4 & 0 & 5.1 \\ 0 & 0 & 4.8 & 0 & 0 & 3.0 & 4.6 & 0 \\ 0 & 0 & 0 & 0.1 & 0 & 0 & 2.9 & 3.4 \\ 0 & 0 & 0 & 0 & 2.8 & 0 & 0 & 1.2 \end{pmatrix}$$

diag:	-3	0	1	3
val:		4.6	9.5	7.6
		1.1	4.9	0.0
		2.5	7.1	6.6
	4.2	1.5	3.3	9.7
	1.8	8.8	9.4	5.1
	4.8	3.0	4.6	
	0.1	2.9	3.4	
	2.8	1.2		

Figure 3.5: Example of the DIAG storage format.

```
for (j=0; j < s; j++){
    d = diag[j];
    for (i=max(0,-d); i < N - max(0,d); i++)
        y[i] += val[i][j] * x[d + i];
}
```

Listing 3.4: DIAG SpMxV implementation.

The DIAG format is efficient for matrices with full diagonals, but can be wasteful for matrices without full diagonals. The row segmented diagonal (*RSDIAG*) format [Vud03] is an approach for exploiting partial diagonals. RSDIAG divides the matrix into row segments (blocks of consecutive rows), and assumes full diagonals within each segment.

3.3.5 Composite formats

Storage formats try to exploit matrix structure regularities. Matrices with multiple types of regularities can be divided into sub-matrices, each stored in a different format. We refer to this class of formats as *composite formats*. Distributive operations (e.g., SpMxV) can be easily implemented by performing the operation for each sub-matrix:

$$A \cdot x = (A_0 + A_1 + \ldots + A_n) \cdot x = A_0 \cdot x + A_1 \cdot x + \ldots + A_n \cdot x$$

An example of a composite format is presented in [AGZ92]. In this work, Agarwal et al. decompose a matrix into three sub-matrices: the first is dominated by dense blocks, the second has a dense diagonal matrix, while the third contains the remainder of the matrix elements. A similar technique is described in [Vud03], where variable-block matrices are split into UBCSR sub-matrices. In composite formats, each sub-matrix is more sparse than the original matrix, i.e., it generally retains the same dimensions but has less non-zero elements. This can lead to computational overheads that can reduce performance (e.g., empty rows when iterating elements).

3.3.6 Symmetric and hermitian matrices

A class of matrices that emerge often in applications are symmetric matrices ($A_{ij} = A_{ji}$). These matrices can be stored in an optimized form, where symmetric elements are stored only once, i.e., by storing only the lower (or upper) triangle along with the main diagonal. To implement SpMxV for a symmetric storage format, each stored element with $i \neq j$ is used for two operations (one for itself and one for its transpose). As a result the performance profile of symmetric SpMxV is different than standard SpMxV for two main reasons: (a) computation load per element is doubled (b) random-access updates to *y* are performed. A similar class of matrices are hermitian matrices, i.e., complex matrices for which: $A_{ij} = A^*_{ji}$ — that is: $Re(A_{ij}) = Re(A_{ji}) \wedge Im(A_{ij}) = -Im(A_{ji})$.

3.4 Conclusions

In this chapter we discussed several sparse-matrix storage formats, focusing on the SpMxV operation. We distinguish two categories: *general* storage formats that do not assume specific matrix properties (e.g., CSR), and *specialized* storage formats that try to exploit specific matrix characteristics that are encountered frequently (e.g., DIAG). We should note that specialized storage formats may result in inefficiencies, when applied to inapt matrices.

Although our treatment is by no means comprehensive, it serves as an introduction to the necessary points for the rest of the text. More details about sparse computations and sparse matrix storage formats can be found in [BBC$^+$94, Saa03, Vud03, Wil08].

Sparse Matrix-Vector Multiplication performance 4

4.1 SpMxV algorithm

The nature of the matrix-vector multiplication algorithm results in memory-bound implementations imposing intense memory accesses. We will illustrate this property using a comparison with matrix-matrix multiplication (MxM). Matrix multiplication of $N \times N$ matrices A and B results in an $N \times N$ matrix, that can be expressed as:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

Using this (naive) algorithm*, computation of a single C element requires N multiplications and N additions. Consequently, computation of all C elements requires N^3 multiplications and additions. This is called the *surface-to-volume* effect, where the solution of a problem requires $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ amount of data.

As discussed in Section 3.2, the resulting y vector of the matrix-vector multiplication can be expressed as:

$$y_i = \sum_{j=1}^{N} A_{ij} \cdot x_j$$

This operation, namely dense matrix-vector multiplication, requires N multiplications and N additions for each element of y. Therefore, it performs $\mathcal{O}(n^2)$ operations on $\mathcal{O}(n^2)$ amount of data, resulting in a significantly higher ratio of memory accesses to floating-point operations compared to MxM. Seen from another point of view, there is little data reuse in the matrix-vector multiplication, i.e., very restricted temporal locality. In fact, matrix elements are used only once.

If the matrix is sparse, index data lead to additional memory references and cache interfer-

*There exist algorithms for matrix multiplication with a lower complexity than $\mathcal{O}(n^3)$. An example is Strassen's widely-known recursive algorithm with a complexity of $\mathcal{O}(n^{log_2 7}) \approx \mathcal{O}(n^{2.81})$ [CLRS01].

ence. In general, sparsity also creates irregular accesses to the input vector *x*. This irregularity complicates exploitation of reuse on *x* and increases the number of cache misses. Providing a through discussion of SpMxV performance, however, is a difficult task without assuming a specific sparse storage format. Thus, to provide a more in-depth analysis, we study the behavior of SpMxV on the CSR storage format. We choose CSR as the baseline for our performance analysis because it is a general format that performs well and it is widely used.

4.2 CSR SpMxV implementation

First, we discuss the CSR SpMxV implementation and analyze its memory references. Because we want to maximize performance, we optimize SpMxV so that the update on *y* vector is performed at the end of the innermost loop. This optimization opportunity is not easily detected by the compiler (mainly due to aliasing issues), and for this reason we use a temporary variable to store the intermediate result in our code. The resulting code is presented in Listing 4.1. Initial experimentation confirmed that this optimization results in significant performance improvement.

As can be seen in Listing 4.1, each element of the output vector `y[i]` is computed by iterating all elements of the `i`-th row. Specifically, all elements of the row are multiplied with the *x* element that corresponds to their column. Summation of the resulting values leads to the desired outcome. Figure 4.1 uses an example to illustrate these operations, and how they relate to SpMxV kernel data structures.

4.2.1 CSR SpMxV memory accesses breakdown

Table 4.1 shows a breakdown of the CSR SpMxV data set and how it is accessed. The `row_ptr` array has N elements, which are accessed sequentially and only once. The `values` and `col_ind` arrays are also accessed sequentially and only once, but have *nnz* elements. The *x* array is accessed randomly, but in increasing order within each row. Moreover, it is the only array that exhibits temporal locality, i.e., its elements are potentially accessed more than once. Finally, the *y* vector is the only array where SpMxV stores results; its elements are updated sequentially and only once. The preceding analysis clearly illustrates the streaming nature of the CSR SpMxV kernel and its restricted temporal locality. Another important aspect of the kernel's performance is its working set, which is discussed in the next paragraph.

4.2.2 Working Set

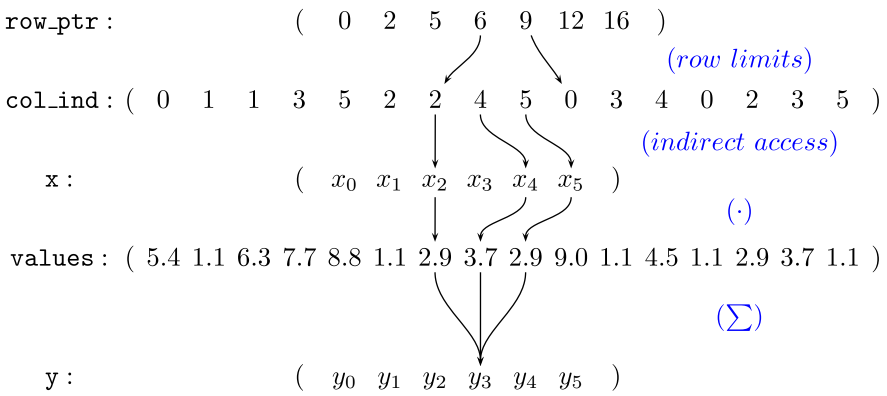
We refer to the data accessed during the execution of the SpMxV kernel as its *working set* (*ws*). The working set consists of the matrix and vector data; its size for the CSR storage format is expressed by the following formula:

$$ws = \overbrace{(nnz \cdot (s_{idx} + s_{val}) + (nrows + 1) \cdot s_{idx})}^{\text{sparse matrix}} + \overbrace{(nrows + ncols) \cdot s_{val}}^{\text{vectors}}$$

In the above formula, s_{idx} and s_{val} represent the storage size required for an index and a

```
for (i=0; i<N; i++){
    yr = 0.0;
    for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
        yr += values[j]*x[col_ind[j]];
    y[i] = yr;
}
```

Listing 4.1: CSR SpMxV implementation. The `y[i]` value is updated at the end of the inner loop



row_ptr : (0 2 5 6 9 12 16)
(row limits)
col_ind : (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5)
(indirect access)
x : (x_0 x_1 x_2 x_3 x_4 x_5)
$(\cdot)$
values : (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)
$(\sum)$
y : (y_0 y_1 y_2 y_3 y_4 y_5)

Figure 4.1: Example of a CSR SpMxV operation

	size	accesses	type	R/W
`row_ptr`	*N*	*N*	sequential	R
`values`	*nnz*	*nnz*	sequential	R
`col_ind`	*nnz*	*nnz*	sequential	R
x	*N*	*nnz*	random, ↑	R
y	*N*	*N*	sequential	W

Table 4.1: Breakdown of CSR SpMxV data set and their access patterns, for an $N \times N$ matrix.

value, respectively. An approximation of *ws* is:

$$ws = nnz \cdot (s_{idx} + s_{val})$$

This approximation, which accounts only for arrays with *nnz* elements, is valid for the majority of real-life sparse matrices because they satisfy: $nnz \gg nrows, ncols$. Commonly, a 4-byte integer is used for index storage, due to memory size restrictions that limit *x* and *y* vectors to a maximum of 2^{32} elements. On the other hand, numerical data — especially for scientific applications (e.g., PDE solvers) — normally require double-precision, i.e., 8 bytes. Under these assumptions ($s_{idx} =$ 4, $s_{val} = 8$) values constitute the larger portion of the working set by a factor of ⅔. For this reason, value data compression is expected to have a greater impact to overall working set reduction.

The ⅔ factor is a result of memory size limitations (we implicitly have assumed that the sparse matrix resides in memory) and can change in the future. Specifically, matrices with dimensions larger than 2^{32}, require indices larger than 32 bits. We consider a square sparse matrix with n = *nrows* = *ncols* = 2^{32} and *nnz* = $100 \cdot n = 100 \cdot 2^{32}$†. The required CSR storage would be $100 \cdot 2^{32} \cdot (4 + 8)$ bytes $\approx$ 4.7 TiB. Currently, only some high-end machines contain this much memory. However, given the current rate of advancement, it is probable that near-future commodity hardware will support these capabilities.

Using the previous working set approximation we can obtain the kernel's ratio of memory accesses to floating-point operations (FLOPs):

$$\rho = \frac{ws}{FLOPs_{(total)}} = \frac{nnz \cdot (4+8)}{nnz \cdot 2} = 6 \, bytes/FLOP$$

In other words, CSR SpMxV performs one floating-point operation per 6 bytes of data. This is a very high ratio and affirms our claim that the kernel is memory-bound. To make a connection with the memcomp benchmark (see 2.2) this ratio corresponds to a c value of ¾.

As another example, we consider a CPU with a clock of f = 2 GHz, which can perform $\alpha = 1$ FLOP per cycle. In this case, the required data transfer rate is:

$$f \cdot \alpha \cdot \rho = 6 \cdot 2 \cdot 10^9 = 12 \approx 11.2 \, GiB/sec$$

Although this ratio is attainable by modern machines (see Section 4.5.2), utilizing more cores will result in a multiplication of the required ratio by the number of added cores. As a result, the memory subsystem would not be able to deliver the necessary data transfer rate, and the kernel will exhibit poor scalability. Before presenting our performance evaluation of multithreaded SpMxV, however, we discuss its parallelization.

†The number 100 has been chosen because it is close to the average value of *nnz*/n for our matrix suite (see Table 4.3)

4.3 Multithreaded SpMxV

The SpMxV kernel is an easily parallelizable kernel, since it does not contain any loop-carried dependencies. Nevertheless, there exist a number of issues that should be taken into account during the parallelization process. These issues are discussed in the following paragraphs.

4.3.1 Data partitioning

There are several data partitioning schemes for parallelizing the SpMxV kernel on a shared memory architecture. For CSR, coarse-grained *row partitioning* is usually applied [WOV+07], where different blocks of rows are assigned to different threads (see Figure 4.2). Threads operate on disjoint subsets of `row_ptr`, `col_ind`, `values`, and y arrays. The only sharing occurs in x array data, but it does not constitute a performance problem: access to x is read only, allowing efficient data caching over all processors. Someone could argue that the common use of x offers potential for constructive cache sharing. In practice this potential is not realized, since shared data constitute a small part of the working set and cache space is limited.

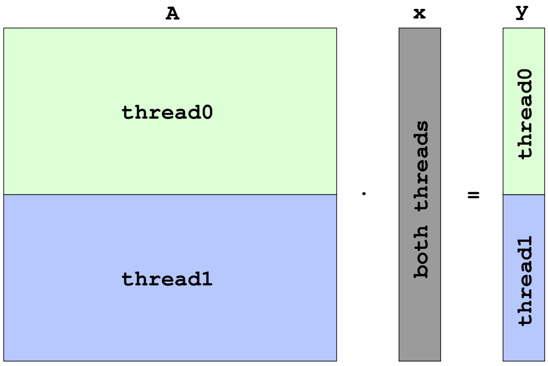


Figure 4.2: Row partitioning on SpMxV for two threads.

The complementary approach to row partitioning is *column partitioning*, where each thread is assigned a block of columns. Although this approach is more naturally applied to the CSC format, it can also be applied to CSR. An advantage of column partitioning is that each thread operates on a different part of the x vector, which allows for better temporal locality on the array's elements in case of distinct caches. A disadvantage, however, is the possibility of cache-line ping-pongs, since each thread performs updates over all y elements. Having each thread use its own y array eliminates this problem. The final result can be obtained by adding the partial y arrays. Nevertheless, a problem with this approach, as described in [BFF+09] by Buluç et al., is that the final accumulation does not scale, because one partial array per core is needed.

Block partitioning is the combination of the two aforementioned schemes where each thread is assigned a two-dimensional block. It has the benefit of allowing configurable data sizes for each thread. For this reason, it is applied when the available memory space is limited (e.g., in the Cell processor [GHF+06]). A work that discusses block partitioning in the context of distributed memory parallel architectures is [VB05].

For simplicity, in the remaining of this dissertation, we assume that row partitioning is used for parallelizing SpMxV. This simplification, however, does not have an effect on the validity of our analysis, since we discuss issues orthogonal to the partitioning scheme used.

4.3.2 Load balancing

An important issue that arises when parallelizing the SpMxV kernel is load balancing among different threads, i.e., how to distribute work so that each thread is assigned an equivalent volume of workload. A first approach is to assign the same number of rows to each thread. This naive scheme, however, can lead to imbalance because the non-zero elements of a sparse matrix are generally distributed unevenly across its rows. Consequently, if the sparsity pattern of the matrix is biased towards the upper or lower half, row partitioning will yield poor results. Initial experiments confirmed that this potential imbalance can have a negative performance effect on a significant number of matrices.

A better approach is to apply static balancing based on the non-zero elements, instead of the rows. In this case, each thread is assigned approximately the same number of elements and thus the same number of floating-point operations (see Algorithm 4.1). An example of this scheme is shown in Figure 4.3. In this example we consider two threads: the first thread is assigned the first 4 rows which contain 9 non-zero elements, while the second thread is assigned the remaining 2 rows which contain 7 non-zeroes. Note that a row balancing scheme would result in 5 elements for the first thread and 11 elements for the second thread — a less balanced partitioning.

Algorithm 4.1: Balancing based on non-zero elements, when row partitioning is applied

```
Input  : The number of non-zero elements ($nnz$)
Input  : The number of threads ($nthreads$)
Input  : The $row\_ptr$ array
Output: A $partition$ array

$tid \leftarrow 0$                                              // current thread id
$r \leftarrow 0$                                                 // current row number
$partition[0] \leftarrow 0$                                      // first thread starts at 0
for $tid \leftarrow 0$ to $nthreads$ do
    $elems \leftarrow 0$                                         // partition elements
    $limit \leftarrow \frac{nnz}{nthreads-tid}$                  // partition limit
    while $(elems < limit)$ do
        $elems \leftarrow elems + row\_ptr[r+1] - row\_ptr[r]$   // add row elements
        $r \leftarrow r + 1$
    $partition[tid+1] \leftarrow r$
    $nnz \leftarrow nnz - elems$
```

Although the aforementioned scheme distributes non-zero elements — almost — evenly among threads, imbalances may still be observed in actual workloads. This is mainly due to the fact that different sparsity patterns lead to different instruction streams regardless of the number of non-zero elements assigned to each thread. For example, if a thread is assigned a large number of short

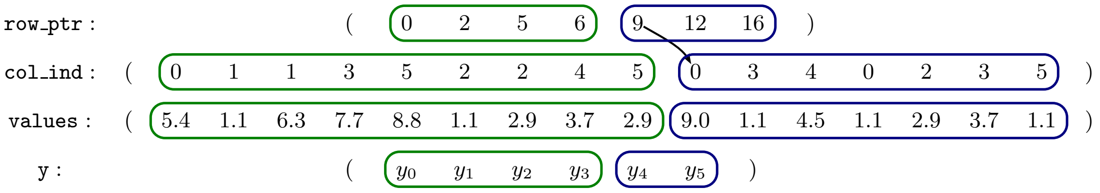

row_ptr : (0 2 5 6 9 12 16)
col_ind : (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5)
values : (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)
y : (y_0 y_1 y_2 y_3 y_4 y_5)

Figure 4.3: Example of non-zero elements balancing for two threads using matrix of Figure 3.3.

rows, then it will be further burdened from an increased amount of loop control instructions. A more sophisticated partition scheme is, however, outside the scope of this thesis.

4.4 SpMxV performance

This section is concerned with the performance issues of the SpMxV kernel. First, however, we need to clarify an important aspect of our methodology. To better simulate scientific applications that use SpMxV iteratively, we base our evaluation on performance measurements over multiple consecutive kernel invocations, which induces temporal locality into our workload. For this reason, we distinguish between two different matrix classes regarding SpMxV performance: (a) matrices whose working set fits perfectly into the aggregate cache size — the size of all available caches — thus experiencing only compulsory misses, and (b) matrices whose working set is larger than the aggregate cache size and experience capacity misses. We concentrate on the latter matrix class.

There are several performance problems of SpMxV reported in related literature (a general discussion of related works is presented in Section 4.6). These problems are listed below.

(a) *No temporal locality in the matrix.* This is an inherent problem of the algorithm which is irrelevant to the sparsity of the matrix. Unlike other important numerical codes, such as MxM and LU decomposition, the elements of the matrix in SpMxV are used only once [BELF07, MCG04].

(b) *Indirect memory references.* This is the most apparent implication of sparsity. In order to save memory space and floating-point operations, only the non-zero elements of the matrix are stored. To achieve this, the indices to the matrix elements need to be stored and accessed from memory via the `col_ind` and `row_ptr` data structures. This fact implies additional load operations, traffic for the memory subsystem, and cache interference [PH99].

(c) *Irregular memory accesses to vector* x. Unlike the case of dense matrices where the access to the vector x is sequential, in sparse matrices this access is irregular and depends on the sparsity structure of the matrix. This fact complicates the process of exploiting any spatial reuse in the access to vector x [GR99, Im00, PHCR04].

(d) *Short row lengths.* Although not so obvious, this problem is very often met in practice. Many sparse matrices exhibit a large number of rows with short length. This fact may degrade performance due to the significant overhead of the outer loop when the trip count of the inner loop is small [BELF07, WS97].

In [GKA$^+$08] we discuss and evaluate the above performance issues on modern microprocessors. For the sake of brevity we do not reproduce the full results here. Instead, in the following paragraphs, we provide only a brief summary of our conclusions, focusing on the parts that are relevant to this dissertation. The basis of our work is an extensive experimental evaluation of the SpMxV kernel for single and multi-threaded versions on a variety of modern commodity architectures. Using a rich set of matrices and various metrics, ranging from floating-point operations per second to processors' performance counters measures, we classify the effect of various SpMxV performance bottlenecks. Based on this classification, we provide a ranked list of optimization guidelines. According to our results, the steering performance impediment that should drive any subsequent optimization efforts is the memory intensity, i.e., the large memory bandwidth requirements, of the kernel. Optimizations that target other areas, e.g., computation, will have small impact on overall performance when the memory subsystem is not able to deliver data in time.

According to our study, the primary SpMxV optimization guideline is to reduce, as much as possible, the working set of the algorithm. Reducing the working set will certainly increase the computation to memory operations ratio, thus alleviating the pressure on memory bus and give better chance to pending memory requests to be served in time. Examples of working set reduction techniques include using 32-bit or 16-bit integers for the indexing structures of the matrix, applying blocking schemes (as in [PH99, IY01, BELF07, VM05]) that effectively reduce the size of indexing structures, or applying compression (as in [WL06]).

Motivated by this guideline, we developed three compression storage formats, which are discussed in chapters 5, 6 and 7. Prior to the discussion of these formats, we present the results of an extended performance evaluation of SpMxV in modern multicore architectures over a rich real-world matrix suite.

4.5 Experimental evaluation

The goal of our experiments is twofold: (a) to provide a quantitative evaluation of the SpMxV performance on real-world hardware and, more important, (b) to illustrate the poor scalability of SpMxV as a result of the memory bandwidth bottleneck. We perform experimental tests on systems that correspond to the ends of the commodity hardware spectrum regarding memory performance: an SMP system with centralized memory, and a new generation NUMA system, with a strong architectural focus on memory throughput performance. Our results clearly indicate that both systems are unable to deliver the required data transfer rate from main memory, when all available cores are utilized.

4.5.1 Experimental setup

Hardware

We conduct experiments on two systems. The first system is equipped with two quad-core Intel Harpertown processors (see Figure 4.4). Cores operate at 2 GHz, include two private L1 32 KiB caches (instructions and data), and are grouped in pairs that share a unified 6 MiB L2 cache. The processors interface with main memory via the Intel 5000p Memory Controller Hub (MCH) which provides four channels of fully buffered DDR2 DIMM (FB-DDR2) memory.

In contrast with Harpertown that uses a unique interface with main memory, the second system consists of two Intel Nehalem‡ processors that implement a NUMA architecture. Each processor has four cores that operate on 2.8 GHz; each core has private L1 (32 KiB instructions and data) and L2 (256 KiB unified) caches, while cores of the same processor share an L3 (8 MiB unified) cache. Moreover, Nehalem is equipped with an on-chip memory controller that supports three DDR3 memory channels. Communication with other memory nodes and I/O devices is implemented via QuickPath (QP) interconnect point-to-point links (see Figure 4.5). Additionally, Nehalem cores implement simultaneous multithreading (SMT) [TEL95], providing two different thread contexts per core.

As depicted in Figures 4.4 and 4.5, real-world systems usually employ a hierarchical topology where different core sets share different parts of the memory hierarchy. To distinguish between different scheduling configurations we will use a notation that explicitly describes the number of threads used in each level of the hierarchy. The levels are represented as:

- `t` : SMT threads on the same core (Nehalem).
- `c0` : cores that share L2 (Harpertown)
- `c1` : cores that do not share L2 (Harpertown)
- `c` : cores that share L3 (Nehalem)
- `d` : different dies (Harpertown and Nehalem)

Table 4.2 provides a concise overview of the two processors used for our experimental evaluation.

Software

For our evaluation we compiled our code with gcc 4.3.2, and performed our experiments in a 64-bit version of the Linux operating system (2.6.30). We explicitly parallelized all versions of the SpMxV kernel using the pthreads interface of the GNU libc library (NPTL 2.7). Moreover, we bound threads to specific cores using the `sched_setaffinity()` system call, and we allocated memory from specific NUMA nodes using the libnuma library (version 2.0.2).

We set the default storage size for indices and values to 32 and 64 bits respectively. The experiments were conducted by measuring the execution time of 128 consecutive SpMxV operations.

‡An initial performance evaluation of a Nehalem system can be found in [BDH$^+$08].

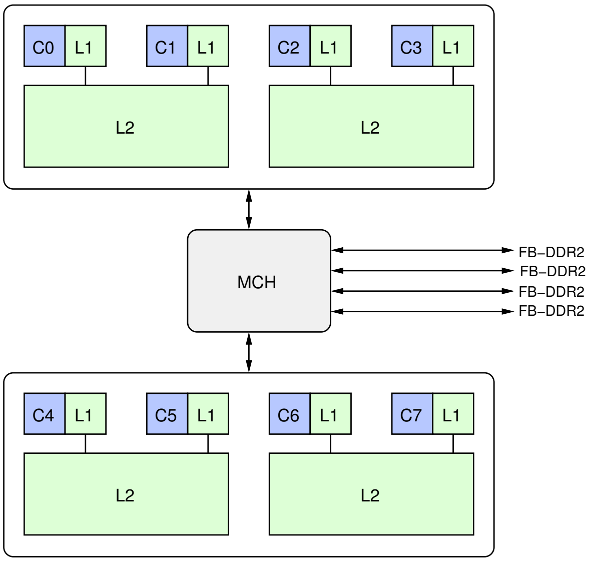


Figure 4.4: An 8-core system comprising of two Harpertown processors.

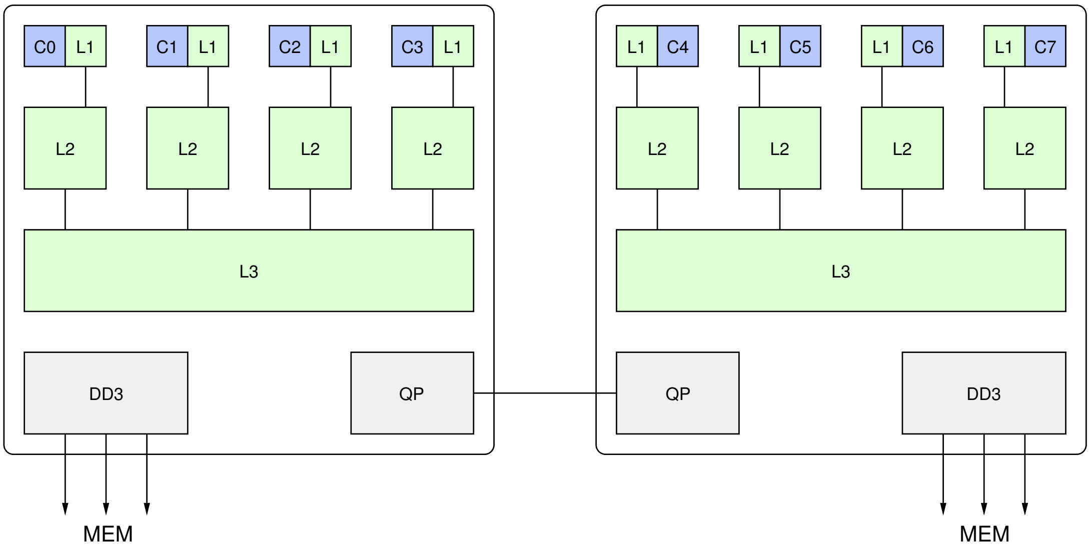


Figure 4.5: An 8-core system comprising of two Nehalem processors.

System	**Harpertown**	**Nehalem**
Model	E5405	X5560
Frequency (Ghz)	2.0	2.8
L1 (data/instruction)	32k/32k	32k/32k
L2 (unified)	6M (1/2 cores)	256k (1/core)
L3 (unified)	-	8M (1/chip)
Multithreading configuration	2c0 x 2c1 x 2d	2t x 4c x 2d

Table 4.2: Overview of the systems used in the experimental evaluation.

We made no attempt to artificially pollute the cache after each iteration, to better simulate iterative scientific application behavior where matrix data are present in the cache hierarchy, because either they have just been produced, or they were recently accessed. Additionally, we set x to be the y vector of the previous iteration, so that our benchmark has similar behavior with scientific methods based on SpMxV (e.g., GMRES). Setting y as x, however, restricts our matrix suite to contain only square matrices.

4.5.2 Memory throughput benchmark

To quantify system limits and the role of the various micro-architectural characteristics, we developed a benchmark to measure maximum throughput when a number of threads read data from the main memory (see A.2 for more details). These measurements can be used to reveal system performance trends for memory-intensive applications, such as the SpMxV kernel.

Results for the Harpertown system are shown in Figure 4.6, which illustrates the achieved memory throughput for different scheduling configurations. As expected, scalability is poor. For example, when all available cores are used, the memory throughput is increased only by a factor of 1.62 compared to the single thread scenario. This scalability problem is more intense for threads that operate on the same die: two threads in the same core achieve only a 1.12 throughput increase when compared to the serial case, while the same number of threads in different dies achieve about 1.54 increase. Another observation from the diagram is that concurrent memory accesses may lead to performance degradation due to contention. For example, the throughput of 8 threads is less than the throughput of the `2c0`×`2d` configuration.

The results for the Nehalem system are presented in Figure 4.7. Figure 4.7a shows the achieved memory throughput of one thread for three different NUMA memory allocation policies: (a) *local*: allocation on the local node, (b) *remote*: allocation on the remote node and (c) *interleaved*: alternating page allocation over all nodes. Local node allocation outperforms remote and interleaved policy by a significant factor (1.51 and 1.25 respectively). A single thread achieves 11.1 GiB/sec when reading from a local node, which constitutes a 3.1 improvement over Harpertown single-thread performance. Figure 4.7b presents results for various thread configurations when using memory allocated on the local NUMA node for each thread. NUMA allows for good scalability when different processors are used. The speedup achieved for two threads running on different dies is — as expected — almost linear (1.96), and when all cores are utilized the speedup is 3.27. It also is worth noting that when all cores are utilized the Nehalem processor outperforms the Harpertown processor by a factor of 6.25 (36.4 vs 5.9 GiB/sec) in this benchmark.

A comparison between these two systems shows a technology shift towards designs that focus on memory throughput performance, and indicates the importance of the memory subsystem for future multicore systems. Regarding SpMxV, we expect that the kernel will scale better in Nehalem, especially if it is assured that data are distributed among NUMA nodes so that each thread accesses local memory.

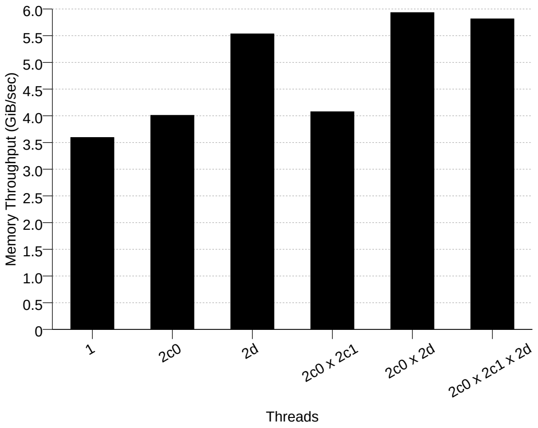
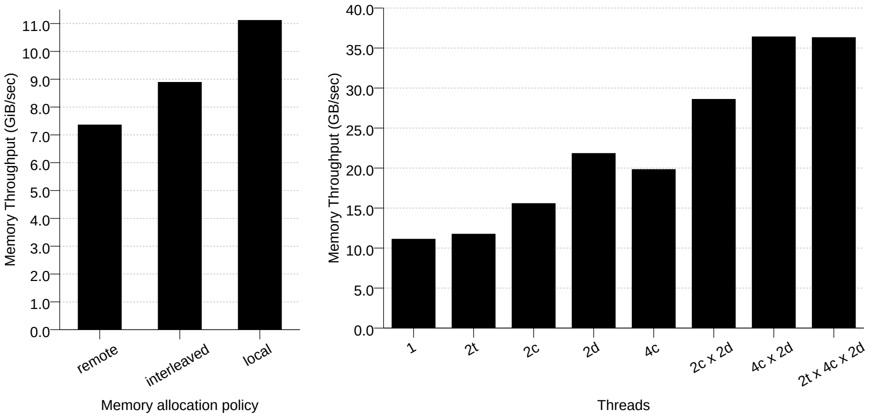


Figure 4.6: Read memory throughput for Harpertown.



(a)

(b)

Figure 4.7: Read memory throughput for the Nehalem. (a): Nehalem read memory throughput for one thread and different NUMA allocation policies. (b): Nehalem read memory throughput for different thread configurations. Local node allocation policy is used.

4.5.3 Matrix suite

Iterative use of SpMxV induces temporal locality to the application. Hence, the streaming behavior of the kernel is maintained only if the working set, and more specifically the matrix data, is significantly larger than the system's aggregate cache. For this reason, we build our matrix suite using matrices with a CSR working set larger than $4 \cdot 6 = 24$ MiB, which is the greater aggregate cache for the systems used in our experimental evaluation. Our matrix set consists of 50 matrices which are listed in Table 4.3.

name	dim	nnz	size	name	dim	nnz	size
	$/10^3$	$/10^6$	/1MiB		$/10^3$	$/10^6$	/1MiB
boneS10	914.9	55.5	638.3	G3_circuit	1,585.5	7.7	93.7
ldoor	952.2	46.5	536.0	cage13	445.3	7.5	87.3
inline_1	503.7	36.8	423.3	rajat30	644.0	6.2	73.1
fdif202x202x102	4,000.0	27.8	333.9	pre2	659.0	6.0	70.7
F1	343.8	26.8	308.4	Hamrle3	1,447.4	5.5	68.6
rajat31	4,690.0	20.3	250.4	largebasis	440.0	5.6	65.3
msdoor	415.9	20.2	233.2	Chebyshev4	68.1	5.4	61.8
Freescale1	3,428.8	18.9	229.6	apache2	715.2	4.8	57.9
Ga41As41H72	268.1	18.5	212.6	s3dkq4m2	90.4	4.8	55.5
af_shell9	504.9	17.6	203.2	ship_001	34.9	4.6	53.3
af_5_k101	503.6	17.6	202.8	torso3	259.2	4.4	51.7
TSOPF_RS_b2383	38.1	16.2	185.2	thread	29.7	4.5	51.3
kkt_power	2,063.5	14.6	175.1	ASIC_680k	682.9	3.9	46.9
Si41Ge41H72	185.6	15.0	172.5	large-dense	2.0	4.0	45.8
random100000	100.0	15.0	171.8	barrier2-9	115.6	3.9	45.0
nd12k	36.0	14.2	162.9	xenon2	157.5	3.9	44.9
crankseg_2	63.8	14.1	162.2	parabolic_fem	525.8	3.7	44.1
pwtk	217.9	11.6	134.0	FEM_3D_thermal2	147.9	3.5	40.5
bmw3_2	227.4	11.3	130.1	sme3Dc	42.9	3.1	36.2
ohne2	181.3	11.1	127.3	stomach	213.4	3.0	35.4
hood	220.5	10.8	124.1	thermomech_dK	204.3	2.8	33.4
Si87H76	240.4	10.7	122.9	helm2d03	392.3	2.7	32.9
bmwcra_1	148.8	10.6	122.4	ASIC_680ks	682.7	2.3	29.3
atmosmodj	1,270.4	8.8	105.7	poisson3Db	85.6	2.4	27.5
thermal2	1,228.0	8.6	102.9	rma10	46.8	2.4	27.3

Table 4.3: Matrix suite used for the experimental evaluation. Columns contain information about each matrix: *dim* contains the number of rows and columns of the matrix in thousands (*nrows* = *ncols*, since we consider only square matrices), *nnz* contains the number of non-zero elements in millions and *size* contains the matrix size in MiB when stored in CSR format.

The majority of the matrices represent real-world problems and were selected from the University of Florida Sparse Matrix Collection [Dav97]. Our suite includes the `fdif202x202x102` matrix, which is a matrix obtained by a 5-pt finite difference problem for a $202 \times 202 \times 102$ regular grid created by SPARSKIT [Saa94], and two artificial matrices that represent the two ends of the sparsity spectrum: (a) a dense 2000×2000 matrix (`large-dense`) and (b) a random 100000×100000 sparse matrix (`random100000`).

4.5.4 CSR performance evaluation

Figure 4.8 illustrates the average speedup of multithreaded CSR over all matrices, for different thread scheduling configurations on the Harpertown system. The speedup for 8 threads is 1.9, demonstrating the poor scalability of SpMxV. The speedup increase observed between the `2c0` and `2c1` cases — 1.17 and 1.23 respectively — can be accredited to matrix data caching during consecutive SpMxV executions. Cases `2c1` and `2c0`×`2c1` achieve roughly the same performance, even though available cores are doubled. We attribute this fact to the limited memory bandwidth since, as is shown in Figure 4.6, the available memory throughput for 2 and 4 cores in a single die is essentially the same. A more detailed view of SpMxV performance for Harpertown is presented in Figure 4.10, illustrating the performance of individual matrices in FLOPS per second for all different thread scheduling configurations.

A NUMA-oblivious multithreaded program can run unmodified in a NUMA system. However, there is no guarantee that data placement will be efficient. Thus, to maximize performance, we developed NUMA-aware versions of our methods, where memory allocation ensures that data accessed from a single thread are placed into the local NUMA node of the corresponding processor.

Figure 4.9 presents results for the Nehalem system for two versions of the CSR SpMxV kernel: default allocation (NUMA-oblivious) and local allocation (NUMA-aware). The large memory throughput capabilities of Nehalem result in noticeably better performance than Harpertown, even for the NUMA-oblivious version. The NUMA-aware version further improves performance, achieving a 4.44 speedup for the `4c`×`2d` case. SMT threads utilization in this case, however, degrades performance (4.31). Figure 4.11 contains detailed experimental results for the performance of the NUMA-aware SpMxV version.

Even though the Nehalem memory subsystem architecture drastically increases CSR SpMxV performance, it is still far from the theoretical maximum, leaving room for performance improvement by applying compression schemes. In the remaining of this dissertation, we present only NUMA-aware versions for all methods on the Nehalem system to focus on cases that maximize performance.

4.5.5 Summary

In conclusion, we argue that the SpMxV kernel is a good candidate for applying compression schemes: (a) its performance is dominated by a memory bandwidth bottleneck (b) its data, at least for real-world applications, are likely to contain redundancies that favour compression and (c) the compression overhead can be amortized, since it is used in an iterative manner. We conclude this chapter by discussing related work.

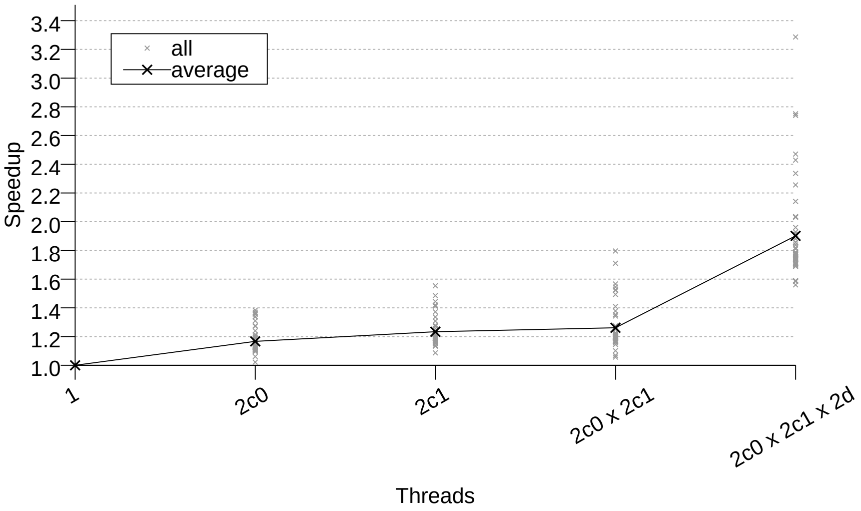


Figure 4.8: Speedups achieved by CSR SpMxV on Harpertown. Gray points mark the speedup for each matrix, while black points designate the average speedup achieved.

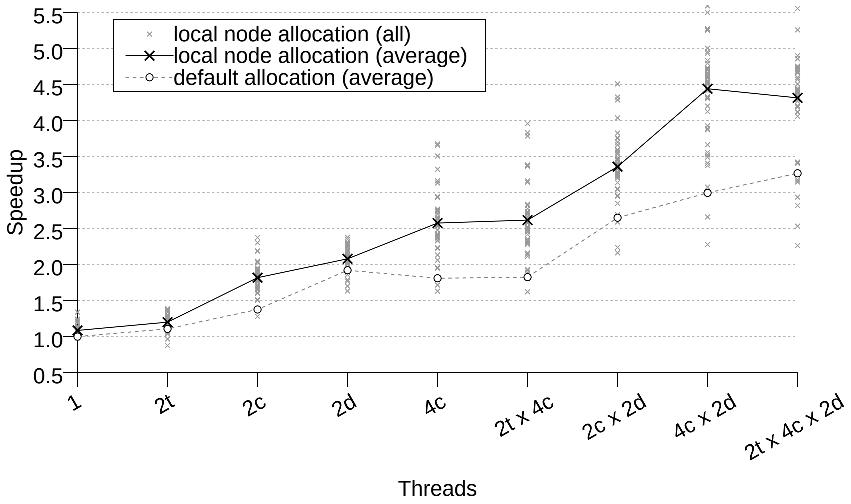


Figure 4.9: Speedups achieved by CSR SpMxV on Nehalem. The "local node allocation" line corresponds to a NUMA-aware version of the kernel, that binds matrix data in local NUMA node memory. The speedup for Nehalem is obtained using the single-threaded performance with default allocation, as the base performance. Gray points show the speedup achieved for individual matrices when local-node allocation is used.

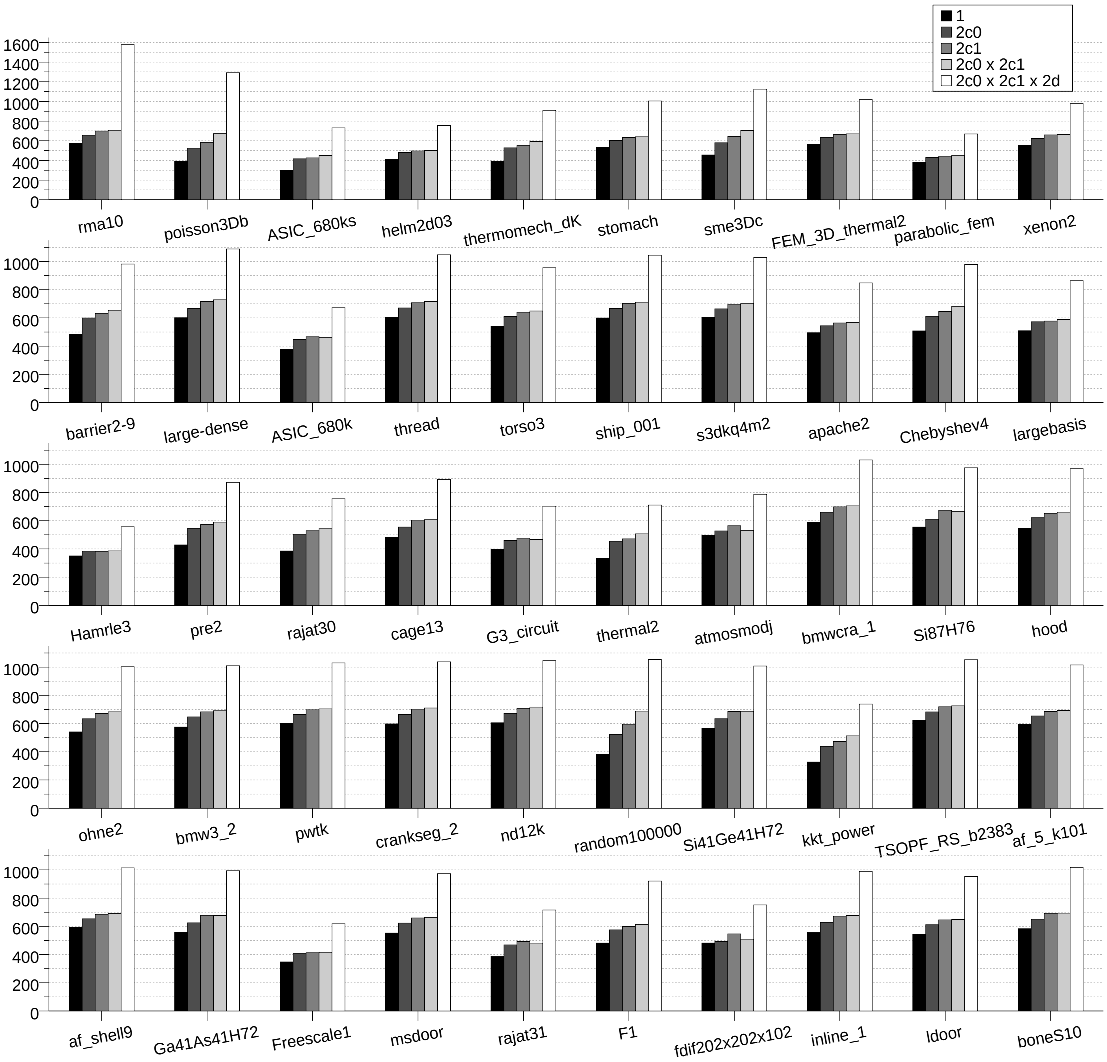


Figure 4.10: CSR SpMxV performance in FLOPs per second for all matrices and all thread scheduling configurations for Harpertown.

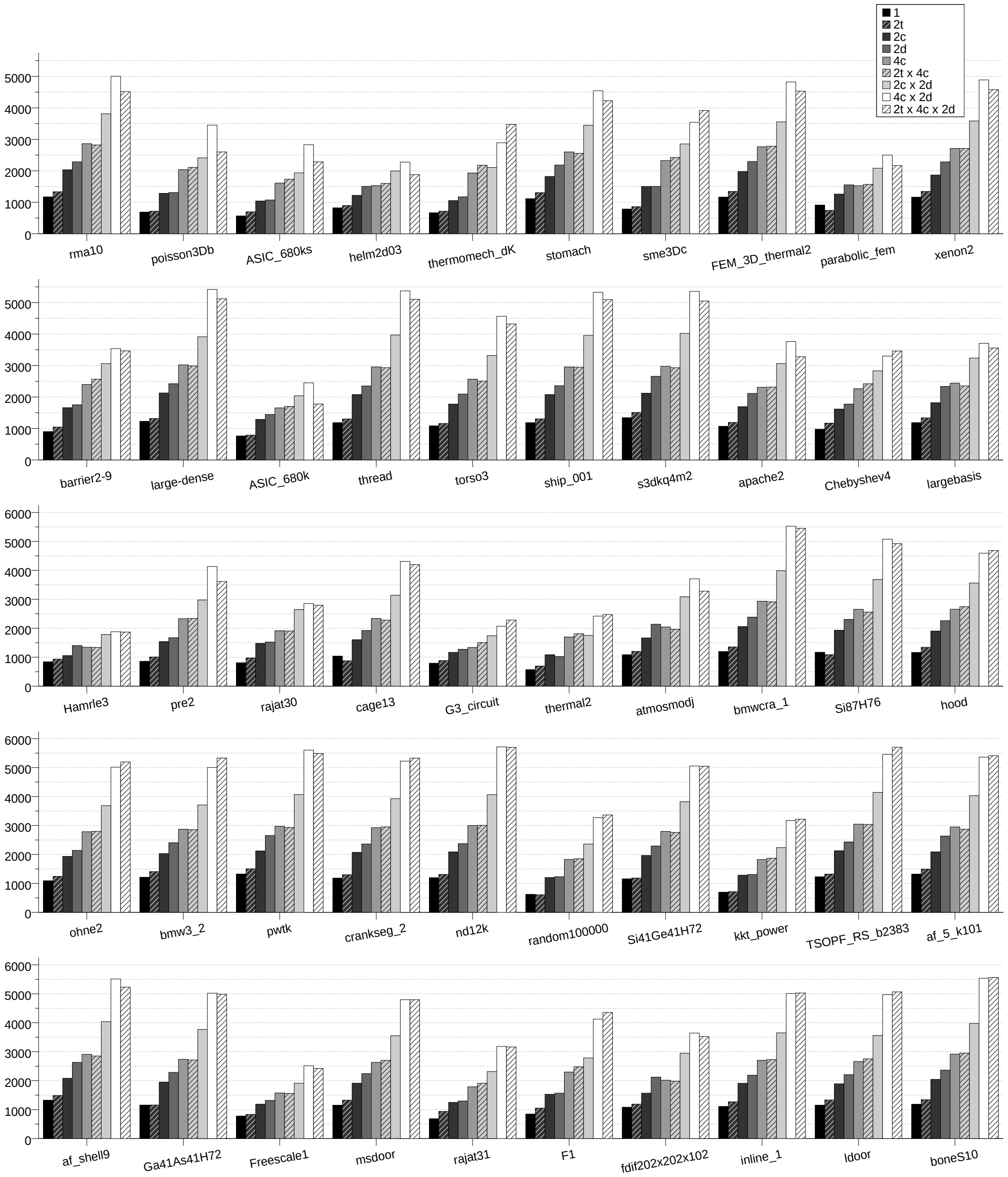


Figure 4.11: CSR SpMxV performance in FLOPs per second for all matrices and all thread scheduling configurations for Nehalem (NUMA-aware version).

4.6 Related work

4.6.1 Serial SpMxV

Because of its importance, sparse matrix-vector multiplication has attracted intensive scientific attention during the past two decades. The proposal of efficient storage formats for sparse matrices like CSR, BCSR, CDS (Compressed Diagonal Storage), Ellpack-Itpack, and JAD (Jagged Diagonal) [PRdB89, BBC$^+$94, Saa03] was one of the primary concerns. Elaborating on storage formats, Agarwal et al. [AGZ92] decompose a matrix into three sub-matrices: the first is dominated by dense blocks, the second has a dense diagonal matrix, while the third contains the remainder of the matrix elements. By using a different format for each sub-matrix, the authors try to optimize execution based on the special characteristics of each sub-matrix. Temam and Jalby [TJ92] perform a thorough analysis of the cache behavior of the algorithm, pointing out the problem of the irregular access pattern in the input vector x. Toledo [Tol97] deals with this problem by proposing a permutation of the matrix that favors cache reuse in the access of x. Furthermore, the application of blocking is also proposed in that work in order to both exploit temporal locality on x and reduce the need for indirect indexing through `col_ind`. Software prefetching for the sparse matrix and `col_ind` is also used to improve memory access performance. The proposed techniques were evaluated over 13 sparse matrices on a Power2 processor and achieved a significant performance gain for the majority of them. White and Sadayappan [WS97] state that data locality is not the most crucial issue in sparse matrix-vector multiply. Instead, small line lengths, which are frequently encountered in sparse matrices, may drastically degrade performance due to the reduction of ILP. For this reason, the authors propose alternative storage schemes that enable unrolling. Their experimental results exhibited performance gains on a HP PA-RISC processor for each of the 10 sparse matrices used. Pinar and Heath [PH99] refer to irregular and indirect accesses on x as the main factors responsible for performance degradation. Focusing on indirect accesses, the application of one-dimensional blocking with the BCSR storage format is proposed in order to drastically reduce the number of indirect memory references. In addition, a column reordering technique which enables the construction of larger dense sub-blocks is also proposed. An average 1.21 speedup is reported for 11 matrices on a Sun UltraSPARC II processor. Silva and Wait [SW05] investigate the effect of keeping both indices and values in a single data structure.

With a primary goal to exploit reuse on vector x, Im and Yelick propose the application of register blocking, cache blocking, and reordering [IY99, Im00, IY01]. Moreover, their blocked versions of the algorithm are capable of reducing loop overheads and indirect referencing while increasing the degree of ILP. Register blocking is the most promising of the above techniques. The authors also propose a heuristic to determine an efficient block size. They perform their experiments on four different processors (UltraSPARC I, MIPS 10000, Alpha 21164, PowerPC604e) for a wide matrix suite involving 46 matrices. For almost a quarter of these matrices, especially those that contained dense sub-blocks derived from FEM discretizations, register blocking achieved significant performance benefits. However, as the matrices were becoming increasingly irregular with few dense blocks, the performance of the proposed approach degraded rapidly due to the overhead imposed by the additional zero elements padded to form dense blocks. For highly irregular matrices the method was not capable of finding any efficient block size, thus collapsing to the proposal of the standard 1×1 block. Geus and Röllin [GR99] apply software pipelining to increase ILP, register blocking to reduce indirect references, and matrix reordering to exploit the reuse on x. They perform a set of experiments on a variety of processors (Pentium III, UltraSPARC, Alpha 21164, PA-8000, PA 8500, Power2, i860 XP) and report significant performance gains on two matrices originating from the discretization of 3-D Maxwell's Equations with FEM. Vuduc et al. [VDY$^+$02] estimate the performance bounds of the algorithm and evaluate the register blocked code with respect to these bounds. Furthermore, they propose a new approach to select near-optimal register block sizes. Mellor-Crummey and Garvin [MCG04] accentuate the problem of short row lengths and propose the application of the well-known unroll-and-jam compiler optimization in order to overcome this problem. Unroll-and-jam achieves a 1.11–2.3 speedup on MIPS R12000, Alpha 21264A, Power3-II, and Itanium processors for two matrices taken from the SAGE package. Pichel et al. [PHCR04] model the inherent locality of a specific matrix with the use of distance functions and improve this locality by applying reordering to the original matrix. The same group proposes also the use of register blocking to further increase performance [PHCR05]. The authors report an average of 15% improvement for 15 sparse matrices on MIPS R10000, UltraSPARC II, UltraSPARC III, and Pentium III processors.

Buttari et al. [BELF07] provide a performance model for the blocked version of the algorithm based on BCSR format and propose a method to select dense blocks efficiently. They experiment on a K6, a Power3, and an Itanium II processor for a suite of 20 sparse matrices and validate the accuracy of the proposed performance model. Vuduc et al. [VM05] extend the notion of blocking in order to exploit variable block shapes by decomposing the original matrix to a proper sum of sub-matrices storing each sub-matrix in a variation of the BCSR format. Their approach is tested on the Ultra2i, Pentium III-M, Power4, and Itanium II processors for a suite of 10 FEM matrices that contain dense sub-blocks. The proposed method achieves better performance than pure BCSR on every processor, except for Itanium II.

Willcock and Lumsdaine [WL06] mitigate the memory bandwidth pressure by providing an approach to compress the indexing structure of the sparse matrix, sacrificing in this way some CPU cycles. They perform their experiments on a PowerPC 970 and an Opteron processor for 20 matrices achieving an average of 15% speedup. Another recent work that targets performance improvement by reducing the index data volume is [BBR09], where Belgin et. al propose a matrix representation that exploits repeated block patterns. The authors search for frequently met block patterns and generate specialized inner loops for those, on top of a dispatch logic. They provide an evaluation of a parallel version, but they focus primarily on serial performance.

4.6.2 Multithreaded SpMxV

As far as the parallel, multithreaded version of the code is concerned, past work focuses mainly on SMP clusters, where researchers either apply and evaluate known uniprocessor optimization techniques on SMPs, such as register or cache blocking [IY99, GR99], or examine reordering techniques in order to improve locality of references and minimize communication cost [PHCR04, CA96]. More specifically, Im and Yelick [IY99] apply register and cache blocking on an 8-way UltraSparc SMP. They also examine reordering techniques combined with register blocking. However, the results are satisfactory only in the case of highly irregular sparse matrices, but the scalability of the algorithm is still very low. Pichel et al. [PHCR04] also examine reordering techniques and locality schemes. They propose two locality heuristics based on row or row-block similarity patterns, which they use as objective functions to two reordering algorithms in order to gain locality. Results are presented in terms of L1 and L2 cache miss rate reduction based mainly on a trace-driven simulation. The effect of these reordering techniques in load balancing is also discussed. Geus and Röllin [GR99] examine three parallelization schemes using MPI combined with Cuthill-McKee reordering technique in order to minimize data exchange between processors. Experiments are conducted on a series of high performance architectures, including, among others, the Intel Paragon and the Intel Pentium III Beowulf Cluster. The authors also outline the problem of the interconnection bandwidth while commenting on the results. In a higher level, Catalyuerek and Ayakanat [CA96] propose an alternative data partitioning scheme based on hypergraphs in order to minimize communication cost. Kotakemori et al. [KHK+05] evaluate different storage formats of sparse matrices on a SGI Altix3700 ccNUMA machine using an OpenMP parallel version of the SpMxV code. The authors implement a NUMA-aware parallelization scheme, which yields almost linear speedup in every case.

Quite recently, Williams et al. [WOV+07, WOV+09] have presented an evaluation of SpMxV on a set of emerging multicore architectures. Their study covers a wide and diverse range of high-end chip multiprocessors, including recent multicores from AMD (Opteron X2) and Intel (Clovertown), Sun's Niagara2 and platforms comprised of one or two Cell processors. The authors offer a clear view of the gap between the attained performance of the kernel, and the peak performance of each architecture it is executed, both in terms of memory bandwidth and computational throughput. Their work includes a rich collection of optimizations, some of which are targeted specifically at multithreading architectures. They perform an experimental evaluation on a set of 14 matrices. In their conclusions they state that memory bandwidth could be a significant bottleneck and advocate working set reduction techniques. It should also be noted that one of the optimizations they apply is a simple index reduction technique, in which 16-bit indices are used when this is applicable. Finally, Buluç et al. [BFF+09], focusing on multicore architectures, propose CSB — a storage format that aims to enable efficient execution of both the sparse matrix-vector and sparse matrix-transpose-vector kernels.

5 Column Index compression using Delta Encoding

5.1 Motivation and general approach

Sparse storage formats traditionally try to exploit contiguous elements, either in one (Figure 5.1a) or two dimensions (Figure 5.1b). Examples include the BCSR format, and the variable length one-dimensional block format described in [PH99]. BCSR can be viewed as a generalization of CSR where the granularity unit is an $r \times c$ dense block. The effect to overall matrix size when converting from CSR to BCSR depends on the aptitude of the selected block shape to capture the matrix structure. If resulting blocks contain a small number of zeroes, significant index reduction is achieved. For example, perfect blocking — i.e., none of the BCSR blocks contain zeroes — leads to an index reduction by a factor of $r \cdot c$. On the contrary, zeroes included in BCSR blocks must be explicitly added to value data, because all BCSR blocks are stored in a dense form. This, depending on the matrix structure and selected block shape, may lead to an increase in overall matrix size.

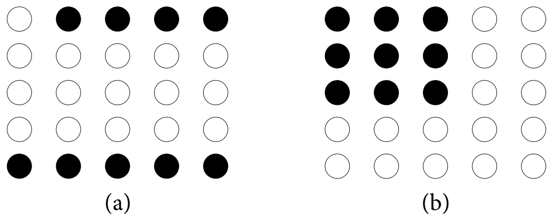


Figure 5.1: Sparse matrix patterns. (a) sequential elements, (b) two-dimensional blocks.

Our index compression approach is based on the general premise that sparse matrices have dense areas that do not necessarily contain contiguous non-zero elements (i.e., areas where elements are close but not sequential). These areas can contribute significantly to index data size reduction when delta encoding is used to reveal the highly redundant nature of the `col_ind` array [WL06]. In a delta encoding scheme the column indices are replaced with *deltas*, each of which is defined as the difference of the current index with the previous one. Within a row, delta values are positive and less or equal than their correspoding column indices. Hence, delta values can be stored in smaller size integers, leading to index data size reduction. For example, Table 5.1 presents column indices along with the corresponding delta-encoded values, taken from the 33th row of the `rajat30` matrix (see Section 4.5.3).

indices	40	41	450	1812	1840	3203	3233	3235	3241	3245
deltas	...	1	409	1362	28	1363	30	2	6	4

Table 5.1: Example of column indices and their corresponding delta-encoded values (taken from matrix `rajat30`).

A simple compression scheme to exploit delta encoding is to use variable-length integers. We consider a method where the integer's bits in its normal form are divided in 7-bits parts. These parts are stored in consecutive bytes in which the most significant bit (MSB) is used to mark the last byte of the integer. Under this scheme, an integer with a value of x needs $1 + \lfloor \frac{\log_2 x}{7} \rfloor$ bytes of storage. This can lead to significant index data volume reduction. For the example of Table 5.1, column indices smaller than $2^7 = 128$ will be encoded using 1 byte and all others using 2 bytes. As a result, the total size required for column indices 41 to 3245 is 12 bytes. A CSR `col_ind` structure, on the other hand, would require $9 \cdot 4 = 36$ bytes.

There is, however, a performance issue with variable-length integers. Normally, if each delta value was encoded separately, the innermost loop of the SpMxV kernel would contain branches to implement decoding (e.g., checking MSB to determine whether the integer includes other bytes). Misprediction of these branches in execution time leads to significant performance degradation. For this reason, instead of encoding each delta value to use only the necessary number of bytes, we propose a coarse-grained approach where the matrix is divided into *units* with a variable number of elements. For each of these units, the maximum delta value is calculated, and a size that can represent this value is selected for all the delta values of the unit. This technique enables for innermost loops with minimum overheads by sacrificing some space.

An important factor for the performance of this method lies in the selection of the unit size. If the size is too small, the decompression overhead introduced will dominate the performance gain from the compression. On the contrary, if the unit size is large, there will be less opportunities for compression, because a single large delta value will enforce big storage requirements for the whole unit.

This approach demonstrates an abstract optimization strategy for the SpMxV kernel, that can be used to exploit matrix-specific structure information. To this direction the concept of *units* could be extended to support more types of regularities, thus providing a number of advantages: (a) It can be used to exploit local regularities in specific areas of the matrix, (b) It operates on a coarse-grained level and thus can effectively minimize the introduced overhead by selecting sufficiently large sizes and (c) it can bound the search space for regularities or patterns and assure that the compression procedure will not exceed the available resources (e.g., time or storage). In Section 5.2.2 we discuss a method for exploiting sequential elements, while Chapter 7 describes a more general storage format towards this direction.

5.2 The CSR-DU storage format

The CSR-DU (CSR with Delta Units) storage format divides index data into units which are stored in a single byte-array called `ctl`. Each unit is limited to elements of a single row and consists of two parts. First, the *header* where the unit's properties are stored. Second, the *main body* where the delta-encoded column indices are stored. The header, in its simplest form, consists of two one-byte fields: (a) `usize`, the number of elements the unit contains and (b) `uflags`, a bit-vector encoding the unit's characteristics. Since `usize` is stored in a single byte, the maximum possible number of elements per unit is $2^8 = 256$. The size (1, 2, 4 or 8 bytes)* of the delta values stored in the main body can be extracted from the `uflags` field, along with a marker that designates the beginning of a new row.

Figure 5.2 presents an example of the CSR-DU format. In this example a row with 8 elements is split into two units. The first unit has 5 elements, 1-byte delta size, and a designator for a new row (`nr`). The second unit has 3 elements, and 2-byte delta size.

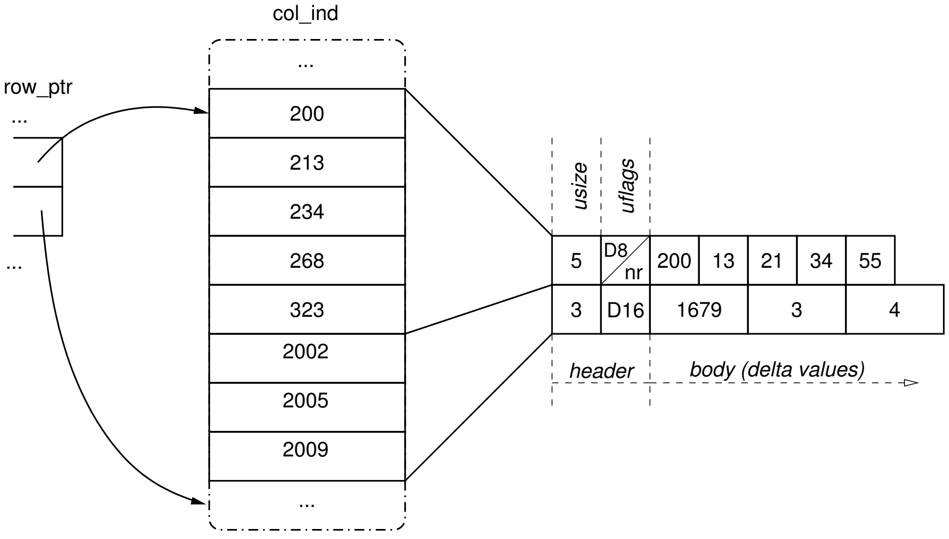


Figure 5.2: Example of the CSR-DU storage format.

The compression procedure of CSR-DU is straightforward. It is performed in $\mathcal{O}(nnz)$ steps by scanning the matrix elements once, while keeping appropriate information in buffers until a unit is finalized. This means that the construction process of CSR-DU involves no overhead in terms of time complexity compared to CSR. An important decision during this procedure is when to finalize a unit. We implemented a simple approach where a unit is finalized if (a) a new row starts in the next element, or (b) the maximum unit size is reached. An algorithm for this procedure is shown in Algorithm 5.1 that uses a finalization function, described in pseudocode in Algorithm 5.2, to append the necessary data to the `ctl` array. A more elaborate scheme would be to finalize a unit if a new element increases the delta storage size, as long as the unit already contains more than a predetermined number of elements.

*8 bytes delta values are unnecessary due to hardware limitations, but supported for completeness.

Algorithm 5.1: Basic CSR-DU encoding procedure.

```
Initialization:
  $deltas$    $\leftarrow [\,]$     // array of column deltas for current unit
  $newrow$    $\leftarrow$ true   // true if current unit starts a new row
  $Y_{prev}$  $\leftarrow 1$      // previous element's row number
  $X_{prev}$  $\leftarrow 0$      // previous element's column number

foreach $(X,Y)$ in Elements do
    if $Y \neq Y_{prev}$ then                                    // start of a new row
        finalize($deltas$, $newrow$)
        $deltas \leftarrow [\,]$
        $newrow \leftarrow$ true
        $Y_{prev} \leftarrow Y$
        $X_{prev} \leftarrow 0$
    $deltas$.add($X - X_{prev}$)
    if $deltas$ reached maximum size then                    // unit finalization check
        finalize($deltas$, $newrow$)
        $deltas \leftarrow [\,]$
        $newrow \leftarrow$ false
```

Algorithm 5.2: Unit finalization: appending appropriate information to `ctl` array.

```
finalize($deltas$, $newrow$):
    set usize equal to the size of the $deltas$ array
    if $newrow$ then
        set new row mark at uflags (nr).
    switch max($deltas$) do
        case $1 .. 2^8$ // 1-byte storage
            set delta's size to $1$ byte at uflags (D8).
            copy values of $deltas$ array as $1$-byte integers to body.
        case $2^8 .. 2^16$ // 2-bytes storage
            set delta's size to $2$ bytes at uflags (D16).
            copy values of $deltas$ array as $2$-bytes integers to body.
        ...
```

The SpMxV implementation for the CSR-DU storage format is presented in Listing 5.1†. Access to the `ctl` array is performed via macros (e.g., `ctl_get_u16()`) that return the appropriate value and advance the array pointer as necessary. Initially, the `uflags` and `usize` header fields are extracted from `ctl`. If the unit belongs to a new row, appropriate initializations are performed: the `y` index is increased and the `x` index is zeroed. Finally, the appropriate multiplication code is executed based on the unit type. The innermost loops implementing the multiplication code for each case do not contain any branches, which allows for fast execution by the processor.

Parallelization is similar to CSR. For the row partitioning scheme, described in Section 4.3.1, each thread needs an offset in the `ctl`, `values` and `y` arrays to mark the beginning of its data, and the total number of rows that have been assigned to it.

The next paragraphs discuss extensions to CSR-DU format for performance improvement.

```
do {
    usize = ctl_get_u8(ctl);
    uflags = ctl_get_u8(ctl);
    if ( flags_new_row(uflags) ){
        y_indx++;
        x_indx = 0;
    }
    switch ( flags_type(uflags)  ){
        case CSR_DU_U8:
        for (i=0; i<usize; i++) {
            x_indx += ctl_get_u8(ctl);
            y[y_indx] += *(values++) * x[x_indx];
        }
        break;

        case CSR_DU_U16:
        for (i=0; i<usize; i++) {
            x_indx += ctl_get_u16(ctl);
            y[y_indx] += *(values++) * x[x_indx];
        }
        break;

        case CSR_DU_U32:
        ...
    }
} while (values < values_end);
```

Listing 5.1: CSR-DU SpMxV implementation.

†Note that the code has been simplified to aid the presentation. For example the optimization for updating the *y* value only at the end of the loop is not shown (see Section 4.2).

5.2.1 Unit offsets

A problem with CSR-DU, as described in the previous paragraphs, is that the unit's first delta value can be significantly larger than the rest, imposing an unnecessarily large storage size for the rest of the deltas. In the example of Figure 5.2 the density of the second unit's elements allows for 1-byte delta values, but the large distance from the first unit dictates 2-byte storage. To counter this problem, we modify the original CSR-DU format to include a column index offset from the previous unit in the header. The offset is called `ujmp` and is stored as a (positive) variable-length integer at the end of the header. This technique improves compression of the column indices at no cost for performance since the change does not affect innermost loops. We implement the variable-length integers using the scheme described in 5.1: each integer is divided in 7-bit parts, which are stored in consecutive bytes; the final byte is distinguished by having its MSB set.

We consider the example of the two units in Figure 5.2. Figure 5.3 shows the encoding of the second unit when unit offsets are used. In this case, unit offsets result in better compression because deltas are stored in 1-byte integers. On the other hand, using unit offsets in the first unit would lead to increased size because the first delta value (200) requires 2-byte storage in our variable-length storage scheme ($1 + \lfloor \frac{\log_2 200}{7} \rfloor = 2$). This issue, however, appears only on a per-unit basis, and selection of sufficiently large units amortizes potential losses. Listing 5.2 shows a portion of the SpMxV implementation for CSR-DU with unit offsets: the offset is added to the x index and the appropriate multiplication code is executed based on the unit type.

5.2.2 Sequential units

Although delta encoding can significantly reduce index data volume, it does not handle the occurrence of sequential elements efficiently. If all unit elements are sequential, column indexing information can be completely omitted. This, in addition to reducing the working set, eliminates indirect accesses on `x` allowing for better optimization from both the compiler and the CPU. Thus, in contrast with typical compression schemes, exploitation of sequential elements not only reduces storage volume but also — potentially — decreases computational overhead.

We extend CSR-DU, in a way similar to the format presented in [PH99], to support units containing sequential elements. An example of this unit type (*sequential units*) is illustrated in Figure 5.4. Besides the `usize` and `uflags` fields, unit data also contain the column index offset from the previous unit as a variable length integer. Note that if unit offsets are used, the last field of the unit coincides with `ujmp`. Listing 5.3 shows the multiplication code for sequential units.

An important parameter that needs to be considered during the compression phase is the minimum possible size for the sequential units. We will refer to this parameter as *seq*. Consecutive elements of size less than *seq* will be encoded using delta encoding as described in previous sections. Tuning of this parameter prevents performance degradation from sequential units with small size. For example, if *seq*=1 then all units of the matrix will be encoded as sequential. This will result in poor performance if the matrix does not contain enough sequential elements. In general, the effect of *seq* on SpMxV performance depends on: (a) the architecture of the execution platform and (b) the structure of the matrix (e.g., frequency of sequential units).

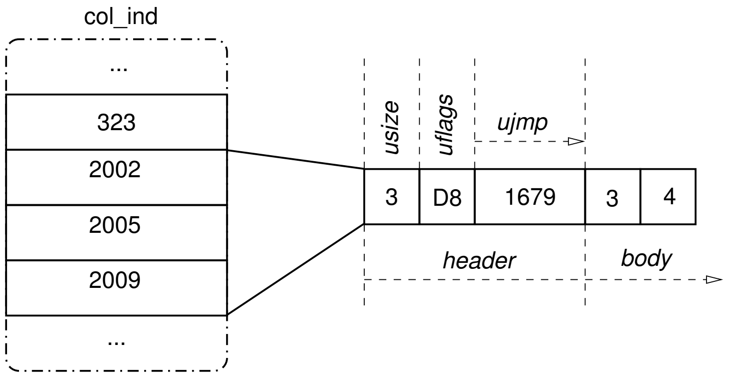


Figure 5.3: Example of a CSR-DU unit with offsets.

```
...
x_indx += ctl_get_varint(ctl); // unit offset (variable-length)
switch ( flags_type(uflags)  ){
    case CSR_DU_U8:
    y[y_indx] += *(values++) * x[x_indx]; // first element
    for (i=1; i<usize; i++) {
        x_indx += ctl_get_u8(ctl);
        y[y_indx] += *(values++) * x[x_indx];
    }
    break;

    case CSR_DU_U16:
    y[y_indx] += *(values++) * x[x_indx]; // first element
    for (i=1; i<usize; i++) {
        x_indx += ctl_get_u16(ctl);
        y[y_indx] += *(values++) * x[x_indx];
    }
    break;

    case CSR_DU_U32:
    ...
}
```

Listing 5.2: portion of the SpMxV implementation for CSR-DU with unit offsets.

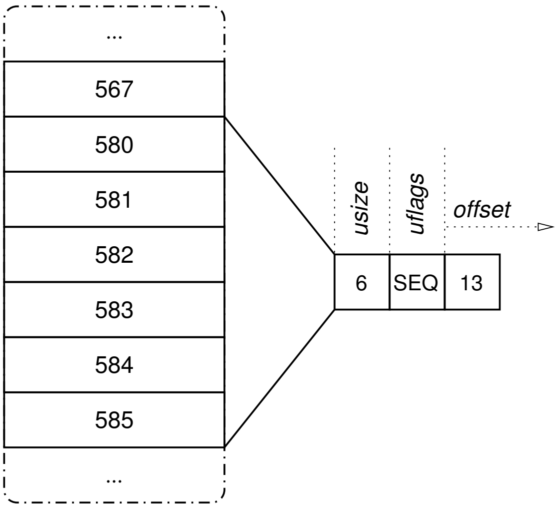


Figure 5.4: Example of sequential elements unit.

```
...
x_indx += ctl_get_varint(ctl);
switch ( flags_type(uflags)  ){
    case CSR_DU_SEQ:
    for (i=0; i < size; i++){
        y[y_indx] += *(values++) * x[x_indx + i]
    }
    x_indx += (size-1)
    ...
}
...
```

Listing 5.3: multpilication code for CSR-DU sequential units.

5.2.3 Alignment of `ctl` array values

Another issue with the CSR-DU format is that packing of delta values larger than 1 byte in the `ctl` array may lead to unaligned storage. For example in the case of Figure 5.2 if the first field of the `ctl` array is aligned then the three 16-bit deltas in the second unit are stored in an unaligned manner. Some ISAs disallow unaligned access. Others (e.g., the `x86` and `x86_64` ISAs) include instructions that allow unaligned access, but may result in performance degradation. In our implementation, we pad the `ucis` sections in the `ctl` array, so that the accesses of delta indices are always performed in an aligned manner.

5.3 Performance evaluation

5.3.1 Experimental setup

To evaluate CSR-DU, we performed experimental runs using several different combinations for the method's parameters. Versions with aligned deltas and unit offsets performed better or similar than the rest, and so we present only them in the following results. Regarding sequential units, we consider three cases: absence of sequential units (*noseq*) and sequential units with a minimum of 8 (*seq*=8) and 4 (*seq*=4) elements. It should be noted that *seq*=4 performs more aggressive compression than *seq*=8.

We compare CSR-DU performance against both CSR and BCSR. For the BCSR method we performed experiments with a number of different block shapes configurations‡, using specialized SpMxV versions (i.e., as the one in Listing 3.3). In the following results we demonstrate best performing BCSR case over all available block shape configurations. Our setup is otherwise similar to the one described in Section 4.5.1: we perform our experiments on two systems (Harpertown and Nehalem), using a suite of 50 matrices (Table 4.3).

5.3.2 Results

Size reduction

Table 5.2 lists the compression ratios achieved for each matrix in our suite. The `large-dense` matrix maximizes CSR-DU size reduction resulting in 24.9% and 33.2% ratios for *noseq* and sequential units, respectively. Compression ratios on other matrices vary largely, ranging from zero (`Freescale1`) to close to maximum (`TSOPF_RS_b2383`). On average, CSR-DU reduces matrix data by 14.2% for *noseq*, 19.3% for *seq*=8 and 21.1% for *seq*=4. BCSR, on the other hand, is not able to efficiently capture the structure of matrices in our suite since it increases the size of 28 matrices. BCSR averages a 13.2% size increase over all matrices and a 16.1% size decrease over matrices that effectively compresses (22 matrices). Moreover, only for 2 matrices (`F1`, `thermomech_dK`) BCSR achieves better reduction than CSR-DU *seq*=4. As these results illustrate, CSR-DU is more stable than BCSR since by design it does not increase matrix size at any case — at worst size will remain unaffected.

Harpertown

First, we discuss results on Harpertown. Figure 5.5 shows the average speedup of CSR, BCSR, and CSR-DU over single-threaded CSR, for different thread affinity configurations. BCSR performs worse than CSR on average for all cases, a result of the large number of matrices for which BCSR increases their size. When all cores are utilized, CSR-DU methods perform better on average than CSR and BCSR. The *seq*=4 case achieves the best average speedup for 8 threads (2.45), improving performance by 28.7% and 35.0% over CSR and BCSR average, respectively. An interesting aspect of CSR-DU variants performance is that the best version for 8 threads (*seq*=4) has the lowest performance in the serial case (7% slowdown compared to CSR). The latter exem-

‡the block shapes considered were: $1 \times 2, 1 \times 3, 1 \times 4, 2 \times 1, 2 \times 2, 2 \times 3, 2 \times 4, 3 \times 1, 3 \times 2, 4 \times 1, 4 \times 2$

matrix name	size reduction (%)				matrix name	size reduction (%)			
	BCSR	DU				BCSR	DU		
	(max)	noseq	seq=8	seq=4		(max)	noseq	seq=8	seq=4
boneS10	22.5	16.6	27.1	30.0	G3_circuit	-27.1	9.3	9.3	9.3
ldoor	9.5	6.9	19.9	30.1	cage13	-56.2	11.1	11.1	11.1
inline_1	22.4	4.7	15.8	22.8	rajat30	-31.6	9.9	10.7	14.0
fdif202x202x102	-38.5	15.9	15.9	15.9	pre2	-38.5	14.1	14.6	14.8
F1	22.3	5.8	16.2	21.0	Hamrle3	-6.2	17.8	17.8	19.6
rajat31	-39.3	21.5	21.5	21.5	largebasis	15.7	0.7	19.2	21.8
msdoor	10.2	11.9	22.3	29.1	Chebyshev4	-28.3	20.8	26.0	26.0
Freescale1	-31.7	0.5	0.5	0.6	apache2	-37.7	15.9	15.9	15.9
Ga41As41H72	-16.9	16.6	21.7	25.1	s3dkq4m2	16.7	16.6	31.8	31.8
af_shell9	12.0	16.6	28.8	30.9	ship_001	6.2	16.8	28.7	31.2
af_5_k101	12.1	16.5	28.8	30.9	torso3	-25.4	15.3	15.3	15.3
TSOPF_RS_b2383	26.3	21.0	33.1	33.1	thread	22.3	17.1	26.5	28.2
kkt_power	-61.2	5.2	5.2	5.2	ASIC_680k	-32.6	7.5	9.1	10.9
Si41Ge41H72	-13.5	16.6	21.8	25.7	large-dense	29.2	24.9	33.2	33.2
random100000	-66.3	16.7	16.7	16.7	barrier2-9	-37.6	16.8	16.8	17.3
nd12k	16.4	16.7	29.3	30.0	xenon2	19.4	21.0	21.0	21.8
crankseg_2	5.8	16.8	25.8	28.7	parabolic_fem	-46.9	1.0	1.0	1.0
pwtk	14.1	15.7	31.3	31.6	FEM_3D_thermal2	-13.7	19.3	19.3	19.3
bmw3_2	7.4	17.5	25.9	30.0	sme3Dc	-58.0	16.8	16.8	16.8
ohne2	-24.8	16.7	17.8	20.0	stomach	-29.7	21.5	21.5	21.5
hood	10.4	11.3	22.5	29.3	thermomech_dK	25.6	12.8	12.8	13.1
Si87H76	-29.6	16.6	20.4	22.7	helm2d03	-52.1	1.4	1.4	3.8
bmwcra_1	22.4	16.7	25.7	28.4	ASIC_680ks	-31.7	17.8	20.3	22.3
atmosmodj	-38.4	16.0	16.0	16.0	poisson3Db	-60.0	17.1	17.1	17.1
thermal2	-42.5	12.5	12.5	13.3	rma10	5.3	19.1	26.1	29.4

Table 5.2: Size reduction achieved by BCSR and CSR-DU compared to CSR. For BCSR we select the block shape that achieved maximum size reduction.

plifies the negative effect of decompression computational overhead when the system's memory bandwidth is adequate to perform SpMxV without memory stalls.

Figure 5.8 shows the performance improvement of BCSR and CSR-DU over CSR for individual matrices, when all 8 cores are utilized. An important observation is that the CSR-DU method does not exhibit significantly reduced performance over CSR for any matrix in our suite. For BCSR, however, a significant number of matrices take a performance hit compared to CSR, due to a significant size increase.

Figure 5.7 shows the association of size reduction and SpMxV performance improvement for BCSR and CSR-DU. Since BCSR is computationally more efficient than CSR (e.g., by applying register blocking), it improves performance even for the serial case on fitting matrices, i.e. matrices whose size is reduced (Figure 5.7a). The additional benefit of alleviating memory bandwidth pressure via size reduction further increases performance improvement when all 8 cores are utilized (Figure 5.7c). On the other hand, CSR-DU imposes a substantial computational overhead that restrains performance improvement on the serial case (Figure 5.7b). When multiple threads are utilized, however, the overhead is amortized and performance significantly improves (Figure 5.7d).

Selecting the optimal storage format is not a straightforward task, since performance depends on the system architecture and the matrix structure in ways that are not always visible nor simple. Figure 5.6 depicts a breakdown of the best performing methods distribution for our matrix suite. Concentrating on the full core utilization case for Harpertown, CSR-DU is a good universal choice for our suite, since it achieves best performance for 43 matrices. For the same case, BCSR achieves the best performance for 4 matrices and CSR for 3. When only one thread is utilized, the best performing methods distribution is more balanced: CSR achieves best performance for 17 matrices, BCSR for 15 and CSR-DU for 18. In general, we argue that as long as the main bottleneck is memory bandwidth, CSR-DU is a more promising option than BCSR and CSR. When the memory bandwidth bottleneck becomes less severe, the computation overhead imposed by decompression is not amortized, making CSR-DU less attractive.

Nehalem

Next, we present experimental results on the Nehalem system, where we only consider NUMA-aware versions for the methods discussed. Figure 5.9 demonstrates the average speedup of considered methods over serial CSR. When all cores are utilized (`4c×2d`), the best average speedup for CSR-DU is 4.6 — a 12.2% and 11.7% improvement over CSR and BCSR averages respectively — and is achieved by *seq*=8. Hence, even for the Nehalem system, where the memory bottleneck is less severe, there is a margin for performance improvement by applying CSR-DU. Interestingly, when all SMT threads are utilized (`2t×4c×2d`), CSR-DU average for *seq*=4 and *seq*=8 is improved; all other methods result in a slowdown.

Figure 5.12 presents BCSR and CSR-DU improvement over CSR for each matrix in our suite when all threads are utilized (`2t×4c×2d`). CSR-DU performs worse than CSR for only five matrices (`helm2d03`, `sme3Dc`, `parabolic_fem`, `G3_circuit`, `Freescale1`) and in some cases the difference is marginal. Thus, we argue that, CSR-DU is fairly stable, even when the memory bottleneck is alleviated by the system's architectural characteristics.

The effect of size reduction on performance improvement for Nehalem (Figure 5.11) is qualitatively analogous to the one for Harpertown. The large memory bandwidth of Nehalem differentiates results by restraining the benefits from compression even when all threads are utilized. Hence, a notable number of points in the CSR-DU graph for 16 threads (Figure 5.11d) are below the *y* axis, i.e. they perform worse than CSR.

Figure 5.10 illustrates the distribution of best performing methods. For a single thread, CSR achieves the best performance for 19 matrices, BCSR for 22 and CSR-DU for 9. However, as the number of cores utilized increases, CSR-DU becomes the most attractive option: for `4c×2d` CSR-DU performs best for 36 matrices, BCSR for 9 and CSR for 5.

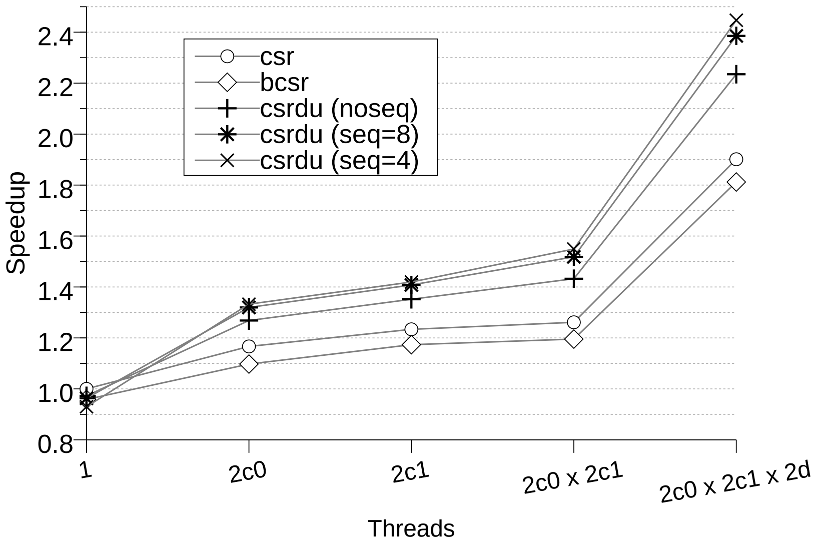


Figure 5.5: Average CSR, BCSR and CSR-DU parallel speedup over serial CSR on Harpertown. BCSR average is derived from best the performing case for each matrix over all considered block shapes.

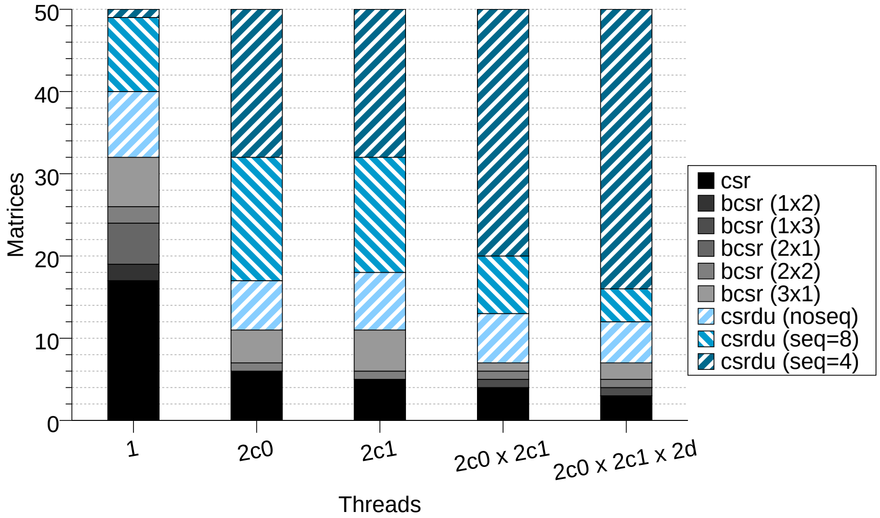
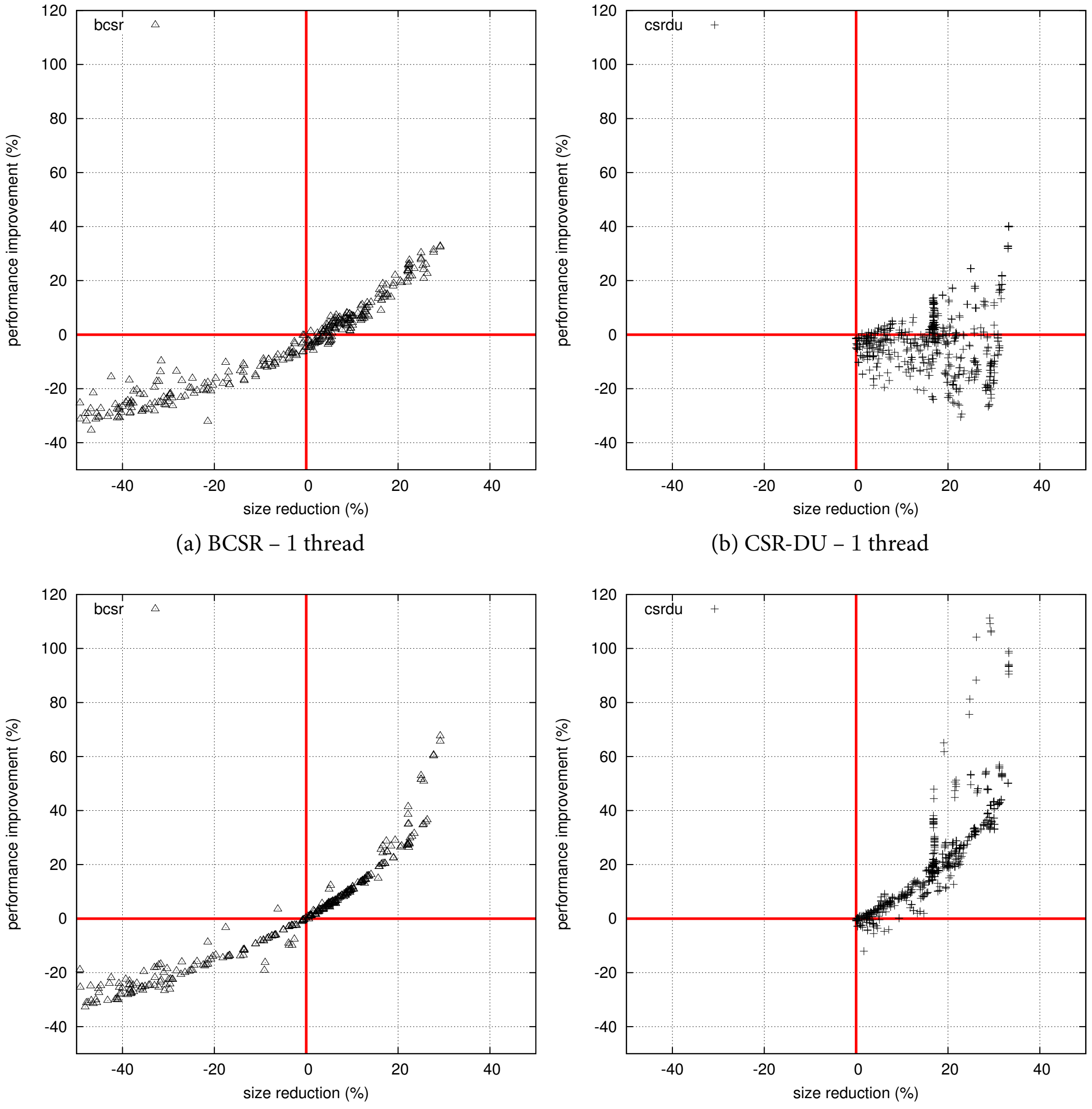


Figure 5.6: Distribution of best performing methods on Harpertown.



(a) BCSR – 1 thread

(b) CSR-DU – 1 thread

(c) BCSR – 8 threads (`2c0`×`2c1`×`2d`)

(d) CSR-DU – 8 threads (`2c0`×`2c1`×`2d`)

Figure 5.7: Relation of size reduction and performance improvement of BCSR and CSR-DU over CSR for 1 and 8 threads on Harpertown. We consider all matrices in our suite and different variants for each method. Specifically, for BCSR we consider all possible block configurations, while for CSR-DU we consider versions with and without unit offsets, with and without aligned deltas and with and without sequential units (*seq*=4, *seq*=8, *noseq*).

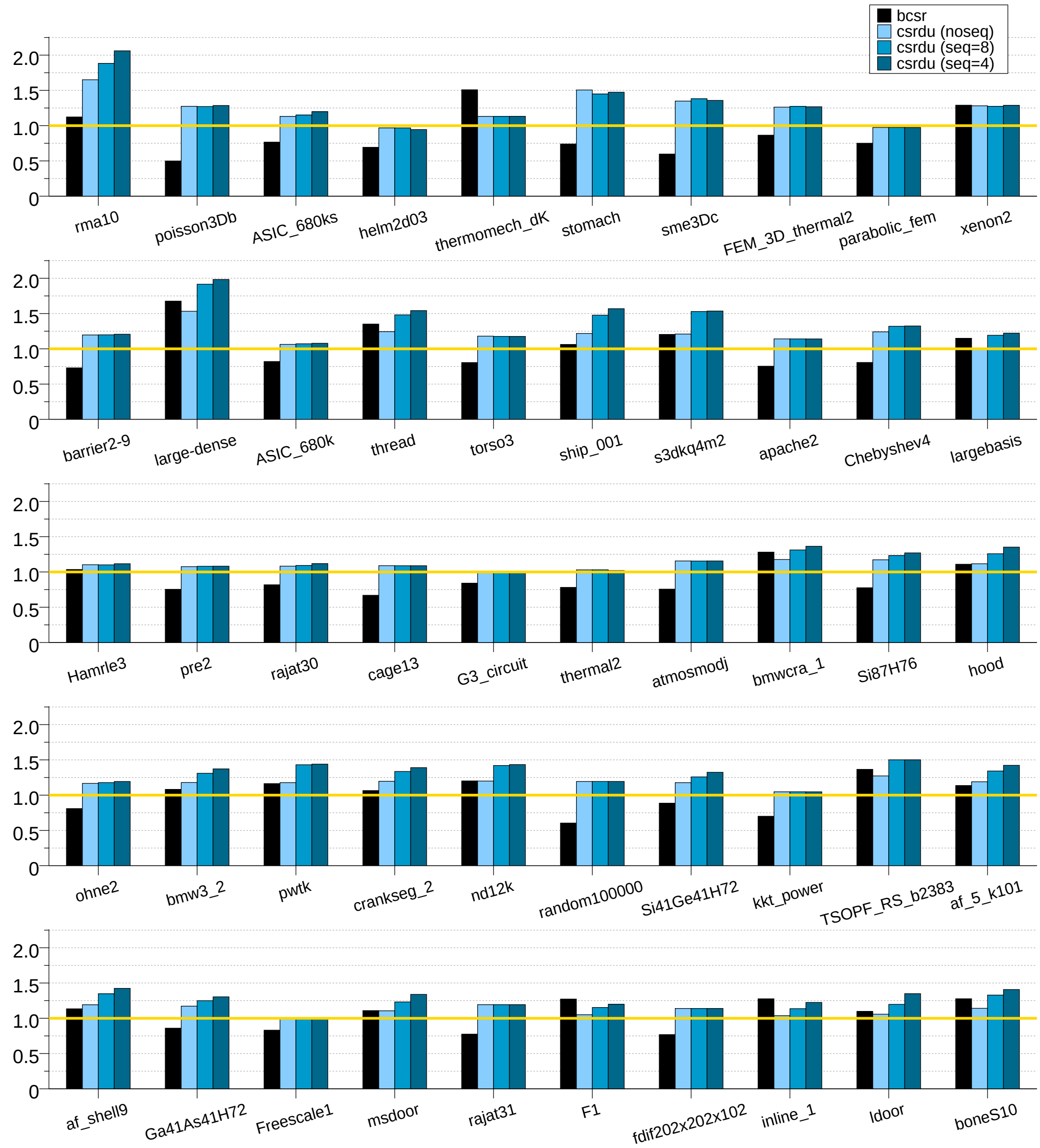


Figure 5.8: Performance improvement of BCSR and CSR-DU over CSR for individual matrices when all 8 Harpertown cores are utilized. We use the best performing block shape for BCSR.

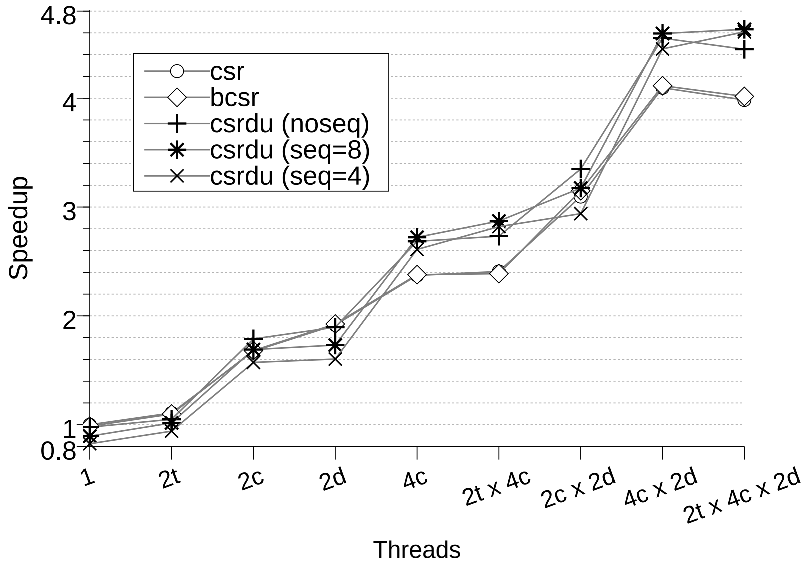


Figure 5.9: Average CSR, BCSR and CSR-DU parallel speedup over serial CSR on Nehalem. BCSR average is derived from best the performing case for each matrix over all considered block shapes.

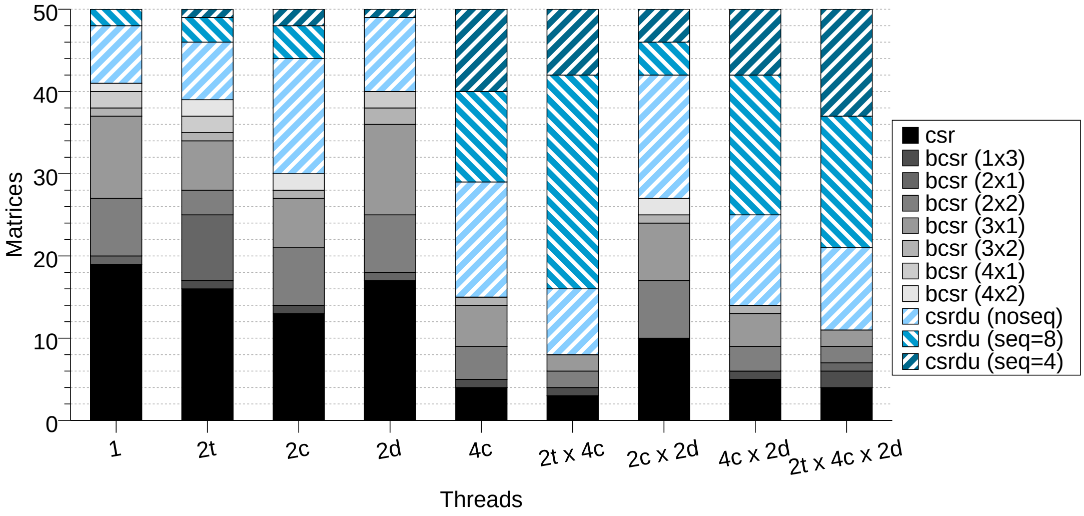
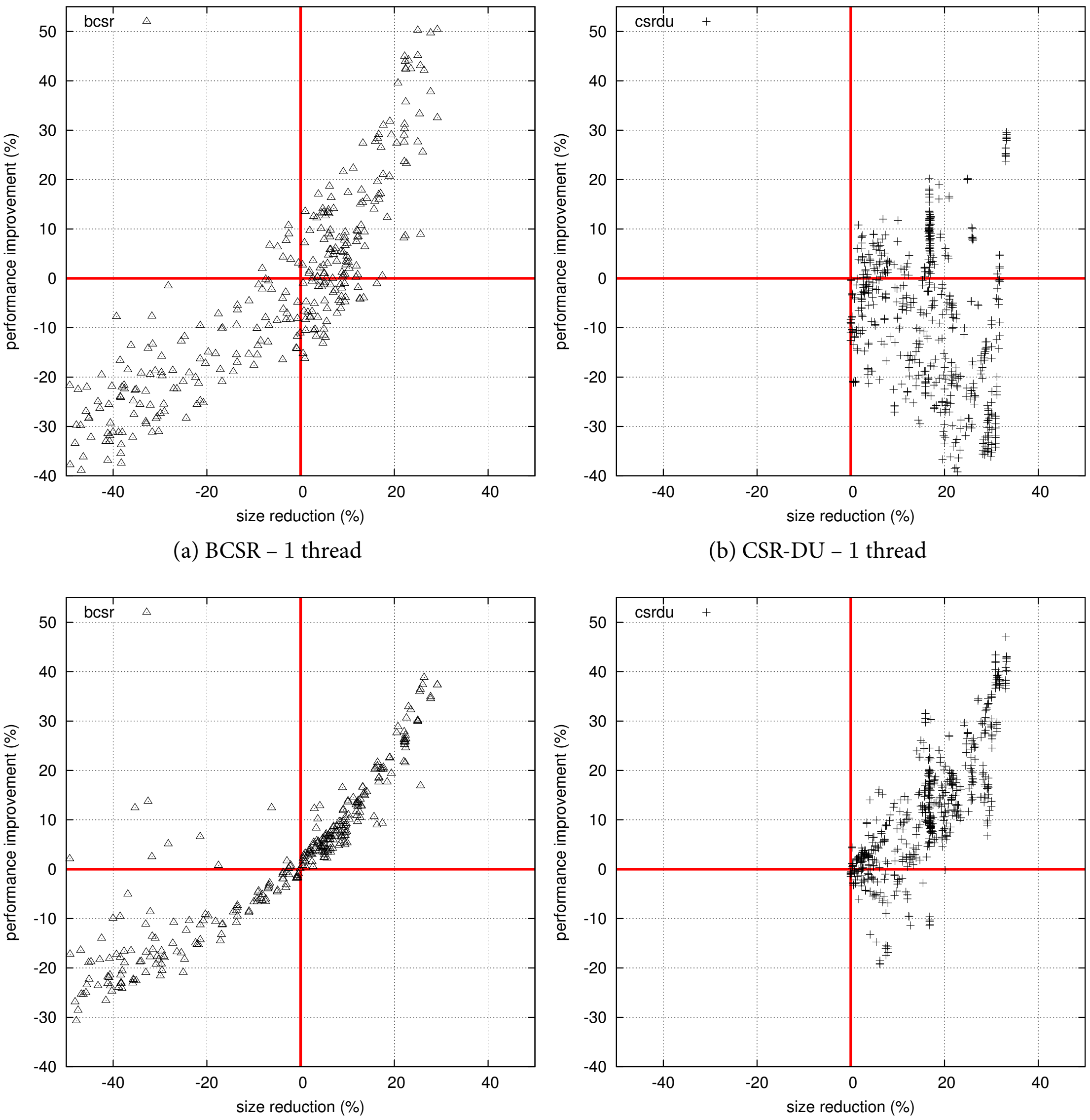


Figure 5.10: Distribution of best performing methods on Nehalem.



(a) BCSR – 1 thread

(b) CSR-DU – 1 thread

(c) BCSR – 16 threads (2t×4c×2d)

(d) CSR-DU – 16 threads (2t×4c×2d)

Figure 5.11: Relation of size reduction and performance improvement of BCSR and CSR-DU over CSR for 1 and 16 threads on Nehalem. We consider all matrices in our suite and different variants for each method. Specifically, for BCSR we consider all possible block configurations, while for CSR-DU we consider versions with and without unit offsets, with and without aligned deltas and with and without sequential units (*seq*=4, *seq*=8, *noseq*).

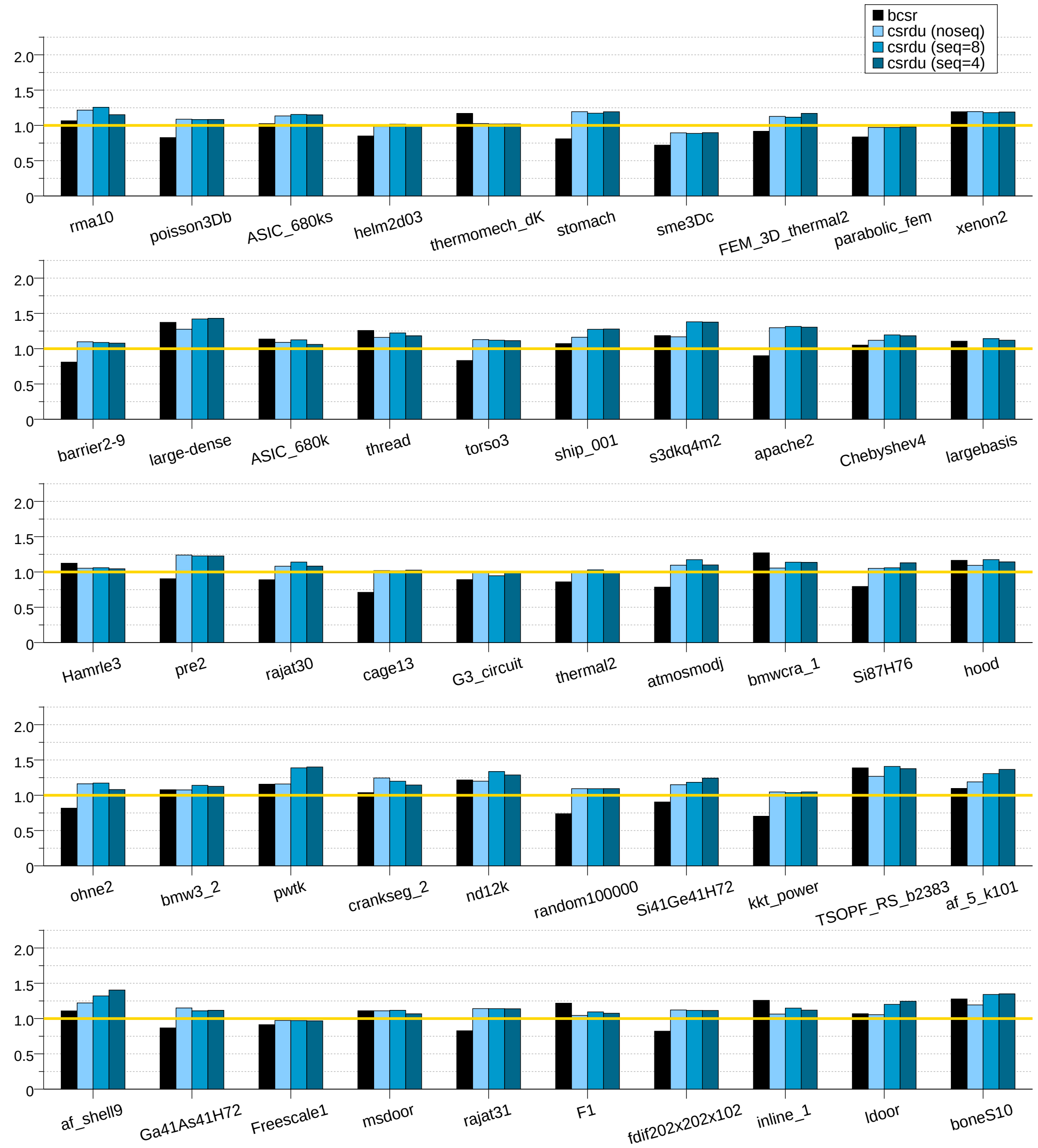


Figure 5.12: Performance improvement of BCSR and CSR-DU over CSR for individual matrices when all 16 Nehalem threads are utilized. We use the best performing block shape for BCSR.

5.4 Related work

A significant part of the SpMxV optimization techniques reported in the related literature result in index data reduction. Typical examples are blocking methods such as BCSR that store only per-block index information. Traditionally the main focus of BCSR has been serial performance improvement (e.g., via register blocking) and not working set reduction. For this reason, BCSR has several disadvantages if used as a compression technique. First and foremost, depending on the matrix structure, it may increase the total size of the matrix due to padding (see Table 4.3). Secondly, it relies on constant-shape blocks, which limits its capability to adapt to more complex matrix structures.

Pinar and Heath [PH99] describe a one-dimensional variable block scheme, similar to the one used for CSR-DU sequential units. They also discuss column reordering techniques that aim to align the non-zero elements of a row in consecutive locations as much as possible. CSR-DU could benefit from similar reordering techniques towards two directions: (a) creating larger sequential units and (b) creating denser units that require smaller delta values. Although not strictly equivalent, the latter is strongly related with matrix bandwidth reduction techniques than have been extensively studied in the past because they relate to SpMxV cache performance [TJ92,PHCR04].

One of the few works that explicitly targets the compression of the index data is [WL06]. In this paper, Willcock and Lumsdaine propose two methods: *DCSR*, which compresses column indices using a byte-oriented delta encoding scheme to exploit the highly redundant nature of the `col_ind` array and *RPCSR*, which generates matrix-specific dynamic code by applying aggressive compression on column indices patterns for the whole matrix. We will focus our comparison on the DCSR method, which operates on the same level as CSR-DU. DCSR encodes the matrix using a set of six command codes for primitive sub-operations that can be used to implement the SpMxV kernel. Examples of such sub-operations are the increment of the current row and column index, and the multiplication of a number of the matrix values with the appropriate vector elements. A significant performance problem of this approach is that the decoding of these sub-operations must be performed very often, which results in frequent mispredicted branches. This problem is dealt by a form of unrolling where patterns of frequent instances of six of these sub-operations are grouped together allowing them to be executed sequentially, i.e., without branches. Contrarily, our approach, which is also based on delta encoding, tackles the problem of branch misprediction performance penalties in a more basic level by being more coarse-grained. This allows for a much simpler and general implementation, while sustaining a small performance gain gap compared to the DCSR method. Moreover, it can handle worst-case scenarios of the DCSR method such as matrices that exhibit large variation with regard to the patterns encountered.

Another recent work that targets performance improvement by reducing the index data volume is [BBR09], which proposes a matrix representation that exploits repeated block patterns. The authors search for frequently met block patterns and generate specialized inner loops for those, on top of a dispatch logic. They provide an evaluation of a parallel version, but they focus primarily on serial performance.

6 Value compression using Indirect Accesses

6.1 Motivation and general approach

As mentioned in Section 4.2.2, values typically constitute the larger part of the working set of a CSR sparse matrix because they require 64-bit storage. Hence, value compression is potentially more beneficial than index compression in terms of working set reduction. Conversely with index data, value data do not inherently contain redundancy in the general case. Moreover, the (lossless) compression of floating point values is not as straightforward as integers, because floating point arithmetic operations produce rounded results.

Nevertheless, we have noticed that a significant number of matrices from our experimental set contain a small number of unique values, relative to the total non-zero values (*nnz*). From an information theory perspective this results in low entropy for the value data, making them a good target for compression. Since we aim at a computationally light decompression scheme we follow a simple approach: instead of storing all the *nnz* values, we store only the common values and pointers to them. If the total-to-unique values ratio is high enough, the working set data volume will be reduced. Adequate size reduction can lead to SpMxV execution time decrease, despite the overhead induced by indirectly accessing each value. In other words, our approach applies a transformation in the value data, where a large number of numeric values is replaced with the same number of indices and a much smaller number of values. Storing individual indices using less space than individual values leads to data volume reduction. Next, we describe the specifics of our proposed format, called CSR-VI.

6.2 The CSR-VI storage format

The CSR-VI (CSR with Values Indirect) format replaces the CSR `values` array with two arrays: `vals_unique` and `val_ind`. The first contains only the unique matrix values. The second contains indices in the `vals_unique` array for each of the *nnz* matrix elements. To achieve working set size reduction, `val_ind` size must be significantly smaller than `values` size. A simple approach towards this goal is to reduce the storage requirements of individual value indices compared to the storage requirements of original values. Hence, in CSR-VI the value index size is determined by the number of the unique values that need to be addressed. For example, if there exist uv unique values and $2^8 < uv \leq 2^{16}$, then a 2-byte integer will be used for each value index. Although this approach does not optimally compresses value indices, it introduces a small overhead because it does not add any branches. An example of this value structure is presented in Figure 6.1, which contains the values of the matrix shown in Figure 3.3.

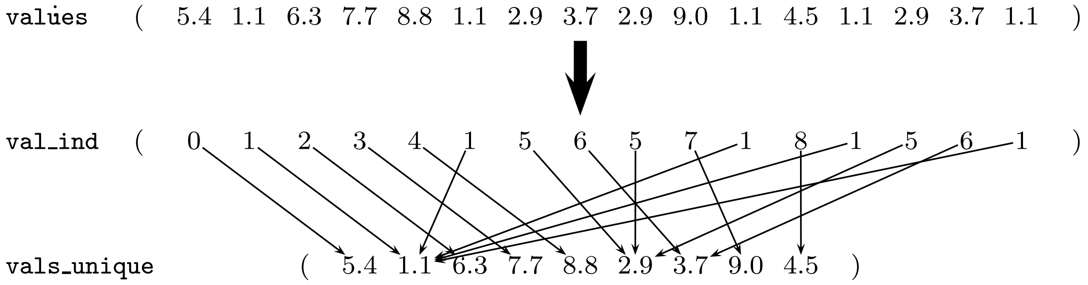

values (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)
val_ind (0 1 2 3 4 1 5 6 5 7 1 8 1 5 6 1)
vals_unique (5.4 1.1 6.3 7.7 8.8 2.9 3.7 9.0 4.5)

Figure 6.1: Example of the value indexing structure for the CSR-VI format.

CSR-VI is a specialized method, i.e., it can be meaningfully applied only to matrices with a large number of common values. To elaborate on the method's applicability for a given matrix, we define the *total-to-unique* (ttu) values ratio, as the fraction of the number of non-zero elements (*nnz*) to the number of unique values (uv):

$$ttu = \frac{nnz}{uv}$$

A high ttu value indicates that the matrix is fitting for the CSR-VI method, while a small one shows that CSR-VI will most likely result in slowdown. Using ttu and the storage size of value indices (s_{vi}) we can express the reduction on value data (κ) when CSR-VI is applied:

$$\kappa = 1 - \frac{\mathcal{V}_{csrvi}}{\mathcal{V}_{csr}} = 1 - \frac{uv \cdot s_{val} + nnz \cdot s_{vi}}{nnz \cdot s_{val}} = 1 - \left(\frac{uv}{nnz} + \frac{s_{vi}}{s_{val}}\right) = 1 - \left(\frac{1}{ttu} + \frac{s_{vi}}{s_{val}}\right)$$

If we store individual value indices as integers with the minimum possible size, as discussed previously, we can express their storage size s_{vi} as:

$$s_{vi} = \begin{cases} 1\,byte, & uv \leq 2^8 & = 256 \\ 2\,bytes, & 2^8 < uv \leq 2^{16} & = 65,536 \\ 4\,bytes, & 2^{16} < uv \leq 2^{32} & = 4,294,967,296 \end{cases}$$

It is possible to further reduce the storage volume of value indices by using more elaborate techniques. For instance, instead of using standard integers we could encode value indices using only the necessary number of bytes ($\lceil \frac{1+\lfloor \log_2 uv \rfloor}{8} \rceil$), or only the necessary number of bits ($1 + \lfloor \log_2 uv \rfloor$). These techniques, however, are more complex and induce additional decompression overhead.

CSR-VI compression can be implemented using a hash table and, as in CSR-DU, its complexity is $\mathcal{O}(nnz)$. Algorithm 6.1 shows the compression procedure. The SpMxV kernel implementation for CSR-VI is shown in Listing 6.1. It can be easily derived from the CSR case by replacing the access of `values` with an indirect access of `vals_unique` based on the value of `val_ind`. Even though the resulting code contains an additional memory reference for each of the *nnz* elements, it will lead to fewer data being transferred from memory when the number of unique values is relatively small. Nevertheless, working set reduction alone is not sufficient to achieve performance improvement; the additional overhead of indirect accesses must be amortized.

Algorithm 6.1: CSR-VI compression procedure.

```
Initialization:
  $h$    $\leftarrow \{\,\}$  // hash table for storing unique values
  $vis$  $\leftarrow [\,]$   // array of value indices
  $uvs$  $\leftarrow [\,]$   // array of unique values
foreach $V$ in $Values$ do
    if $V$ not in $h$ then
        $vi \leftarrow uvs$.size()                      // get (new) value index
        $uvs$.add($V$)                 // add current value to unique values
        $h[Val] \leftarrow vi$            // store value index in hash table
    else
        $vi \leftarrow h[Val]$           // restore value index from hash table
    $vis$.add($vi$)                              // add value index in array
```

```
for(i=0; i<N; i++)
    for(j=row_ptr[i]; j<row_ptr[i+1]; j++)
        y[i] += vals_unique[val_ind[j]] * x[col_ind[j]];
```

Listing 6.1: CSR-VI SpMxV implementation.

6.3 Combining CSR-DU and CSR-VI

CSR-DU and CSR-VI can be applied independently, because they operate on different data sets: CSR-DU is concerned with index data, while CSR-VI with matrix numerical values. We will refer to their combination as CSR-DUVI, a storage format that applies compression to both index and value data. Obviously, CSR-DUVI is not a general format, but can only be applied to matrices with a small number of unique values. The CSR-DUVI SpMxV kernel implementation is shown in Listing 6.2.

```
usize = ctl_get_u8(ctl);
uflags = ctl_get_u8(ctl);
if ( flags_new_row(uflags) ){
   y_indx++;
   x_indx = 0;
}
switch ( flags_type(uflags)  ){
   case CSR_DU_U8:
   for (i=0; i<usize; i++) {
      x_indx += ctl_get_u8(ctl);
      v_indx = *(val_ind++);
      y[y_indx] +=  vals_unique[v_indx] * x[x_indx];
   }
   break;

   case CSR_DU_U16:
   for (i=0; i<usize; i++) {
      x_indx += ctl_get_u16(ctl);
      v_indx = *(val_ind++);
      y[y_indx] +=  vals_unique[v_indx] * x[x_indx];
   }
   break;

   ...
}
```

Listing 6.2: SpMxV code for CSR-DUVI.

6.4 Performance evaluation

6.4.1 Experimental setup

Our experimental setup is similar to the one described in Section 4.5.1: we use two multicore systems (Harpertown and Nehalem), 32-bit indices and 64-bit values, and a suite of 50 matrices (see Table 4.3) as a starting point. Table 6.1 lists the number of unique values and the ttu ratio for each matrix. Since not all matrices are suitable for our method, we refine our set using the empirical criterion $ttu \geq 5$. Moreover, we discard matrices `random100000` and `large-dense` because they have randomly created values. The resulting subset has 22 matrices, which is a significant portion of the original set. For NUMA-aware methods on the Nehalem system, data shared between threads, e.g., the unique values array, are allocated using standard mechanisms (i.e., `malloc()`).

matrix characteristics			**size reduction (%)**			
name	**uvals**	**ttu**	VI	DUVI		
				noseq	seq=8	seq=4
boneS10	40	1,386,710.6	58.0	74.6	85.2	88.1
ldoor	21,675,099	2.1	-	-	-	-
inline_1	18,016,122	2.0	-	-	-	-
fdif202x202x102	4	6,960,000.0	55.7	71.6	71.6	71.6
F1	13,038,962	2.1	-	-	-	-
rajat31	3,985	5,098.2	46.4	67.9	67.9	67.9
msdoor	9,777,773	2.1	-	-	-	-
Freescale1	9,418,239	2.0	-	-	-	-
Ga41As41H72	3,597,854	5.1	20.3	36.9	41.9	45.4
af_shell9	968,711	18.2	29.4	45.9	58.1	60.3
af_5_k101	9,027,150	1.9	-	-	-	-
TSOPF_RS_b2383	762,680	21.2	30.2	51.1	63.2	63.2
kkt_power	84,245	173.5	31.5	36.7	36.7	36.7
Si41Ge41H72	4,665,454	3.2	-	-	-	-
random100000	10,000	1,497.8	-	-	-	-
nd12k	4,857,071	2.9	-	-	-	-
crankseg_2	4,397,887	3.2	-	-	-	-
pwtk	5,592,868	2.1	-	-	-	-
bmw3_2	4,126,650	2.7	-	-	-	-
ohne2	5,271,361	2.1	-	-	-	-
hood	5,048,077	2.1	-	-	-	-
Si87H76	334,180	31.9	31.0	47.6	51.4	53.7
bmwcra_1	3,153,346	3.4	-	-	-	-
atmosmodj	4	2,203,720.0	55.7	71.6	71.6	71.6
thermal2	4,819,424	1.8	-	-	-	-
G3_circuit	241	31,787.7	54.6	63.9	63.9	63.9
cage13	417	17,936.1	49.0	60.1	60.1	60.1
rajat30	683,418	9.0	25.1	35.0	35.8	39.1
pre2	781,486	7.6	23.7	37.8	38.3	38.5
Hamrle3	53	104,042.3	53.6	71.5	71.5	73.2
largebasis	317	17,539.7	48.7	49.4	67.9	70.6
Chebyshev4	1,550,644	3.5	-	-	-	-
apache2	40	120,446.8	55.6	71.5	71.5	71.5
s3dkq4m2	74,283	64.9	32.1	48.7	63.9	63.9
ship_001	1,209,604	3.8	-	-	-	-
torso3	3,121,632	1.4	-	-	-	-
thread	2,085,970	2.1	-	-	-	-
ASIC_680k	80,211	48.3	30.2	37.7	39.3	41.1
large-dense	32,767	122.1	-	-	-	-
barrier2-9	1,095,875	3.6	-	-	-	-
xenon2	93,364	41.4	31.3	52.3	52.3	53.1
parabolic_fem	259,125	14.2	27.3	28.3	28.3	28.3
FEM_3D_thermal2	1,880,768	1.9	-	-	-	-
sme3Dc	2,358,393	1.3	-	-	-	-
stomach	2,257,584	1.3	-	-	-	-
thermomech_dK	1,967,432	1.4	-	-	-	-
helm2d03	109,526	25.0	29.3	30.7	30.7	33.1
ASIC_680ks	40,708	57.2	44.5	62.3	64.7	66.8
poisson3Db	2,374,908	1.0	-	-	-	-
rma10	1,223,223	1.9	-	-	-	-

Table 6.1: Size reduction of CSR-VI and CSR-DUVI over CSR. The columns of the table have the following meaning: *uvals* is the number of unique values in the matrix, *ttu* is the total-to-unique values ratio of the matrix and columns *VI* and *DUVI* show the size reduction achieved by methods CSR-VI and CSR-DUVI respectively.

6.4.2 Size reduction

The achieved compression of CSR-VI and CSR-DUVI over CSR for the selected matrices is shown in Table 6.1. CSR-VI achieves an average reduction of 39.2%, the maximum and the minimum being 58.8% (`boneS10`) and 20.3% (`Ga41As41H72`), respectively. Combining CSR-VI and CSR-DU further decreases matrix data volume: CSR-DUVI averages a size reduction of 52.4% for *noseq*, 56.2% for *seq*=8 and 57.3% for *seq*=4.

6.4.3 CSR-VI

First, we present SpMxV performance results for the Harpertown system. Figure 6.2 shows the average parallel speedup of CSR and CSR-VI over serial CSR. As expected, performance gain from value compression is larger than index compression, since we consider 32-bit indices and 64-bit values for our reference CSR implementation. Even in the serial case, CSR-VI achieves a 12.4% performance improvement over CSR. As the number of utilized cores increases, memory bandwidth bottleneck becomes more intense and working set reduction becomes more beneficial. For 8 threads the average CSR-VI speedup is 2.75, which is a 51.7% improvement over the corresponding CSR case. Figure 6.3 shows the performance improvement of CSR-VI over CSR for each individual matrix, when all 8 cores are utilized. CSR-VI leads to reduced performance for only `Ga41As41H72`, which is the matrix with the lower *ttu* value (5.1) in our suite.

Next, we discuss CSR-VI results on the Nehalem system. The ample memory throughput capabilities of Nehalem limit the potential CSR-VI benefits. CSR is able to utilize a large portion of these capabilities due to hardware prefetching. This technique, employed by modern processors, detects easily-predicted memory access patterns (e.g., sequential) and prefetches successive data into the cache hierarchy. Contrarily, CSR-VI performs random accesses on the `vals_unique` array; these accesses cannot be predicted, leading to increased memory latencies.

As can be seen in Figure 6.4, CSR-VI performs worse than CSR in the serial case (slowdown of 15%). In the `4c`×`2d` case, however, CSR-VI reaches a speedup of 4.13, which is a 8.6% improvement over the corresponding CSR average speedup. When all available SMT threads at each core are utilized (`2t`×`4c`×`2d`) CSR-VI average is increased to 4.23. For `2t`×`4c`×`2d`, CSR-VI performs worse than CSR for 6 matrices (Figure 6.5).

6.4.4 CSR-DUVI

Figure 6.6 shows that, on average, serial CSR-DUVI SpMxV performance on Harpertown is similar to CSR. In the `2c0`×`2c1`×`2d` case, however, CSR-DUVI results in a significant parallel speedup increase. More specifically, *seq*=8 achieves a speedup of 4.04, which improves upon CSR and CSR-VI by 123% and 47%, respectively. Evidently, part of this large improvement is due to matrices which now fit, in whole or in a significant portion, into L2 cache. Moreover, as can be seen in Figure 6.7 CSR-DUVI improves performance over CSR for all matrices.

Regarding Nehalem, the best CSR-DUVI average speedup for `4c`×`2d` (4.41) and `2t`×`4c`×`2d` (4.57) is achieved by *seq*=8 (Figure 6.8). The respective improvements over CSR-VI are 6.6% and 8.2%, and over CSR's best performing case (`4c`×`2d`) 15.7% and 20%. Finally, there are 5 matrices with reduced CSR-DUVI performance over CSR (Figure 6.9).

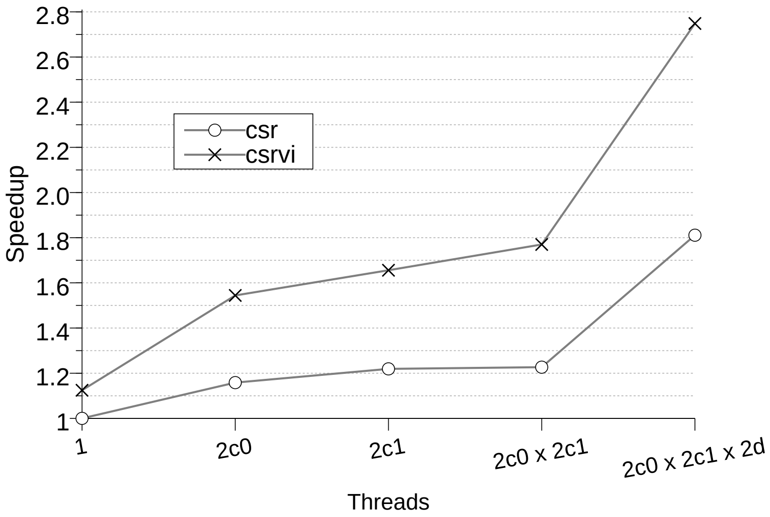


Figure 6.2: CSR and CSR-VI average parallel speedup of SpMxV over serial CSR on Harpertown.

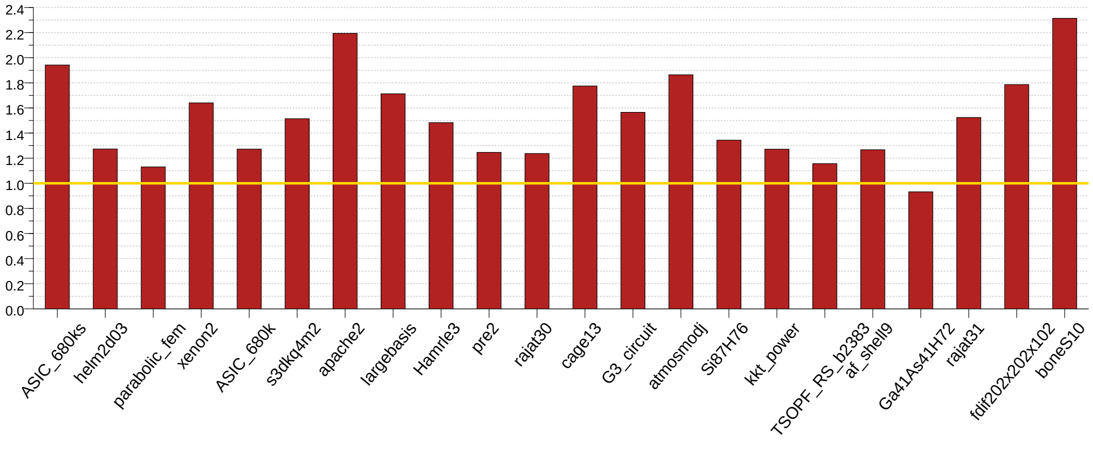


Figure 6.3: SpMxV performance improvement of CSR-VI over CSR for individual matrices, when all 8 Harpertown cores are utilized.

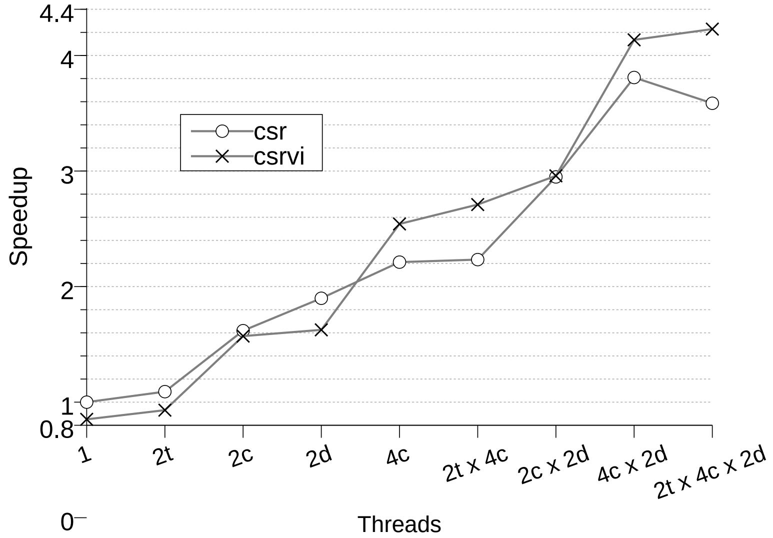


Figure 6.4: CSR and CSR-VI average parallel speedup of SpMxV over serial CSR on Nehalem.

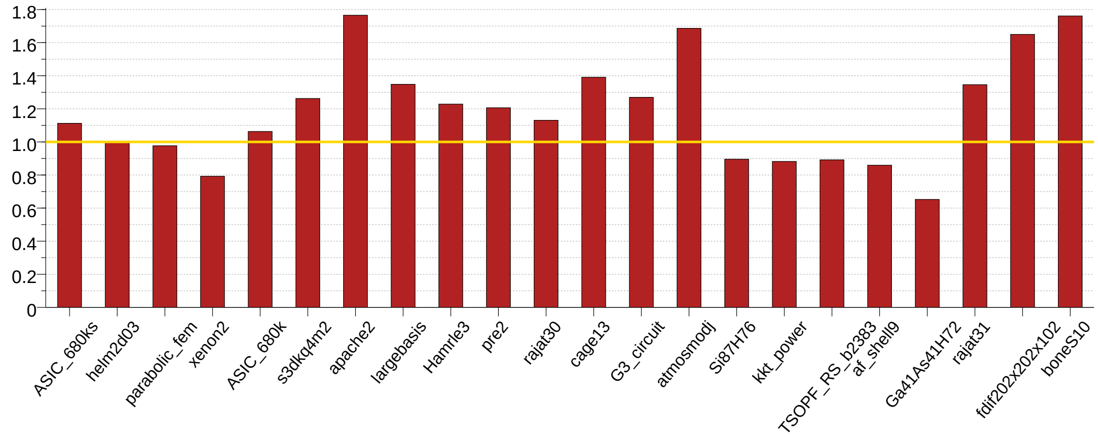


Figure 6.5: SpMxV performance improvement of CSR-VI over CSR for individual matrices, when all 16 Nehalem threads are utilized.

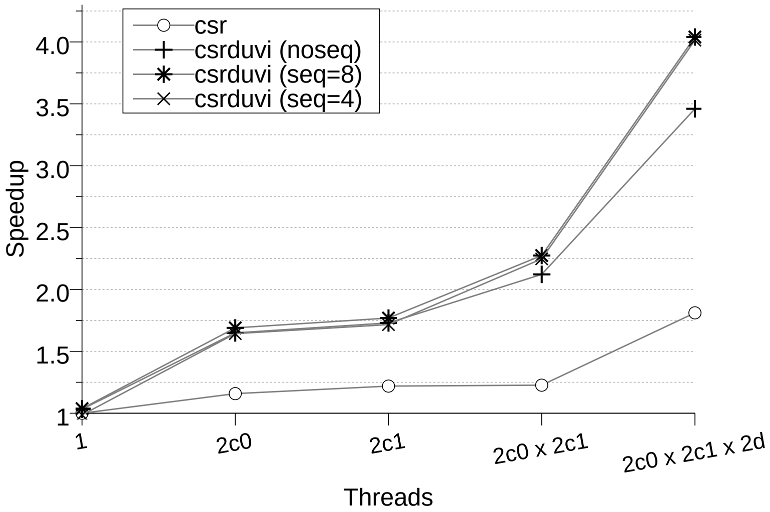


Figure 6.6: CSR and CSR-DUVI average parallel speedup over serial CSR on Harpertown.

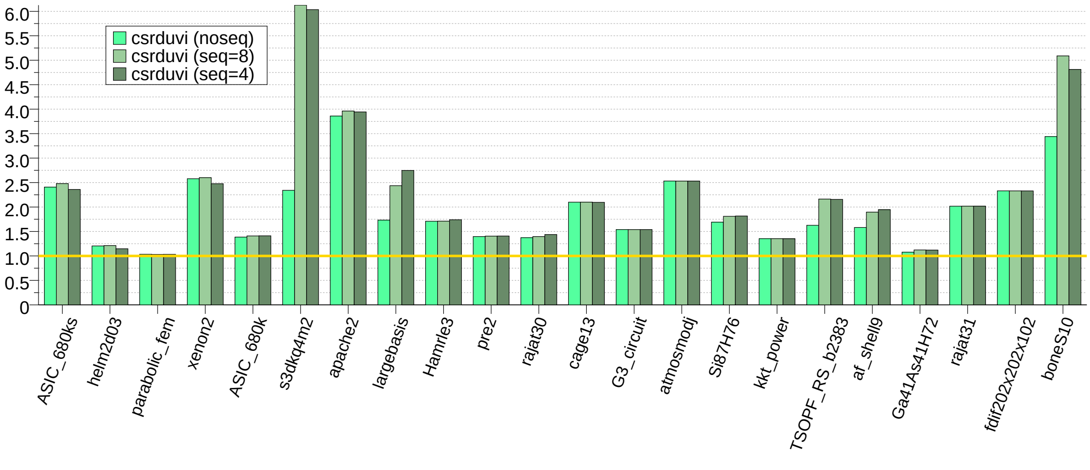


Figure 6.7: Performance improvement of CSR-DUVI over CSR for individual matrices, when all Harpertown 8 cores are utilized.

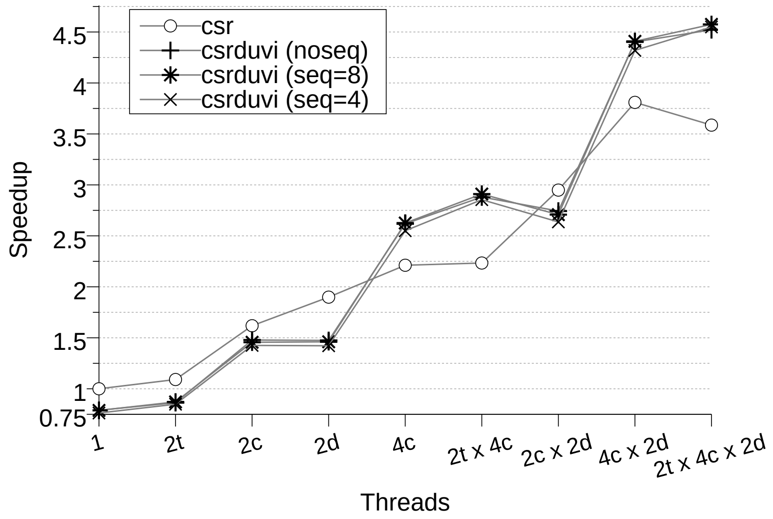


Figure 6.8: CSR and CSR-DUVI average parallel speedup over serial CSR on Nehalem.

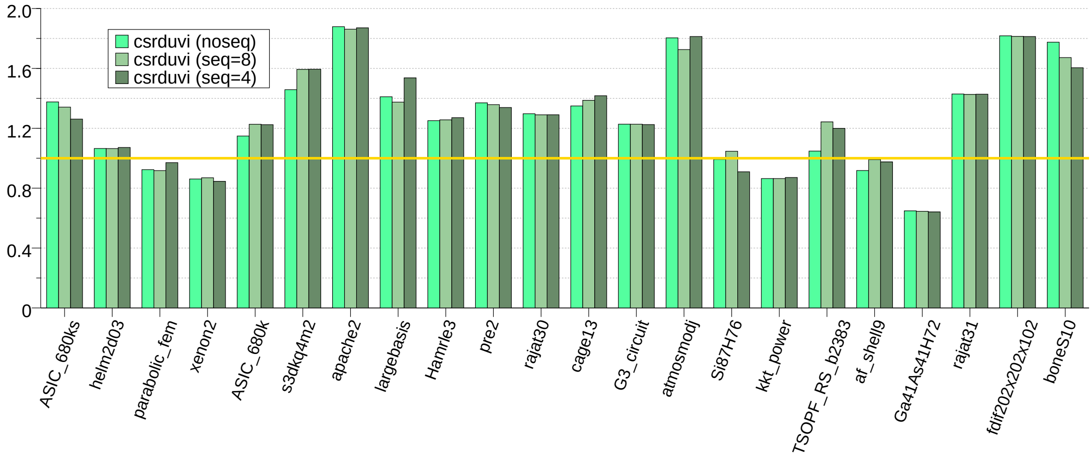


Figure 6.9: Performance improvement of CSR-DUVI over CSR for individual matrices, when all Nehalem 16 threads are utilized.

6.5 Related work

Despite that, in the common case, the value data constitute the larger part of the working set of SpMxV, there has been little research effort targeting its reduction. Lee et al. [LVDY04] exploit matrix symmetry by storing only half the matrix, i.e., reducing matrix data volume by 50%. However, our methods can lead to larger than 50% size reduction. For example, CSR-DUVI applied to the symmetric matrix `boneS10` leads to a reduction of 88.1%. In the context of specialized hardware accelerators for SpMxV, Moloney et al. [MGMM05] discusses compression techniques for both index and value data. Additionally, there exist a number of works in the general area of scientific computation that are related to the value compression for the SpMxV kernel. Keys [Key00], proposes the use of lower precision representation for data that do not pose problems in the convergence procedure, while Langou et al. [LLL$^+$06] propose mixed precision algorithms, which deliver double-precision arithmetic, while performing the bulk of the work in single-precision. Even though these approaches target the exploitation of characteristics of modern architectures (e.g., vectorization), they also contribute significantly to memory bandwidth reduction. In a different context, Burtscher and Ratanaworabhan [BR07] propose a method for the efficient compression of double-precision floating-point values, targeting network data transfers.

7 Enriched Matrix Structure Exploitation

7.1 Motivation and general approach

SpMxV is, in itself, a simple kernel, yet its performance, both actual and maximum, varies substantially for different sparse matrices. Variation on actual performance is exemplified in Figures 4.10 and 4.11, where the achieved FLOPs-per-second rates span a wide range of values. In other words, SpMxV execution time depends strongly on the structure of the sparse matrix, i.e., the input data of the algorithm. Moreover, different sparse matrices have different potential for performance improvement. For example we consider two matrices with the same dimensions and the same number of non-zero elements: (a) a matrix with a full main diagonal and (b) a matrix with one element per row, each element placed on a random column. Obviously, the first matrix's SpMxV execution time would greatly improve if it was stored in a format that exploits diagonal structure (e.g., the DIAG format). On the other hand, it would be difficult to optimize SpMxV performance on the second matrix, if column selection was truly random.

Usually real-world sparse matrices bear at least some structure on their elements as they represent relationships that arise from structured problems. A general storage format, like CSR, does not make any assumptions about the matrix data, and thus it cannot exploit this structure. On the other hand, specialized storage formats aim to improve SpMxV performance by taking advantage of matrix-specific structural properties. Examples of typical structural forms and corresponding storage formats include two-dimensional blocks targeted by the BCSR format and its variants (e.g., VBR), large diagonals targeted by the DIAG format, and contiguous non-zero elements targeted by the format described in [PH99].

As shown by the preceding examples, most storage formats deal with only one type of structural regularity. To efficiently represent matrices with more than one type of regularity, composite formats are used. In composite storage formats a matrix is split in sub-matrices, each stored in a different format [AGZ92]. SpMxV is implemented in two steps: (a) do the multiplication on each sub-matrix and (b) perform a reducing addition at the end.

CSR-DU, similarly with other storage formats, aims at exploitation of specific structural properties — dense areas and contiguous elements. A difference, however, is that CSR-DU is based on the concept of units, i.e. specific structural forms (*substructures*) that are frequently repeated on the matrix. In its current form, CSR-DU is unable to efficiently represent all matrices, the most prominent example being matrices with diagonal structures. To overcome this limitation we extend the concept of units to support several different substructures, providing greater flexibility and eliminating the need for composite formats. We call the resulting storage format compressed sparse extended (CSX) and argue that it can be used for a wide range of optimizations that exploit matrix-specific knowledge.

The goal of CSX is to provide a framework for aggressively optimizing the SpMxV kernel using matrix-specific features. To do this, each matrix is described as a sequence of units. Units are characterized by their type and contain encoded information for generating matrix elements. Each unit type represents a substructure that enables efficient storage of the unit's elements. The SpMxV kernel is implemented as an iteration over these units, where each unit is dispatched into a specialized multiplication routine. Extra care must be taken to ensure that data shared between these routines remain in a consistent state.

Ideally, the matrix, from its creation, would be efficiently described based on the substructures it contains. Currently, however, this not the case since sparse matrices are represented as a series of *(row, column, value)* tuples (e.g., in the Matrix Market exchange formats [BPR96]). Thus, substructures must be identified in a pre-processing phase, which induces potential overhead. We provide an initial evaluation of CSX using some simple substructures described in the following paragraphs. Our evaluation aims at investigating two issues: (a) the ability of the proposed substructures to efficiently describe matrices and (b) the resulting performance benefit on the SpMxV kernel when these substructures are exploited.

7.2 CSX substructures

7.2.1 Horizontal substructures

Using CSR-DU as a starting point, we observe that the encoding of sequential units is a special case of performing run-length encoding on the delta values (e.g., Table 7.1). In CSX we generalize the notion of sequential units to units with elements that have a constant distance. Thus, elements of the form: $(\alpha, \alpha + \delta, \alpha + 2\delta, \ldots)$ are encoded using only their initial value (α), their constant distance (δ), and their quantity. We refer to this encoding as *delta run-length encoding* and to the generated units as *DRLE units*. The SpMxV code for this units type is shown in Listing 7.1. The distance of the unit's elements δ is considered a compile-time constant, while unit size is considered a run-time variable contained in the unit's data.

indices	2	3	5	7	8	9	10	11	12	13	17
deltas	2	1	2	2	1	1	1	1	1	1	4
run-length	(2,1)	(1,1)	(2,2)		(1,6)						(4,1)

Table 7.1: Example of delta run-length encoding.

7.2.2 Vertical and diagonal substructures

To further enhance index compression, we augment DRLE units to include multiple directions for the sparse matrix elements. The directions we consider are vertical, diagonal and anti-diagonal (Table 7.2) using the same rationale as in the horizontal case. Hence, DRLE units are characterized by two parameters: direction and delta value. The multiplication code for vertical, diagonal and anti-diagonal units is shown in Listings 7.2, 7.3 and 7.4, respectively.

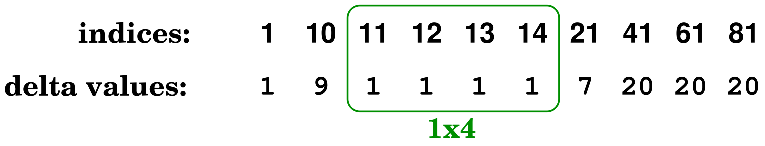
Direction		**Elements**	
		y	x
Horizontal	$\rightarrow$	y_0	$x_0 + i\delta$
Vertical	$\downarrow$	$y_0 + i\delta$	x_0
Diagonal	$\searrow$	$y_0 + i\delta$	$x_0 + i\delta$
Anti-diagonal	$\swarrow$	$y_0 + i\delta$	$x_0 - i\delta$

Table 7.2: Directions for delta run-length encoding. The y and x columns contain an expression for generating the matrix elements for the specific direction given its delta value δ. Note that $0 \leq i < size$, where $size$ the number of elements in the unit.

7.3 CSX matrix construction

7.3.1 Substructure detection

Our substructure detection algorithm handles the supported substructures in a uniform way. The algorithm is based on a delta run-length encoding detector for the horizontal direction, which detects sequences (*runs*) of the same delta value. If the number of elements in a run is larger or equal than a specific configuration parameter, then the items are grouped together in a single unit. The detector can be easily implemented if it is assumed that the elements are iterated in lexicographical order. The detection algorithm pseudocode is shown in Algorithm 7.1.



indices:	1	10	11	12	13	14	21	41	61	81
delta values:	1	9	1	1	1	1	7	20	20	20

Figure 7.1: Horizontal detection example.

To simplify the detection process, detection of overlapped runs is not supported. An example is presented in Figure 7.1, where the detector has been configured to detect runs of size larger or equal than 4. Note that it does not detect the run of the indices 41,61,81, since its size is 3, and it also does not detect the run of the indices 1,41,61,81 since it overlaps with other elements.

In order to detect the rest of the substructures discussed in the previous paragraph, we use the horizontal detector and apply appropriate transformations on the matrix elements coordinates. For example, to detect vertical runs we swap the coordinates of the elements. The transformation

```
xi = x_indx;
yi = y_indx;
for (i=0; i < size; i++){
    y[yi] += *(values++) * x[xi];
    xi += DELTA;
}
```

Listing 7.1: SpMxV code for horizontal DRLE units

```
xi = x_indx;
yi = y_indx;
for (i=0; i < size; i++){
    y[yi] += *(values++) * x[xi];
    yi += DELTA;
}
```

Listing 7.2: Code for vertical DRLE units

```
xi = x_indx;
yi = y_indx;
for (i=0; i < size; i++){
    y[yi] += *(values++) * x[xi];
    xi += DELTA;
    yi += DELTA;
}
```

Listing 7.3: Code for diagonal DRLE units

```
xi = x_indx;
yi = y_indx;
for (i=0; i < size; i++){
    y[yi] += *(values++) * x[xi];
    xi -= DELTA;
    yi += DELTA;
}
```

Listing 7.4: Code for anti-diagonal DRLE units

Algorithm 7.1: Delta run-length encoding algorithm.

```
Input: An $indices$ array containing column indices for a specific row
Input: A $limit$ integer depicting the minimum size for DRLE units
$deltas$ = deltaEncode($indices$)                // perform delta-encoding on indices
$delta_{rle} \leftarrow deltas[0]$                 // delta value of current run
$freq_{rle} \leftarrow 1$                          // frequency of current run
for $i \leftarrow 1$ to $deltas$.size() do
    if $deltas[i] == delta_{rle}$ then                       // the run continues
        $freq_{rle} \leftarrow freq_{rle} + 1$
    else                                                     // the run stops
        if $freq_{rle} \geq limit$ then encode in DRLE units  // run is large enough
        else keep individual indices                          // run is small
        $delta_{rle} \leftarrow deltas[i]$
        $freq_{rle} \leftarrow 1$
```

functions for the substructures implemented are shown in Table 7.3. An example of applying the diagonal transformation function is shown in Figure 7.2. As it is illustrated in the example, the transformation function maps the coordinate pair of a matrix element to a new pair so that: the first coordinate identifies the diagonal of the element and the second coordinate identifies the location of the element in the diagonal. For example, element $(3, 2)$ is mapped to $(5, 2)$, which is the second element of the fifth diagonal.

Substructure	**Transformation**
Horizontal	$(i', j') = (i, j)$
Vertical	$(i', j') = (j, i)$
Diagonal	$(i', j') = (nrows + j - i, \min(i, j))$
Anti-diagonal	$(i', j') = \begin{cases} (nrows + j - i, j), & i < nrows \\ (j, i + j - nrows), & i \geq nrows \end{cases}$

Table 7.3: The transformations used to convert the different substructures to horizontal, in order to feed our detector.

Applying the transformation function, however, is not enough: coordinate pairs must be sorted before entered into the detection algorithm. The most common metric for measuring the computational complexity of a sorting algorithm is the number of comparisons performed and for n elements at least $\mathcal{O}(n \log n)$ comparisons are required in the typical case [Knu73]. Thus, the transformation-based approach is simple and elegant, but in the general case it requires more expensive pre-processing than other methods (e.g., CSR-DU and CSR-VI).

A detailed example of the detection of a diagonal substructure is illustrated in Figure 7.3.

$$\begin{pmatrix} (1,1) & (1,2) & (1,3) & (1,4) \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & (3,2) & (3,3) & (3,4) \\ (4,1) & (4,2) & (4,3) & (4,4) \end{pmatrix} \xrightarrow[j'=\min(i,j)]{i'=nrows+j-i} \begin{pmatrix} (4,1) & (3,1) & (2,1) & (1,1) \\ (5,1) & (4,2) & (3,2) & (2,2) \\ (6,1) & (5,2) & (4,3) & (3,3) \\ (7,1) & (6,2) & (5,3) & (4,4) \end{pmatrix}$$

Figure 7.2: Transformation example for diagonal DRLE units.

Initially, the transformation function T is applied to the elements (❶). Next, the transformed coordinates are lexicographically sorted and passed to the detector (❷). The detector groups the diagonal elements together in a single DRLE unit. For each DRLE unit, the following information is maintained: the first element of the run $(5, 1)$, the current direction $(DIAG)$, the delta value (1), and the unit's size (5). The last step is to apply the inverse transformation function T^{-1} on the first element of the unit (❸). The final result is equivalent to the original data and can be used to generate the corresponding matrix elements.

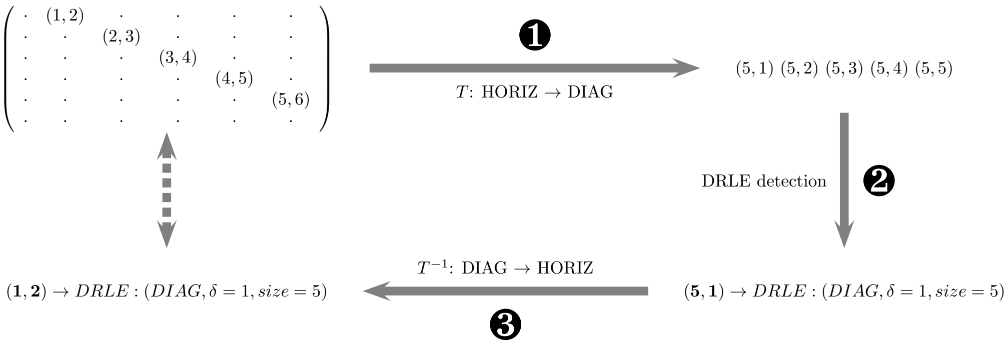


Figure 7.3: Diagonal DRLE detection example.

7.3.2 Substructure selection

Selecting a proper substructure subset for encoding the matrix is critical for SpMxV performance. If we encode matrix data using all possibilities, we run the risk of including a large number of substructures which makes the dispatch logic inefficient. For this reason, during the selection procedure, we filter out substructures that cover less than 10% of the total matrix elements.

We solve the selection problem using a greedy algorithm shown in Algorithm 7.2. At each iteration all possible transformations are applied on the matrix element's coordinates. For each transformation we preform DRLE detection and generate a score value that represents the suitability of the transformation for the matrix data. The transformation with the largest score is selected and appropriate substructure units are constructed. The algorithm is repeated for all remaining transformations until no more elements can be encoded. Remaining elements are placed in delta units, i.e., CSR-DU units, stored in row-major order.

We base the rating (i.e., scoring) of different encodings on two metrics, the first being the number of non-zero elements matched by the substructure. The second, less straightforward, metric is the number of units. If, for example, two encodings match the same number of ele-

Algorithm 7.2: Substructure selection algorithm.

```
Input: An array ($elems$) containing the elements of the matrix
Input: A set ($xforms$) of transformations
while $True$ do
    $score_{max} \leftarrow 0$
    foreach $xf$ in $xforms$ do
        $elems \leftarrow xf(elems)$
        sort($elems$)
        $score \leftarrow$ getScore($elems$)
        if $score > score_{max}$ then
            $score_{max} \leftarrow score$
            $xf_{max} \leftarrow xf$
        $elems \leftarrow xf^{-1}(elems)$
    if $score_{max} == 0$ then
        break
    encode $elems$ using $xf_{max}$
    remove $xf_{max}$ from $xforms$
```

ments, we favor the encoding with the smaller number of units because it results in less overhead, both in storage and computation (e.g., in SpMxV). Specifically, we score encodings using the number of elements encoded minus the total number of units. The scoring value represents a simplistic approach for calculating index size reduction by considering that the storage size of each substructure is equal to the storage size of individual elements:

$$score = total_{nnz} - (units + total_{nnz} - encoded_{nnz}) = encoded_{nnz} - units$$

A selection example is illustrated in Figure 7.4. The matrix in this example contains two substructures: a diagonal and a vertical. Since the vertical substructure contains more elements than the diagonal it is selected on the first iteration and its elements are replaced with a unit positioned at $(1, 2)$. On the second iteration a diagonal unit is created starting at $(2, 3)$. Note that an element, e.g., $(1, 2)$, cannot be a member of two or more units.

7.3.3 Matrix encoding

Similarly to the CSR-DU format, we encode the matrix index data in a single byte-array called `ctl`. Each unit starts with two bytes: `usize` and `uflags`. `usize` contains the number of unit elements, and `uflags` the unit's type along with some bookkeeping information. From the 8 bits of `uflags`, 6 are reserved for the encoding of the type, and 2 are used for a new row marker and a row offset marker. If the row offset bit is set, the header is followed by a variable-length integer equal to the number of empty rows. This is necessary, because the use of CSX units in directions other than the horizontal may lead to empty rows. Finally, a unit offset field (`ujmp`) is appended to the header. A unit, based on its type, may or may not include a main body: delta units contain the

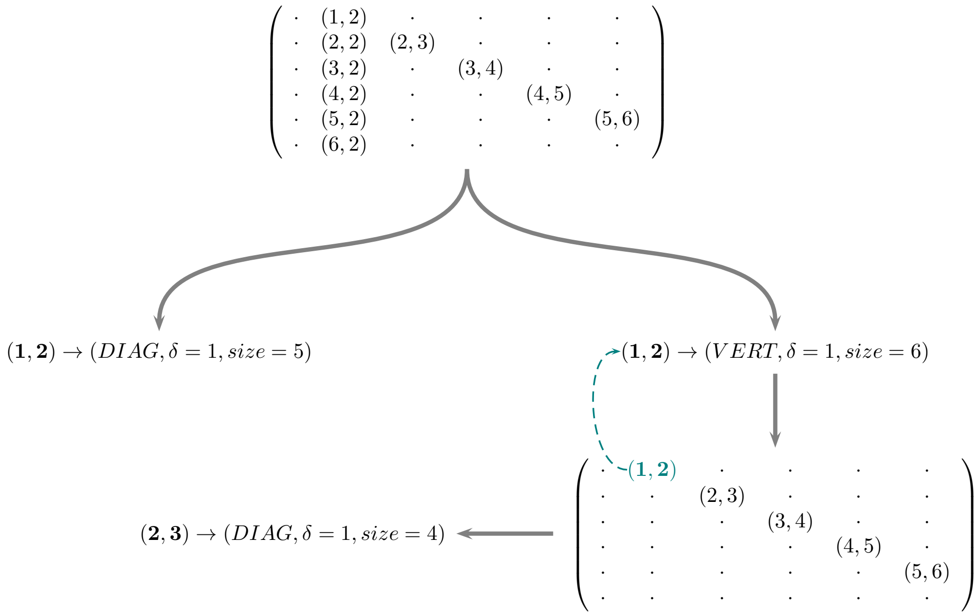


Figure 7.4: Substructure selection example

delta-encoded column indices, while DRLE units require only the header for element generation.

7.3.4 Multiplication routines

Conceptually the SpMxV operation on a CSX matrix is a two-level procedure. At the first level all units are iterated, while at the second level the appropriate multiplications and additions are performed. Note that for supporting a different operation, only the second level needs to change. In our implementation the units are iterated in row-major order based on their starting elements. We use a `switch` statement to transfer control from the first to the second level, based on each unit's type.

To accommodate for all possible cases, we employ a run-time code generation technique [KEH91], where a specialized SpMxV routine is generated for each matrix based on the substructure types its encoding contains. We base our implementation on the LLVM [LLV, LA04] compiler infrastructure. A core component of LLVM is its intermediate representation (IR), which resembles a RISC-like assembly, and it can be manipulated by optimization passes and used to produce native code for a number of different ISAs. The code for the SpMxV operation is generated programmatically in LLVM's IR. Subsequently, it is optimized and dynamically compiled to native code. A persistent cache of generated versions can be used to reduce the overhead of compilation and optimization.

7.4 Restrictions and extensions

Overall, CSX is a very flexible storage format. In our implementation, we detect substructures using delta run-length encoding and two-dimensional transformations. This unified and simple detection algorithm allows support for different substructures, provided that they can be expressed as transformations. For example, we can extend our system to support two-dimensional block structures by defining proper coordinate mappings. On the other hand, the detection algorithm induces a significant run-time overhead in pre-processing, making our approach impractical for a number of real-world scenarios.

We argue, however, that it is possible to reduce the pre-processing overhead by altering the detection algorithm. A first step towards this end is to perform detection on a limited window of the matrix elements. If the size of the window is constant, sorting is applied to a constant number of elements and does not affect overall complexity. The problem with this approach is that, by limiting our view to a local scope of constant size, it is difficult for the detector to make sound decisions on a global scale. For example, when acting locally the detector is unable to affirm whether a particular substructure will encode a large portion of the total matrix elements.

7.5 Experimental evaluation

7.5.1 Experimental setup

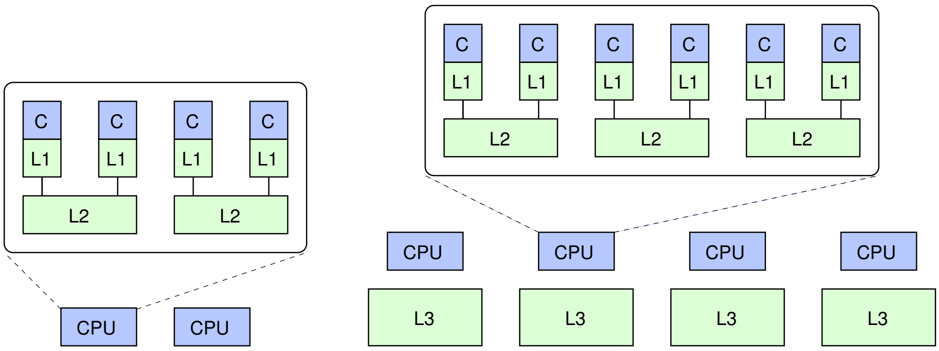
We perform our evaluation on two systems: (a) a two-way quad-core system* (8 cores total) based on Intel Harpertown processors (Figure 7.5a) and (b) a four-way 6-core system (24 cores total) based on Intel Dunnington processors (Figure 7.5b). Table 7.4 provides a more detailed description of the main characteristics of these systems.

Both systems run a 64-bit version of the Linux OS (kernel version 2.6). We used version 2.5 of the LLVM compiler infrastructure and llvm-gcc 4.2.1 (a modified version of gcc that acts as a front-end for LLVM) as a static compiler. Threads are always scheduled to run on cores that are as "close" as possible. For example, in the Harpertown processor, 2 threads are scheduled on cores which share the L2 cache, while 4 threads are scheduled on the same physical package. For CSX, we perform a separate substructure selection for each thread's data and group together units with a size larger or equal than 4. Our setup is otherwise similar to that of previous chapters: we use 32-bit indices and 64-bits values, measure the performance of 128 consecutive SpMxV operations, use the y output vector as the next iteration's x vector, and evaluate our methods on a suite of 50 matrices (Table 4.3).

7.5.2 CSX encoding

Initially, we discuss the effectiveness of CSX in capturing substructures for the matrices of our set. Figure 7.6 presents a breakdown of the resulting unit types from the substructure selection phase for 1 thread. Delta units are designated with *Dx*, where x is the number of bits used for the delta values, while DRLE encoded elements are designated with *DIR* (δ), where *DIR* is the

*The same system used in the experimental evaluation of previous chapters (4, 5, 6).



(a) Harpertown system

(b) Dunnington system

Figure 7.5: Cache hierarchy for the systems used for CSX performance evaluation

	Harpertown	Dunnington
Frequency (Ghz)	2.0	2.66
L1 (data/instruction)	32k/32k	32k/32k
L2 (unified)	6M (1/2 cores)	3M (1/2 cores)
L3 (unified)	-	16M (1/chip)
Number of cores	$2 \times 4 = 8$ cores	$4 \times 6 = 24$ cores

Table 7.4: Overview of the systems used for CSX performance evaluation

direction and δ is the corresponding delta value. For instance, *D16* is used for delta encoded elements using 2-bytes delta values and *Horizontal (2)* is used for horizontal elements with a delta value of 2†.

A significant number of elements in the matrices of our set adhere to DRLE substructures. The majority of the elements are encoded in horizontal, vertical or diagonal directions with $\delta = 1$. There are some cases of matrix elements encoded in a anti-diagonal direction (e.g., `Ga41As41H72`) or DRLE substructures with $\delta \neq 1$ (e.g., `Chebyshev4`), but they are limited. Hence, for the majority of the matrices in our set, we could reduce the overhead of pre-processing by limiting the detectable substructures. For example, detection restrained in a single direction and a single delta value (e.g., Diagonal with $\delta = 1$) can be implemented in $\mathcal{O}(nnz)$ steps, by keeping appropriate information in buffers.

7.5.3 CSX SpMxV performance

Next, we discuss SpMxV performance for the CSX format. We consider three CSX variations: (a) CSX without DRLE substructures (*delta*) (b) CSX with only horizontal DRLE substructures (*horiz*) and (c) CSX with all possible DRLE substructures (*full*) . Note that the second and the first variations are roughly equivalent to CSR-DU with and without sequential units, respectively. The average speedups of the aforementioned CSX methods against serial CSR are illustrated in

†$(x_0 + 2i, y_0), i = 0, 1, \ldots$

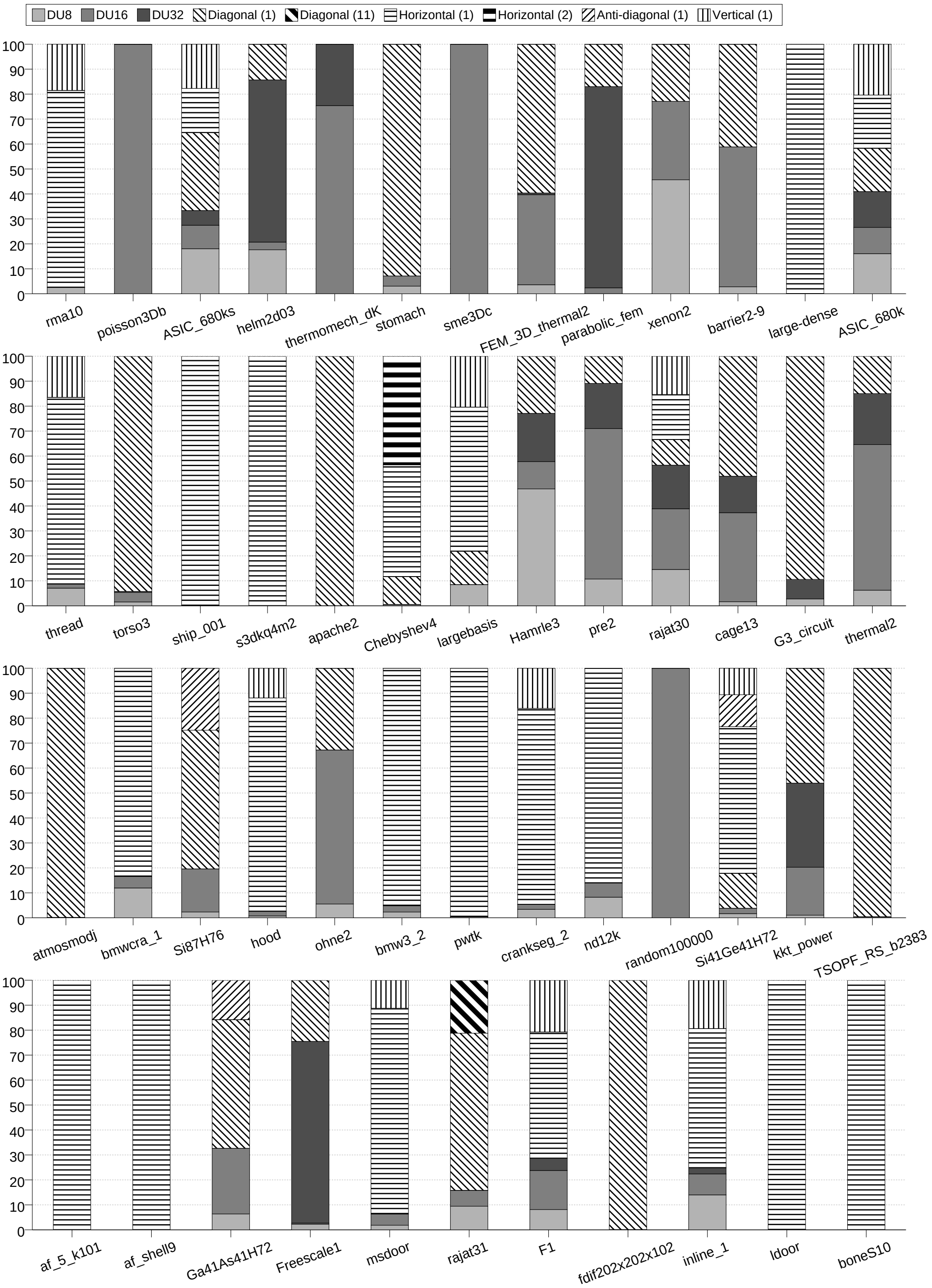
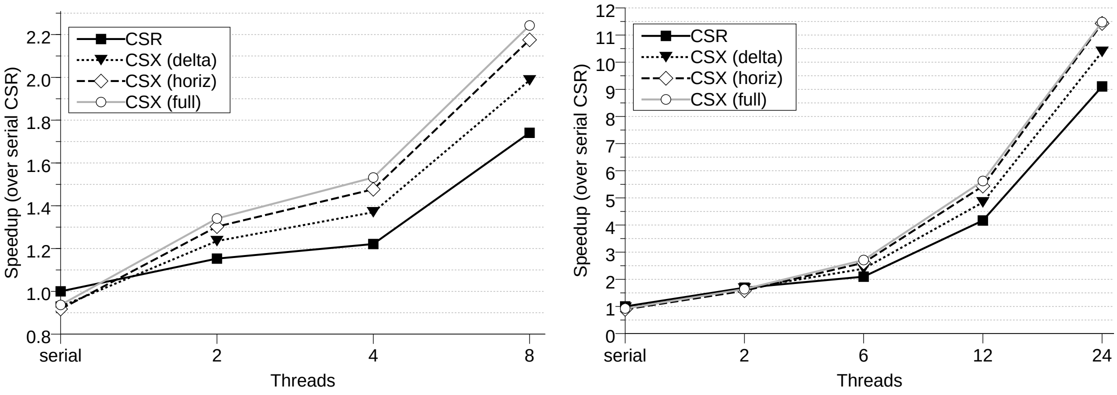


Figure 7.6: Breakdown of types resulting from CSX selection for each matrix

Figures 7.7a (Harpertown) and 7.7b (Dunnington). To avoid possible confusion, we note that the mismatching of Harpertown CSR results with the results presented in Chapter 4 is an outcome using different compilers.

The two systems exhibit similar behavior. On average, the *delta* and *horiz* CSX versions lead to significant speedup increase. When all 8 cores are utilized in Harpertown, the average speedups of *delta* and *horiz* versions are 1.99 and 2.17, improving upon CSR by 14% and 25%, respectively. The corresponding Dunnington speedups for 24 threads are 10.38 and 11.43 — improving upon CSR by roughly the same percentages as in the Harpertown case. The *full* version of CSX, however, provides little improvement over the *horiz* version. This is because the majority of the matrices in our experimental set are dominated by horizontal substructures. For these matrices, further compression leads to diminishing returns or even performance degradation.



(a) Harpertown

(b) Dunnington

Figure 7.7: Average CSR and CSX parallel speedup over serial CSR.

On the other hand, *full* CSX is able to significantly improve the performance for matrices that are not dominated by horizontal structures. Detailed, per-matrix results are illustrated in Figures 7.8 and 7.9 that show the performance improvement of CSX methods over CSR, when all available cores are utilized. These graphs show that several matrices exist, for which the full version of CSX offers significant performance advantages over other CSX variants. Examples of such matrices are `stomach`, `torso3`, `apache2`, `G3_circuit`, `atmosmodj` and `Si87H76`, which are earpdominated by diagonal elements.

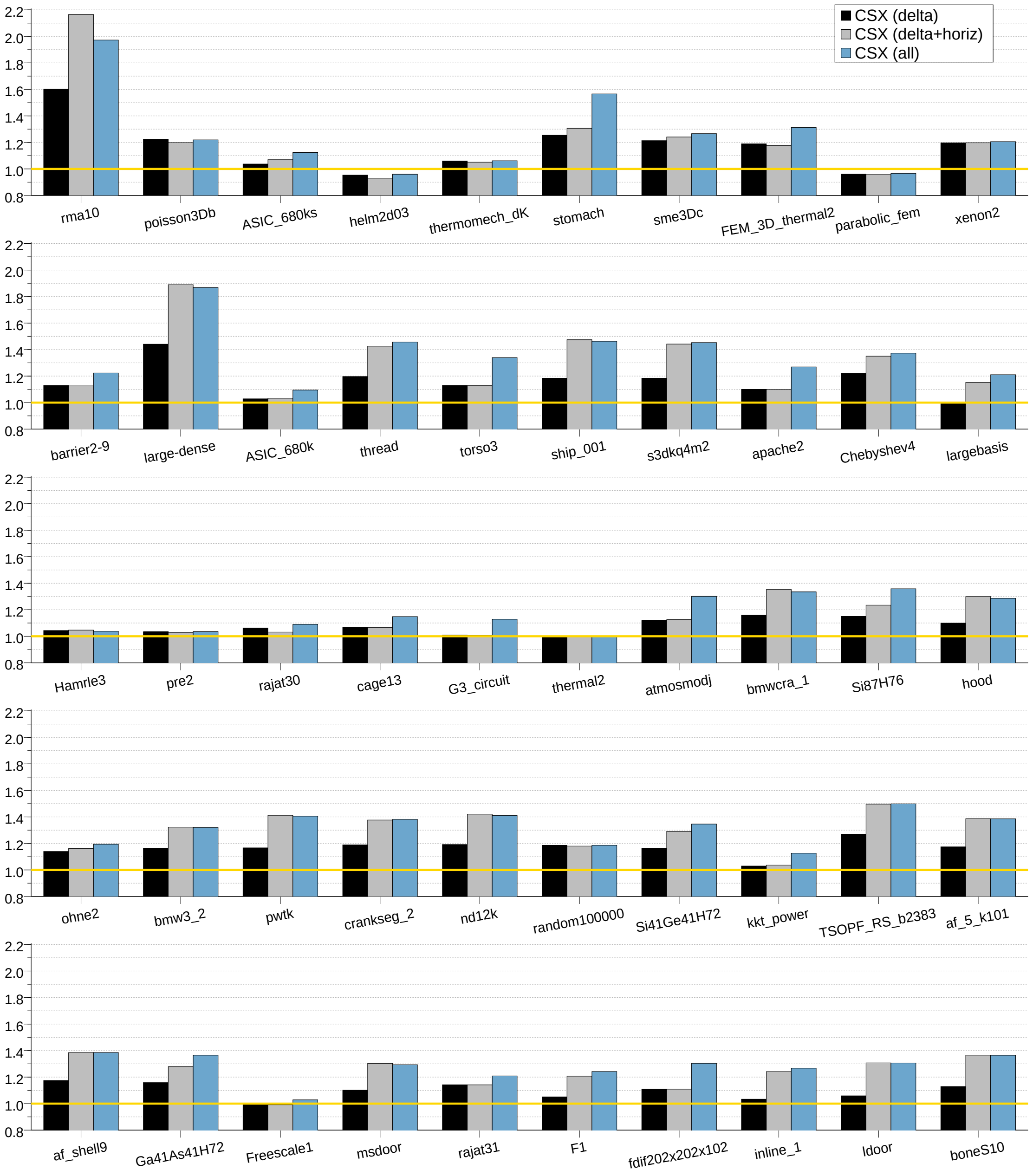


Figure 7.8: SpMxV performance improvement of CSX over CSR for individual matrices when all 8 Harpertown cores are utilized.

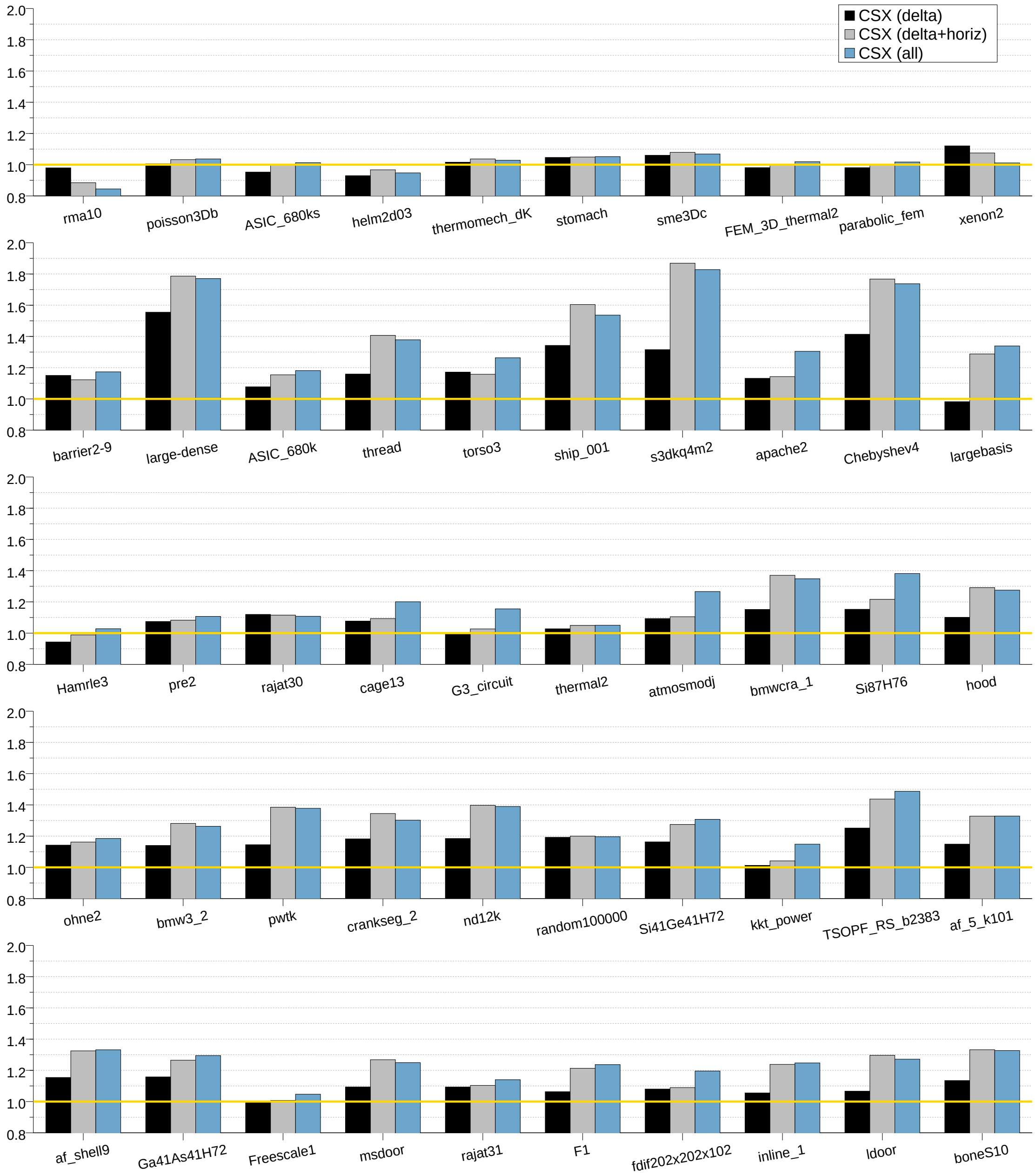


Figure 7.9: SpMxV performance improvement of CSX over CSR for individual matrices when all 24 Dunnington cores are utilized.

8 Conclusions and future work

One of the major performance bottlenecks of multicore architectures is the main memory subsystem because it is shared among cores. For the majority of applications this problem is solved via a cache hierarchy which reduces accesses to the main memory. In this thesis, however, we are concerned with applications that: (a) are unable to benefit from caching due to limited temporal locality and (b) they are characterized by a low computation to memory access ratio. These applications will typically perform poorly in a multithreaded environment, even if their parallelization introduces minimal overhead.

Our work proposes the use of compression techniques to tackle the aforementioned problem by sacrificing (scalable) CPU cycles to alleviate memory pressure. We direct our efforts towards SpMxV, an important scientific kernel used in a variety of applications. We devise two sparse matrix storage formats: CSR-DU and CSR-VI, which apply compression to the index and value data of the matrix, respectively. More specifically, CSR-DU applies a coarse-grained delta encoding compression scheme for column indices, and optionally supports dense variable-length one-dimensional blocks. CSR-VI, on the other hand, uses indirect indexing for the numerical value data, and can be meaningfully applied to matrices that exhibit a large percentage of common values. Moreover, we also considered the combination of these two formats (CSR-DUVI), that employs both the aforementioned techniques. Our experimental evaluation showed that all methods demonstrate a noticeable performance improvement when all available cores are employed. Additionally, our proposed methods exhibited performance stability, since only a small subset of our suite resulted in a significant slowdown compared to base-line performance (CSR).

Based on our previous work, we identify the need for a sparse matrix format that can be adapted to different structural properties. Towards this direction, we present a generalization of the CSR-DU format called CSX. CSX is able to utilize one-dimensional substructures across the same row, column, diagonal or anti-diagonal using a delta run-length encoding scheme. CSX is a very flexible storage format and can be further extended to incorporate other families of substructures if necessary. Since the majority of the matrices in our experimental set adhere to horizontal patterns already supported by CSR-DU, further compression applied by CSX leads to diminishing returns, i.e., small performance improvements. CSX, however, was able to significantly improve the performance for a number of matrices with diagonal substructures.

Next, we list possible future work directions.

- *Framework for adaptable SpMxV*: The optimal performance of the SpMxV kernel depends on two factors: the nature of the sparse matrix and the execution architecture. Our work focuses on general matrices consisting of double-precision floating-point values, and shared memory architectures that are unable to deliver the necessary memory bandwidth when all cores are utilized. Although one could argue that these conditions are the most common, they are not universal. In different conditions, the effect of the various optimizations can change. For instance, index compression is expected to be less beneficial when the matrix values are complex, while the converse is true for matrices with integer values.

  Hence, an ideal SpMxV implementation should be able to adapt to different conditions (e.g., matrix symmetry, data type of matrix values, number of threads used, characteristics of the underlying micro-architecture) by being able to transform both the data and the code. We argue that the CSX storage format is a good starting point for such an attempt, due to its generality and flexibility.

- *Support for other sparse operations*: The SpMxV operation is a very important kernel for sparse computations, yet it is not the only one. We believe that our work, and specifically the CSX format, can be used to improve performance of other operations as well. CSX substructures allow for a semantically richer representation of the matrix, which is an essential requirement in realizing and exploiting optimization opportunities. A related, but more difficult, problem is the creation of proper representations for operations and substructures, such that their automatic composition is possible.

- *Use of compression techniques in other application domains*: As multicore processors become the norm and core counts increase, more applications will experience reduced performance due to limited memory bandwidth. Although the use of compression is not applicable to all cases, we believe that it can be used in applications domains other than sparse computations. Graph and database domains consist good candidates for such an approach — especially in read-only or read-mostly environments.

Bibliography

[AAKK06] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Exploring the performance limits of simultaneous multithreading for scientific codes. In *ICPP*, pages 45–54. IEEE Computer Society, 2006. 14

[AAKK08] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *The Journal of Supercomputing*, 44(1):64–97, 2008. 14

[ABC$^+$06] 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, EECS Department, University of California, Berkeley, December 18 2006. 20

[ABD$^+$09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009. 5

[AGZ92] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing'92*, pages 32–41, Minn., MN, November 1992. IEEE. 6, 27, 46, 79

[ATKS07] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007. 6

[BBC$^+$94] 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, Philadelphia, 1994. 22, 27, 46

[BBR09] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 100–109, New York, NY, USA, 2009. ACM. 6, 47, 66

[BDH$^+$08] Kevin Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. A Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing. *Parallel Processing Letters*, December 2008. 37

[BELF07] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *IJHPCA*, 21:467–484, 2007. 6, 25, 35, 36, 47

[BFF^{+}09] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, New York, NY, USA, 2009. ACM. 33, 48

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998. 20

[BPR96] R. F. Boisvert, R. Pozo, and K. A. Remington. The matrix market exchange formats: Initial design. Technical Report NISTIR 5935, Applied and Computational Mathematics Division, National Institute of Standards and Technology, Gaithersburg, December 1996. 80

[BR07] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *DCC '07: Proceedings of the 2007 Data Compression Conference*, pages 293–302, Washington, DC, USA, 2007. IEEE Computer Society. 77

[CA96] U. V. Catalyuerek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. *Lecture Notes In Computer Science*, 1117:75–86, 1996. 6, 47, 48

[CLRS01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001. 29

[CS99] D.E. Culler and J.P. Singh. *Parallel computer architecture*. Morgan Kaufmann Publishers San Francisco, 1999. 9

[CSC^{+}05] Adrian Cristal, Oliverio J. Santana, Francisco Cazorla, Marco Galluzzi, Tanausu Ramirez, Miquel Pericas, and Mateo Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, 2005. 5

[Dav97] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997. 19, 20, 42

[Deu96] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996. 17

[DG96] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996. 17

[Gee05] D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, 2005. 5

[GHF$^+$06] M. Gschwind, H. P. Hofstee, B. K. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. 33

[GKA$^+$08] 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*, 2008. 36

[GR99] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In *Parallel Computing: Fundamentals and Applications, International Conference ParCo*, pages 308–315. Imperial College Press, 1999. 6, 35, 47, 48

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, 42(10):695–706, 2007. 6

[HK99] Bruce Hendrickson and Tamara G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.*, 21(6):2048–2072, 1999. 19

[HP07] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007. 9

[Im00] E. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000. 6, 35, 46

[IY99] E. Im and K. Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *9th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999. 6, 46, 47

[IY01] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. *Lecture Notes in Computer Science*, 2073:127–136, 2001. 6, 23, 36, 46

[Joh99] Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB*, pages 278–289, 1999. 17

[KCA09] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. Speeding up algorithms on compressed web graphs. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 272–281, New York, NY, USA, 2009. ACM. 19, 20

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991. 86

[Key00] D.E. Keyes. *Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations*. Springer, 2000. 77

[KGK08] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM. 1

[KGK09a] 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. 6, 25

[KGK09b] 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 and Distributed Processing (Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications)*, Rome, Italy, 2009. IEEE. 6, 24, 25

[KHK$^+$05] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance evaluation of parallel sparse matrix-vector products on SGI Altix3700. In *1st International Workshop on OpenMP (IWOMP)*, Eugene, OR, USA, June 2005. 6, 48

[KKR$^+$99] J.M. Kleinberg, S.R. Kumar, P. Raghavan, S. Rajagopalan, and A.S. Tomkins. The web as a graph: Measurements, models and methods. *Lecture Notes in Computer Science*, pages 1–17, 1999. 19

[Knu73] D.E. Knuth. *The art of computer programming. Vol. 3, Sorting and Searching.* Addison-Wesley Reading, MA, 1973. 83

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 86

[LLL$^+$06] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113, New York, NY, USA, 2006. ACM. 77

[LLV] The LLVM Compiler Infrastructure. `http://www.llvm.org/`. 86, 105

[LVDY04] B.C. Lee, R.W. Vuduc, J.W. Demmel, and K.A. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP '04: Proceedings of the International Conference on Parallel Processing*, pages 169–176 vol.1, 15-18 Aug. 2004. 77

[MCG04] 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, 2004. 6, 35, 47

[MGM$^+$09] Paul E. McKenney, Manish Gupta, Maged M. Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole. Is parallel programming hard, and if so, why? Technical report, Portland State University, Portland, OR, USA, February 2009, 2009. 6

[MGMM05] David Moloney, Dermot Geraghty, Colm McSweeney, and Ciarán McElroy. Streaming sparse matrix compression/decompression. In Thomas M. Conte, Nacho Navarro, Wen mei W. Hwu, Mateo Valero, and Theo Ungerer, editors, *HiPEAC*, volume 3793 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 2005. 77

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. 5

[ONH$^+$96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM. 5

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49, New York, NY, USA, 1997. ACM. 16

[PDG06] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multicore: preparing for a new exponential. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, New York, NY, USA, 2006. ACM. 5

[PH99] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing'99*, Portland, OR, November 1999. ACM SIGARCH and IEEE. 6, 35, 36, 46, 49, 54, 66, 79

[PHCR04] 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 *PDP*, pages 66–71. IEEE Computer Society, 2004. 6, 35, 47, 48, 66

[PHCR05] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing*, 31(8-9):858–876, 2005. 6, 47

[PRdB89] G.V. Paolini and G. Radicati di Brozolo. Data structures to vectorize CG algorithms for general sparsity patterns. *BIT Numerical Mathematics*, 29(4):703–718, 1989. 46

[Saa94] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2. 42

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2003. 19, 20, 22, 27, 46

[SBB$^+$07] Manish Shah, Jama Barreh, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, Bikram Saha, Denis Sheahan, Lawrence Spracklen, and Aaron Wynn. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC. In *Asian Solid-State Circuirts Conference*, 2007. 14

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. 6

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. 5

[SW05] Malik Silva and Richard Wait. Sparse matrix storage revisited. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 230–235, New York, NY, USA, 2005. ACM. 46

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM. 14, 37

[TJ92] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing'92*, pages 578–587, Minnesota., MN, November 1992. IEEE. 6, 46, 66

[Tol97] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997. 6, 46

[UGMW01] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. 16

[VB05] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005. 19, 33

[VDY$^+$02] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing*, Baltimore, MD, November 2002. 6, 47

[VM05] R. W. Vuduc and H. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816. Springer, 2005. 6, 24, 36, 47

[Vud03] Richard Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003. 6, 22, 23, 25, 26, 27

[Wil08] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. 6, 27

[WL06] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS '06: Proceedings of the 20th annual International Conference on Supercomputing*, pages 307–316, New York, NY, USA, 2006. ACM Press. 6, 36, 47, 49, 66

[WOS06] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006. 17

[WOV$^+$07] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, November 2007. 33, 48

[WOV$^+$09] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications. 6, 48

[WS97] J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *HiPC '97: 4th International Conference on High Performance Computing*, 1997. 6, 36, 46

A Implementation Details

A.1 Blocked version of matrix multiplication

Listing A.1 shows an implementation of blocking matrix multiplication for N×N matrices.

```
for (bi = 0; bi < N; bi += mstep)
    i_max = MIN(N, bi+mstep);
    for (bk = 0; bk < N; bk += rstep)
        k_max = MIN(N, bk+rstep);
        for (bj = 0; bj < N; bj += cstep)
            j_max = MIN(N, bj+cstep);
            for (i = bi ; i < i_max; i++)
                for (k = bk; k < k_max; k++)
                    for ( j = bj ; j < j_max; j++)
                        C[i][j] += A[i][k] * B[k][j];
```

Listing A.1: Matrix multiplication kernel (blocked version)

A.2 Memory throughput benchmark

To measure the memory throughput of a system, we developed a benchmark that allocates and initializes large memory areas and subsequently performs read operations using streaming instructions. The benchmark supports multiple threads and NUMA-aware allocation. The following function is the core function of our memory benchmark. It uses x86 Streaming SIMD Extensions (SSE) instructions to load data from memory to registers. At each iteration, it performs 16 independent 16-byte loads.

```
void bwm_read(void *data)
{
        asm volatile (
                ”movdqa 0(%[d]),   %%xmm0 \n\t”
                ”movdqa 16(%[d]),  %%xmm1 \n\t”
                ”movdqa 32(%[d]),  %%xmm2 \n\t”
                ”movdqa 48(%[d]),  %%xmm3 \n\t”
                ”movdqa 64(%[d]),  %%xmm4 \n\t”
                ”movdqa 80(%[d]),  %%xmm5 \n\t”
                ”movdqa 96(%[d]),  %%xmm6 \n\t”
                ”movdqa 112(%[d]), %%xmm7 \n\t”
                ”movdqa 128(%[d]), %%xmm8 \n\t”
                ”movdqa 144(%[d]), %%xmm9 \n\t”
                ”movdqa 160(%[d]), %%xmm10 \n\t”
                ”movdqa 176(%[d]), %%xmm11 \n\t”
                ”movdqa 192(%[d]), %%xmm12 \n\t”
                ”movdqa 208(%[d]), %%xmm13 \n\t”
                ”movdqa 224(%[d]), %%xmm14 \n\t”
                ”movdqa 240(%[d]), %%xmm15 \n\t”
                :
                : [d] ”r”(data)
        );
}
```

Listing A.2: Memory throughput benchmark: streaming reads on x86 (64 bit)

A.3 Memcomp benchmark

The memcomp benchmark creates synthetic instruction streams and measures their performance. The streams are created based on three parameters: *c*, *unroll* and *loops*. Each stream is essentially a loop that is executed *loops* times. At each iteration of this loop there are *unroll* instances of: a memory load followed by *c* additions. The implementation is based on the LLVM [LLV] compiler infrastructure. We consider an example where $c = 3$ and $unroll = 64$; Listings A.3 and A.4 show the resulting code for LLVM and x86 ISA, respectively.

```
%2 = load double* %val_ptr
%next_cnt = add i64 %cnt, 1
%val_add = add double %2, %1
%val_add1 = add double %2, %val_add
%val_add2 = add double %2, %val_add1
%val_ptr3 = getelementptr double* %0, i64 %next_cnt
%3 = load double* %val_ptr3
%next_cnt4 = add i64 %next_cnt, 1
%val_add5 = add double %3, %val_add2
%val_add6 = add double %3, %val_add5
%val_add7 = add double %3, %val_add6
%val_ptr8 = getelementptr double* %0, i64 %next_cnt4
(...)
%65 = load double* %val_ptr313
%next_cnt314 = add i64 %next_cnt309, 1
%val_add315 = add double %65, %val_add312
%val_add316 = add double %65, %val_add315
%val_add317 = add double %65, %val_add316
```

Listing A.3: Memcomp benchmark example for $c=3$ and $unroll=64$ (LLVM)

```
movsd   (%rdi,%rax,8), %xmm1
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
movsd   8(%rdi,%rax,8), %xmm1
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
(...)
movsd   504(%rdi,%rax,8), %xmm1
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
addsd   %xmm1, %xmm0
```

Listing A.4: Memcomp benchmark example for $c=3$ and $unroll=64$ (x86 assembly)