Autotuning Sparse Matrix-Vector Multiplication for Multicore

Jong-Ho Byun
Richard Lin
Katherine A. Yelick
James Demmel

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-215
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.html

November 28, 2012

Acknowledgement

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C.Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia,
NVIDIA, Oracle, and Samsung. Also supported by U.S. DOE grants DE-SC0003959, DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-AC02-05-CH11231, DE-FC02-06ER25753, DE-FC02-07ER25799, and DE-FC03-01ER25509.

Autotuning Sparse Matrix-Vector Multiplication for Multicore

Jong-Ho Byun[1], Richard Lin[1], Katherine Yelick[1,2], and James Demmel[1,2]

[1]EECS Department, University of California at Berkeley, Berkeley, CA, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract

Sparse matrix-vector multiplication (SpMV) is an important kernel in scientific and engineering computing. Straightforward parallel implementations of SpMV often perform poorly, and with the increasing variety of architectural features in multicore processors, it is getting more difficult to determine the sparse matrix data structure and corresponding SpMV implementation that optimize performance. In this paper we present pOSKI, an autotuning system for SpMV that automatically searches over a large set of possible data structures and implementations to optimize SpMV performance on multicore platforms. pOSKI explores a design space that depends on both the nonzero pattern of the sparse matrix, typically not known until run-time, and the architecture, which is explored off-line as much as possible, in order to reduce tuning time. We demonstrate significant performance improvements compared to previous serial and parallel implementations, and compare performance to upper bounds based on architectural models.

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Sparse matrix-vector multiplication, Auto-tuning, Multicore.

1 Introduction

Sparse matrix-vector multiplication (SpMV) is an important kernel for a diverse set of applications in many fields, such as scientific computing, engineering, economic modeling, and information retrieval. Conventional implementations of SpMV have historically performed poorly, running at 10% or less of system peak performance on many uniprocessors, for two major reasons: (1) indirect and irregular memory accesses generally result in little spatial or temporal locality, and (2) the speed of accessing index information in the data structure is limited by memory bandwidth [17, 32, 35, 15]. Since multicore architectures are widely used (starting with dual-core processors in 2001, and now throughout supercomputer, desktop and embedded computing systems), we want to make SpMV as efficient as possible by exploiting multicore's architectural features such as the number of cores, simultaneous multithreading, SIMD intrinsics, non-traditional memory hierarchy including NUMA and shared/private hardware resources [33, 35, 19].

Since SpMV performance depends strongly both on the matrix sparsity pattern and the micro-architecture, optimizing SpMV requires choosing the right combination of data structure and corresponding implementation that best exploit the architecture. This is difficult for two reasons: First, the large variety of sparsity patterns and architectures, and their complicated interdependencies, make the design space quite large. Second, since the sparsity pattern is typically not known until run-time, we have to explore this large design space very quickly. This is in contrast to situations like dense matrix multiplication [4, 33] where off-line tuning is sufficient, so significant time can be spent optimizing. Since the increasing complexity of architectures also makes exploring the design space by hand more difficult, we are motivated to develop autotuning systems that automatically and quickly provide users with optimized SpMV implementations.

The HPC community has been developing autotuning methodologies with empirical search over design spaces of implementations for a variety of important scientific computational kernels, such as linear algebra

and signal processing. For examples, ATLAS [33] is a auto-tuning system that implements highly optimized dense BLAS (basic linear algebra subroutines) and some LAPACK [22] kernels. FFTW [13] and SPIRAL [26] are similar systems for signal processing kernels.

Our work is most closely based on OSKI [32] (Optimized Sparse Kernel Interface), which applies autotuning to several sparse linear algebra kernels, including SpMV and sparse triangular solve. OSKI automatically searches over the several sparse storage formats and optimizations. The sparse storage formats (see Section 2.1 for more details) include CSR, CSC, BCSR and VBR, and the optimizations include register blocking and loop unrolling. Since the nonzero pattern of the sparse matrix is typically not known until run-time, OSKI combines both off-line and run-time tuning. To reduce the run-time tuning costs, OSKI uses a heuristic performance model to select the best data structure instead of exhaustive search. However, OSKI only supports autotuning for cache-based superscalar uniprocessors, while ATLAS, FFTW and SPIRAL support autotuning for multicore platforms.

In this paper, we present pOSKI, an autotuning framework for sparse matrix-vector multiplication to achieve high performance across variety of multicore architectures (the "p" in pOSKI stands for "parallel"). Since its predecessor OSKI supports autotuning only for uniprocessors, where the most important optimization is the data structure with register blocking, we extend OSKI to support an additional set of optimizations for diverse multicore platforms. Our new optimizations include the following (see Section 3 for more details): (1) We do off-line autotuning of in-core optimizations for the individual register blocks into which we decompose the sparse matrix. These optimizations include SIMD instrinsics, software prefetching and software pipelining, to exploit in-core resources such as private caches, registers and vector instructions. (2) We do run-time autotuning of thread-level parallelism, to optimize parallel efficiency. These optimizations include array padding, thread blocking and NUMA-aware thread mapping, to exploit parallel resources such as the number of cores, shared caches and memory bandwidth. (3) We reduce the non-trivial tuning cost at run-time by parallelizing the tuning process, and by using history data, so that prior tuning results can be reused.

We conducted autotuning experiments on three generations of Intel's multicore architectures (Nehalem, Sandy Bridge-E and Ivy Bridge), and two generations of AMD's (Santa Rosa and Barcelona), with ten sparse matrices from a wide variety of real applications. Additionally, we compared our measured performance results to SpMV performance bounds on these platforms using the Roofline performance model [34, 36]; this shows that SpMV is memory bound on all of the platforms and matrices in our test suite. Experimental results show that our autotuning framework improves overall performance by up to 9.3x and 8.6x over the reference serial SpMV implementation and over OSKI, respectively. We also compare to parallel Intel MKL Sparse BLAS Level 2 routine $mkl\_dcsrmv()$ and a straightforward OpenMP implementation, getting speedups of up to 3.2x over MKL and 8.6x over OpenMP.

The rest of this paper is organized as follows. In section 2 we overview SpMV including sparse matrix storage formats and the existing auto-tuning system. In section 3 we describe our autotuning framework for SpMV, including our optimizations spaces for both off-line and run-time tuning. In section 4 we present overviews of the multicore platforms and sparse matrices in our test suite. In section 5, we present our experimental results and analysis comparing to the Roofline performance model, and reference serial and parallel implementations. Finally, we conclude and describe future work in section 6.

2 Background

2.1 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) means computing $y = Ax$ where A is a sparse matrix (i.e. most entries are zero), and x and y are dense vectors. We refer to x as the source vector and y as the destination vector. More generally, we also consider $y = \beta y + \alpha Ax$ where α and β are scalars.

2.1.1 Data Structures

Sparse matrix data structures generally only store nonzero entries along with additional index information to determine their locations. There are numerous possible storage formats, with different storage requirements, memory access patterns, and computing characteristics, see [30, 31, 11, 27, 3, 5, 20, 35, 12, 16, 37]. Here are some examples; for more details see [31]. The simplest sparse format is coordinate (COO) format, which stores both the row and column indices for each nonzero value. Another widely used format is compressed sparse (CSR) format, which stores matrices row-wise, so the row index does not need to be stored explicitly as in COO format. The compressed sparse column (CSC) format is similar to CSR, but stores the matrix column-wise. The diagonal (DIAG) format is designed for sparse matrices consisting of some number of (nearly) full nonzero diagonals. Since each diagonal is assumed to be full, we only need to store one index for each nonzero diagonal, and no indices for the individual nonzero elements. The ELLPACK/ITPACK (ELL) format is designed for the class of sparse matrices in which most rows have the same number of nonzeros. If the maximum number of nonzeros in any row is s, then ELL stores the nonzeros values in a 2D array of size $m \times s$, where m is the number of rows, and a corresponding 2D array of indices. The jagged diagonal (JAD) format was designed to overcome the problem of variable length rows/columns by storing them in decreasing order by the number of nonzeros per row, plus an additional permutation matrix. The skyline (SKY) format is a composite format which stores the strictly lower triangle of the matrix in CSR, the strictly upper triangle in CSC, and the diagonal in a separate array. The block compressed sparse row (BCSR) format is a further improvement of CSR format by using block structure, to exploit the naturally occurring dense block structure typical of matrices arising in finite element method (FEM) simulations ; for more details of BCSR see Section 3.1. The variable block row (VBR) format generalizes the BCSR format by allowing block rows and columns to have variable sizes.

Comparisons of storage formats are reported in [31, 27, 2, 20]. Vuduc [31] reports that CSR tends to have the best performance on a wide class of matrices and on a variety of superscalar architectures (Sun Ultra 2i, Ultra 3, Intel Pentium III, Pentium III-M, Itanium 1 and Itanium 2, IBM Power 3 and Power 4), among the basic formats (including CSR, CSC, DIAG, ELL and JAD) considered for SpMV. He also reports that BCSR with a proper block size achieves up to 4x speedups over CSR, and VBR shows up to 2.1x speedups over CSR. Shahnaz *et al.* [27] review the several storage formats (including COO, CSR, CSC, JAD, BCSR, and DIAG) for sparse linear systems. They report that COO, CSR and CSC are quite similar to each other with the difference in column and row vector, and CSR has minimal storage requirements. BCSR is useful when the sparse matrix is compressed using square dense blocks of nonzeros in some regular patterns, however it does not perform significantly better with different block sizes. Bell *et al.* [2] reported the the hybrid format (ELL + COO format) is generally fastest for a broad class of unstructured matrices, comparing among the basic formats (COO, CSR, DIAG, and ELL) on a GPU (GeForce GTX 280); block storage formats (BCSR, VBR) are listed as future work. Karakasis *et al.* [20] conduct a comparative study and evaluation of block storage formats (including BCSR and VBR formats) for sparse matrices on multicore architectures. They report that one-dimensional VBR provides the best average performance while the best storage format depends on a matrix and underlying architecture. They also report that BCSR can provide more than 50% performance improvement over CSR, but it can lead to more than 70% performance degradation when selecting improper blocks on a variety of multicore architectures (Intel Hapertown, Intel Nehalem and Sun UltraSPARC T2 Niagara2). As demonstrated in this literature, the performance benefit from a particular storage format depends strongly on nonzero pattern and underlying micro-architectures. Furthermore, the performance benefit from block storage formats depends on selecting proper block size.

Additionally, to increase the effectiveness of those data structures, reordering rows and columns of matrix can be used, since this can increase the available block structure. There are numerous reordering algorithms such as Column count [14], Approximate Minimum degree (AMD) [1], reverse Cuthill-McKee (RCM) [9], King's algorithm [21], and the Traveling Salesman Problem (TSP) [25].

2.2 OSKI: An autotuning System for SpMV

For sparse matrix computations, OSKI, based in large part on the earlier SPARSITY framework [16, 18], has successfully generated automatically tuned sparse kernels for cache-based superscalar uniprocessors. The kernels include SpMV and sparse triangular solve (SpTS), among others. OSKI automatically searches over the several sparse storage formats and optimizations to find the data structure and tuned code that best exploit properties of both the sparse matrix and the underlying micro-architecture. The sparse storage formats include CSR, CSC, BCSR and VBR, and the optimizations include register blocking and loop unrolling. Since the nonzero pattern of the sparse matrix is not known until run-time, the need for run-time tuning differs from the dense case where only off-line tuning has proved sufficient in practice [4, 33]. To reduce cost of run-time tuning, OSKI has two phases, off-line (compile-time) and run-time. The first phase is an off-line benchmarking phase to characterize the performance of possible implementations on the given machine. The second is a run-time search consisting of (a) estimating relevant matrix structural properties, followed by (b) evaluating a heuristic performance model that combines the estimated properties and benchmarking data to select an implementation. The heuristic performance model usually chooses an implementation within 10% of the best implementation found by exhaustive search [31]. However, OSKI only supports autotuning for cache-based superscalar uniprocessors.

3 pOSKI's Approach

We extend OSKI's autotuning framework for SpMV, shown in Figure 1, to support additional optimizations for multicore platforms. Our work extends OSKI as follows: (1) We perform off-line autotuning of in-core optimizations for performance on individual register blocks. (2) We perform run-time autotuning of thread-level parallelism to optimize parallel efficiency. (3) We reduce the non-trivial tuning cost at run-time by parallelizing the tuning process, and by using history data, so that prior tuning results can be reused. Based on prior work of our own and others [3, 16, 23, 19, 20, 24, 27, 28, 31, 32, 33, 34, 35, 36], we present the set of the possible optimization strategies for multicore architectures, classified into three areas as shown in Table 1: (1) Data structures, (2) In-core optimizations and (3) Thread-level parallelism. The choice of data structures depend mostly on the nonzero pattern of the sparse matrix, the in-core optimizations depend mostly on single core architectural resources, and the thread-level parallelizations depend mostly on parallel architectural resources. In this paper we consider only the subset of optimizations which are marked with an asterisk(*) in Table 1; implementing the others is future work. Note that low- and high-level blocking and partitioning also influence the data structure. Array padding is useful to avoid conflict misses. Index compression is useful to reduce memory traffic when it can use fewer bits to store column and row information.

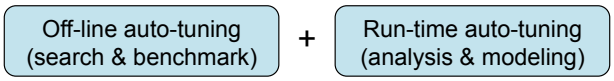


Figure 1: Overview of autotuning framework for SpMV.

Data structures	**In-core optimizations**	**Thread-level Parallelism**
- Storage formats (*CSR, *BCSR, etc.) - Reordering rows and columns - Index compression	- Low-level blocking (*Register, Cache, TLB blocking) - *Software pipeline - *Software prefetching - *SIMDization - *Loop unrolling	- High-level blocking (*Thread blocking by row blocks) - Other partitioning schemes - *NUMA-aware mapping (process and memory affinity) - *Array padding

Table 1: Overview of possible SpMV optimizations for multicore architectures. Asterisks (*) denote the optimizations considered in this paper.

3.1 Block Compressed Sparse Row (BCSR)

We use BCSR storage format for A, because BCSR format can achieve reasonable performance improvements compared to CSR when the proper register block size is selected, and OSKI's autotuning heuristic has been shown to inexpensively select the proper size. We treat each block of BCSR as a dense block, which may require filling in explicit zeros. Register blocks are used to improve register reuse (i.e. locality) by storing as much as possible of the matrix as sequence of small dense blocks, keeping corresponding small blocks of vector x and y in registers for SpMV. In BCSR format with $r \times c$ blocks, an $m \times n$ sparse matrix in CSR format is divided into up to $(m/r) \times (n/c)$ blocks, each of size $r \times c$. An example of 2×2 BCSR format is shown in Figure 2. BCSR format stores only one column index per register block, and one row pointer per block row starting position in the array of column indices. The memory requirement for BCSR is therefore $O((r \times c \times k) + (m/r + 1) + (k))$, where k is the number of blocks. Thus BCSR format can store fewer row pointers and column indices than CSR, but at the possible cost of filling in explicit zeros, and so increasing storage and arithmetic.

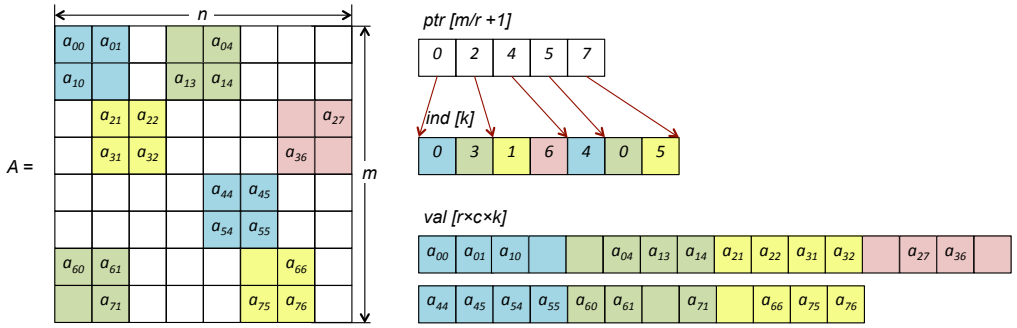


Figure 2: Example of 2×2 BCSR storage format for an 8×8 sparse matrix. Nonzero values including explicit zeros are stored in the *val* array. The column index of the $(0, 0)$ entry of each block is stored in *ind* array. The *ptr* array points to block row starting positions in the *ind* array.

3.2 Off-line (compile-time) autotuning

Even though we cannot choose the best register block size until run-time when the input matrix structure is known, the fastest implementations for all likely $r \times c$ block sizes can be selected off-line, by autotuning over a design space of in-core optimizations including loop unrolling, SIMD intrinsics, software prefetching, and software pipelining. Additionally, we can tune over the set of available compilers and compiler optimization flags.

Our off-line autotuning system performs three major operations as shown in Figure 3: (1) It automatically generates codes for various implementations with a set of tunable parameters. The parameters and their ranges are shown in Table 2; thus $8 \cdot 8 \cdot 4 \cdot 3 = 768$ implementations are generated altogether (not counting compilers and their flags). (2) It collects benchmarking data for each of these implementations for a dense matrix stored in sparse matrix format on each micro-architecture of interest. This can take a while, but is only done once per architecture and compiler. (3) It selects the best implementation for each register block size, storing all the data for later use, including the best observed performance $P_{rc}(dense)$ in Mflops/sec for each $r \times c$ block size, and all the relevant hardware and software environment parameters (OS, compiler, compiler flags, etc.) needed to make the results reproducible. The benchmark and related data are stored in *.lua* files (using the embedded scripting language Lua), and the implementations are stored in a shared object *.so* file.

Our off-line autotuning is designed to be extensible using a Python-based code generation infrastructure. This leads to more compact and maintainable code in which adding new machine-specific intrinsics for prefetching and SIMD support is straightforward; new architectural concepts like co-processors require a more

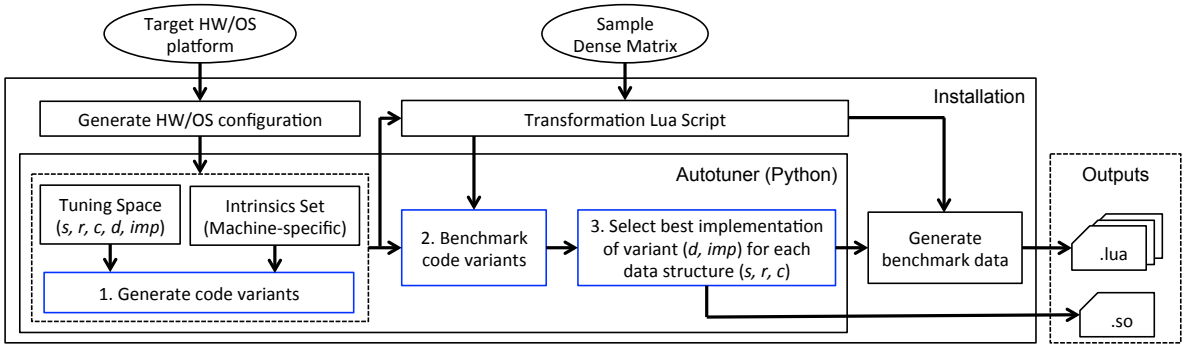


Figure 3: Off-line autotuning phase. The sample dense matrix is stored in sparse storage format. The generated benchmark data is stored in files (*.lua*) using Lua, a lightweight embedded scripting language. The selected implementations are compiled based on HW/OS configuration into a shared object (*.so*).

Tuning Space	
Tunable parameter	Range of tunable parameter
Storage format (s)	$s \in \{ CSR, BCSR \}$
Number of rows of register block (r)	$1 \leq r \leq 8$
Number of columns of register block (c)	$1 \leq c \leq 8$
*Software prefetching distance (d) in Bytes	$d \in \{ 0, 64, 128, 256 \}$
*SIMD implementation (imp)	$imp \in \{ none, SIMD_{row}, SIMD_{col} \}$

Table 2: Tunable parameters for off-line autotuning. Software prefetching distance $d = 0$ means no prefetching is done. The SIMD implementation denotes the different computation orders to try to facilitate efficient use of SIMD intrinsics with pipelined and balanced additions and multiplications: SIMD_{row} indicates a row-wise implementation, and SIMD_{col} indicates column-wise, and none indicates no intrinsics are used (so it is up to the compiler). All SIMD implementations for each register block are fully unrolled. Choosing $d = 0$ and imp =none corresponds to the implementation in OSKI. The tunable parameters with asterisks (*) require the machine-specific intrinsics set.

substantial extension to the code generator, but can still be done in a modular way within the framework.

3.3 Run-time (on-line) autotuning

Run-time autotuning is performed only after the locations of the nonzeros of the sparse are known. The three major autotuning steps are shown in Figure 4.

In Step 1, we partition the sparse matrix into submatrices, to be executed by independent threads in the existing threadpool. As described later, we try to have one thread per core (or more if hyperthreading is supported), accessing a submatrix pinned to the memory most local to that core, i.e. we use a nonuniform-memory access (NUMA) aware mapping. pOSKI currently partitions the matrix into consecutive row blocks (one-dimensional row-wise partitioning) with roughly equal numbers of nonzeros per block, in order to balance the load. There are many other possible ways to partition into submatrices, such as one-dimensional column-wise or two-dimensional partitioning schemes with graph or hypergraph partitioning models [7, 8]; implementing these is future work.

As discussed in more detail in Section 5, it is not always fastest to use all available cores, so choosing the optimal number of cores is part of the autotuning problem. Trying all subsets of the available cores at run-time is too expensive, so doing this tuning well is future work; pOSKI currently just uses all available cores that are provided. In Section 5 we suggest a natural simple formula for the optimal number of cores, based on measured performance data, but show that it only predicts the right number of cores for some test matrices and some platforms.

In steps 2 and 3, we need to choose the best data structure and implementation for each submatrix (each

submatrix may have a different best choice). Since it is too expensive to exhaustively try all implementations described in the last section, we use the OSKI heuristic performance model [31, 32] to quickly select the block size $r \times c$ likely to be fastest (more details below). Alternatively, if the user supplies a "hint" that we could use the same data structure and implementation as used before (based on a user-selected matrix name), this could more quickly be obtained from a "history database" (a SQLite database *.db* file). Conversely, tuning results from using the heuristic performance model can be stored in the database for future use.

Finally each submatrix is copied into its new format. This is often the most expensive autotuning step. All autotuning steps after the matrix is partitioned are performed in parallel by different threads.

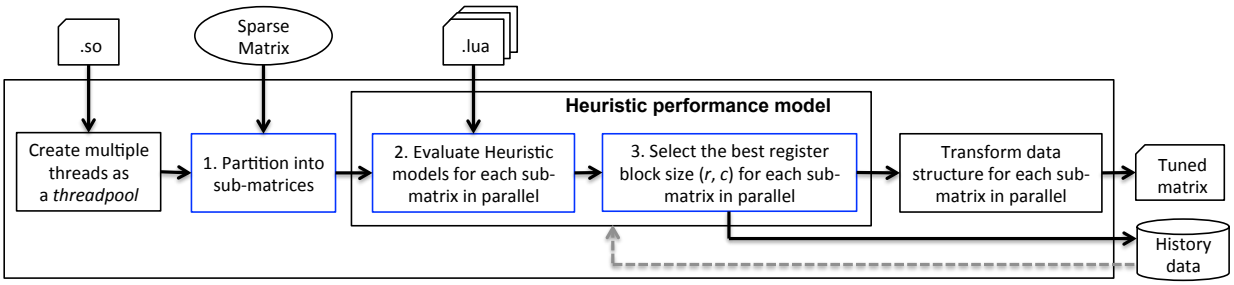


Figure 4: Run-time autotuning phase. The outputs of off-line autotuning, benchmark data (*.lua*) and tuned codes (*.so*) are used for run-time autotuning.

Table 3 summarizes the run-time design space. We note that pOSKI currently only implements 1D row-wise partitioning, other partitioning schemes are future work.

Tunable parameters	Range of tunable parameters
Partitioning scheme (Thread blocking)	1D row-wise
Storage format (s)	$s \in \{ CSR, BCSR \}$
Number of rows of register block (r)	$1 \leq r \leq 8$
Number of columns of register block (c)	$1 \leq c \leq 8$

Table 3: Tunable parameters for run-time autotuning.

3.3.1 Thread-level Parallelization

As shown in Figure 4, we first implement a *threadpool*, reusable multiple threads using thread affinity and a spin lock, based on POSIX Threads (Pthreads) API for thread-level parallelism. We use a first-touch allocation policy to allocate submatrices to the closest DRAM interface to the core that will process them. We also use the affinity routines to pin the each thread to particular core. As mentioned earlier, it is important to use a NUMA-aware mapping to minimize memory access costs. Additionally, our NUMA-aware mapping attempts to maximize usage of multicore resources (from memory bandwidth to physical cores in an architectural hierarchy on a multicore platform) by scaling with the number of threads. This must be handled carefully since the physical core ID depends on the platform. We pad the unit-stride dimension to avoid cache line conflict misses for each sub-matrix.

As mentioned before, so far we only implement a one-dimensional row-wise partitioning scheme, and attempt to load balance by approximately equally dividing the number of non-zeros among thread blocks. However, this can result in a load balance problem between submatrices that use different register block sizes $r \times c$ and so run at different speeds. Future work on other partitioning schemes will address this potential problem.

3.3.2 Heuristic performance model

We briefly describe OSKI's heuristic performance model [31, 32] that we also use to quickly select the optimal register block size $r \times c$ for each submatrix: We choose r and c to maximize the following performance estimate $\hat{P}_{rc}(A_s)$ for each sub-matrix A_s of A:

$$\hat{P}_{rc}(A_s) = \frac{P_{rc}(dense)}{\hat{f}_{rc}(A_s)} \tag{1}$$

Here $P_{rc}(dense)$ is the measured performance value (in Mflops/sec) of SpMV for a dense matrix in $r \times c$ BCSR format (from the off-line benchmark data), and $\hat{f}_{rc}(A_s)$ denotes the estimated fill ratio of A_s caused by storing explicit zeros in the $r \times c$ register blocks:

$$\hat{f}_{rc}(A_s) = \frac{(number\ of\ true\ nonzeros) + (number\ of\ filled\ in\ explicit\ zeros)}{number\ of\ true\ nonzeros} \tag{2}$$

$\hat{f}_{rc}(A_s)$ is estimated cheaply by statistical sampling of A_s [31, 32].

3.3.3 History data

Despite our use of a heuristic to quickly estimate $\hat{f}_{rc}(A_s)$, run-time tuning still has a non-trivial cost. Vuduc [31] reported that the total cost of run-time tuning could be at most 43 unblocked SpMV operations on the matrices and platforms in his test suite. Since many users often reuse the same matrix structure in different runs, this motivates us to quickly reuse previously computed tuning data, by keeping it in a database. We still have to pay the costs of searching the database (given the matrix name, dimensions, number of nonzeros, and number of submatrices), and converting the matrix from CSR format to the optimized format (stored by the dimensions, number of nonzeros, and optimal $r \times c$ for each submatrix). To manage history data, we use SQLite C/C++ interfaces.

4 Experimental Setup

Before we discuss the measured performance results in the following section, we will briefly summarize characteristics of multicore platforms in our test suite, discuss an SpMV performance bound for our multicore platforms, and present an overview of the sparse matrices.

4.1 Evaluated multicore platforms

In Table 4, we summarize characteristics of multicore platforms in our test suite, including architectural configuration, system peak performance, and compiler. The system peak performance is for double-precision floating point operations, considering neither the max turbo frequency nor AVX, although some of the platforms support these techniques. The system peak bandwidth is the peak DRAM memory bandwidth in billions of bytes transfered per second (GB/s). The system flop:byte ratio is the ratio of the system peak performance to the system peak bandwidth.

Gainestown (Nehalem) platform is dual-socket quad-core, so 8 cores altogether, with NUMA support. Each core runs at 2.66 GHz, with two hardware threads (hyper threading) to allow simultaneous execution of one 128b SSE multiplier and one 128b SSE adder. The peak double-precision floating point performance per core is therefore 10.64 GFlops/s, and the system peak double-precision floating point performance is 85.12 GFlops/s. Each socket with three fully buffered DDR3-1066 DRAM channels can deliver 21.3 GB/s. The system peak bandwidth is therefore 52.6 GB/s.

Jaketown (Sandy Bridge-E) platform is single-socket six-core. Each core runs at 3.3 GHz, with two hardware threads to allow simultaneous execution of one 128b SSE multiplier and one 128b SSE adder. The peak double-precision floating point performance per core is therefore 13.2 GFlops/s, and the system peak

Platform	Intel			AMD	
	Gainestown	Jaketown	Ivy	Santarosa	Taurus
Core Architecture Model No.	Nehalem Xeon X5550	Sandy bridge-E Core i7-3960X	Ivy bridge Core i5-3550	Santa Rosa Opteron 2214	Barcelona Opteron 2356
Core GHz (Max)	2.66 (3.06)	3.3 (3.9)	3.3 (3.7)	2.2 (-)	2.3 (-)
# Sockets	2	1	1	2	2
Cores/Socket	4	6	4	2	4
HW-threads/Core	2	2	1	1	1
SSE (AVX)	128bit (-)	128bit (256bit)	128bit (256bit)	128bit (-)	128bit (-)
L1D cache (private)	64KB	32KB	32KB	64KB	64KB
L2 cache (private)	256KB	256KB	256KB	1MB	512KB
L3 cache (shared)	2×8MB	15MB	6MB	-	2×2MB
NUMA	Yes	Yes	No	Yes	Yes
DRAM Type (channels per socket)	DDR3-1066 (3×64b)	DDR3-1600 (4×128b)	DDR3-1600 (2×128b)	DDR2-667 (1×128b)	DDR2-667 (2×64b)
DP GFlops/s	85.12	79.2	26.4	17.6	73.6
DRAM GB/s	2×21.3	2×25.6	25.6	2×10.66	2×10.66
DP flop:byte ratio	2	1.55	1.03	0.83	3.45
Compiler	icc 12.0.4	icc 12.0.4	icc 12.0.4	gcc 4.3.1	gcc 4.3.2

Table 4: Overview of evaluated multicore platforms. The Gainestown, Jaketown and Ivy platforms enable each core to run at max turbo frequency (Max). Each L1D and L2 cache is a private cache per core, and each L3 cache is a shared cache per socket for all platforms. The system peak performance (DP GFlops/s) is for double-precision floating-point performance. The system peak memory bandwidth (DRAM GB/s) is the peak DRAM memory bandwidth in GB/s. The system flop:byte ratio (DP flop:byte ratio) is the ratio of the system peak performance to the system peak memory bandwidth for double-precision floating-point performance.

double-precision floating point performance is 79.2 GFlops/s. The system peak bandwidth on single-socket with four fully buffered DDR3-1600 DRAM channels is 51.2 GB/s.

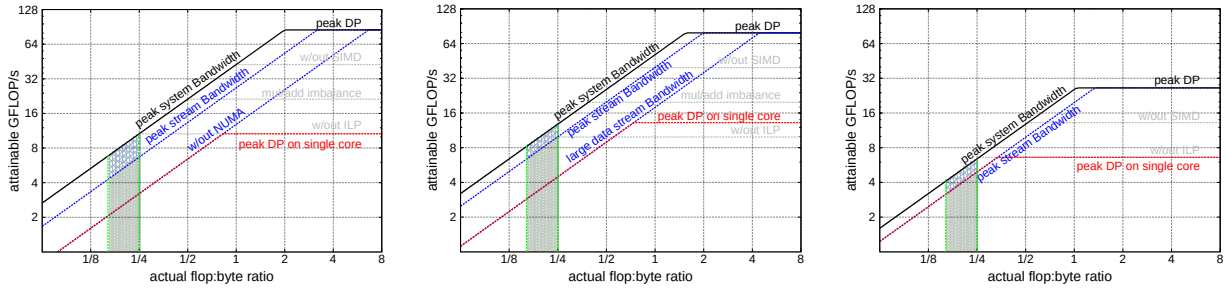
Ivy (Ivy Bridge) platform is single-socket quad-core. Each core runs at 3.3 GHz, win only one hardware thread with 128b SSE instructions. The peak double-precision floating point performance per core is therefore 6.6 GFlops/s, and the system peak double-precision floating point performance is 26.4 GFlops/s. The system peak bandwidth on single-socket with two fully buffered DDR3-1600 DRAM channels is 25.6 GB/s.

Santarosa (Santa Rosa) platform is dual-socket dual-core, so 4 cores altogether, with NUMA support. Each core runs at 2.2 GHz, can fetch and decode three x86 instructions per cycle, and executes 6 micro-ops per cycle. The each core supports 128b SSE instructions in a half-pumped fashion, with a single 64b multiplier datapath and a 64b adder datapath, thus requiring two cycles to execute an SSE packed double-precision floating point multiply. The peak double-precision floating point performance per core is 4.4 GFlops/s, and the system peak double-precision floating point performance is 17.6 GFlops/s. Each socket with one fully buffered DDR2-667 DRAM channel can deliver 10.66 GB/s. The system peak bandwidth is therefore 21.33 GB/s.

Taurus (Barcelona) platform is dual-socket quad-core, so 8 cores altogether, with NUMA support. Each core runs at 2.3 GHz, can fetch and decode four x86 instructions per cycle, executes 6 micro-ops per cycle, and fully supports 128b SSE instructions. The peak double-precision floating point performance per core is 9.2 GFlops/s, and the system peak double-precision floating point performance is 73.6 GFlops/s. Each socket with two fully buffered DDR2-667 DRAM channels can deliver 10.66 GB/s. The system peak bandwidth is therefore 21.33 GB/s.

4.2 Performance predictions

For the expected range of SpMV performance on the above multicore platforms, we adapt the simple roofline performance model [36, 34]. In Figure 5, the system peak performance and the system peak bandwidth (black lines) are derived from the architectural characteristics as shown in Table 4, and the stream bandwidth (blue dashed lines) is the measured memory bandwidth obtained via Stream benchmark [29]. We measure the stream bandwidth with both small and large data sets. The performance of the stream bandwidth with large data set can indicate the performance without NUMA support for NUMA architectures (or the peak DP on single core - red lines). The arithmetic intensity is the ratio of compulsory floating-point operations (Flops) to compulsory memory traffic (Bytes) of SpMV. In CSR storage format, SpMV performs $2 \times nnz$ floating-point operations, and each nonzero is represented by a double-precision value (8-Bytes) and a integer column index (4-Bytes), where nnz is the total number of nonzeros. Therefore, each SpMV must read at



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform

(d) Santarosa platform (e) Taurus platform

	Memory bandwith (GB/s)			Performance for SpMV (GFlops/s)					
				CSR (actual flop:byte ratio = 1/6)			BCSR (actual flop:byte ratio = 1/4)		
	System peak	Stream peak		System peak	Stream peak		System peak	Stream peak	
		small	large		small	large		small	large
Gainestown	41.6	26.71	12.85	6.93	4.45	2.14	10.40	6.68	3.21
Jaketown	51.2	39.79	17.9	8.53	6.63	2.98	12.80	9.95	4.48
Ivy	25.6	19.7	18.71	4.27	3.28	3.12	6.40	4.93	4.68
Santarosa	21.33	12.12	6.69	3.56	2.02	1.12	5.33	3.03	1.67
Taurus	21.33	16.5	6.9	3.56	2.75	1.15	5.33	4.13	1.73

(f) Summary of performance model

Figure 5: The roofline performance model of SpMV for our evaluated multicore platforms. The black line denotes system peak performance bound. The gray dashed line denotes system performance without indicated optimizations (SIMD, mul/add imbalance, ILP). The blue dashed line denotes measured stream bandwidth. The red dashed line denotes the peak single core performance bound limited by the stream bandwidth with a large data set. The green dashed lines denotes bounds of the arithmetic intensity, actual flop:byte ratio, for SpMV. Note the log-log scale.

least $12 \times nnz$ bytes. In addition, we assume no cache misses associated with the input/output vectors and the row pointers of CSR. As a result, SpMV has a compulsory arithmetic intensity less than $2 \times nnz$ / $12 \times nnz$, or about 0.166 (leftmost vertical green lines). In BCSR storage format, the register blocking encodes only one column index for each register block. As register blocks can be up to 8×8, it is possible to amortize this one integer index among many nonzeros, raising the Flop_byte ratio from 0.166 to 0.25 when no explicit zeros are added in BSRC format (shown by the rightmost vertical green lines). By the roofline performance model, we expect SpMV to be memory bound on all of our evaluated platforms, since the two vertical green lines intersect the slanted (memory bound) part of the roofline. Thus, we can also expect that it can achieve the system peak performance of SpMV with fewer cores.

4.3 Sparse matrix test suite

We conduct experiments on sparse matrices from a variety of real applications, including nonlinear optimization, power network simulation, and web-connectivity analysis, available from the University of Florida sparse matrix collection [10], as well as a dense matrix stored in sparse matrix storage format. These matrices cover a range of properties relevant to SpMV performance, such as overall matrix dimension, non-zeros per row, the existence of dense block substructure, and space required in CSR format. An overview of their characteristics appears in Figure 6. Note that Matrix 1 is a dense matrix, and Matrices 2-4 are small matrices which might fit into the higher level of caches, and Matrices 5-10 are large matrices. From their densities, we expect that some matrices (Matrices 2, 4, 7, 10) might not benefit from BCSR, while others (Matrices 1, 3, 5, 6, 8, 9) might.

#	**Name** Description	Spyplot	Dimensions Nonzeros	ave. nnz/row (nonzero density $r = c$ in {2, 4, 8})	Space required in CSR format
1	**Dense** Dense matrix in sparse format		4K × 4K 16.0M	4K (100, 100, 100)%	144 MB
2	**poisson3Da** Computational fluid dynamics		1.3K × 1.3K 0.3M	230 (28, 8, 2)%	4 MB
3	**FEM-3D-thremal1** Thermal problem		1.7K × 1.7K 0.4M	235 (57, 35, 24)%	5 MB
4	**ns3Da** Computational fluid dynamic		20K × 20K 1.6M	3.6 (28, 8, 2)%	19 MB
5	**Largebasis** Optimization problem		440K × 440K 5.24M	11.9 (92, 45, 28)%	64 MB
6	**Tsopf** Power network problem		35.7K × 35.7K 8.78M	246 (99, 93, 84)%	100 MB
7	**Kkt_power** Optimization problem		2.06M × 2.06M 12.77M	6.2 (43, 9, 3)%	154 MB
8	**Ldoor** Structural problem		952K × 952K 42.49M	44.6 (78, 56, 33)%	490 MB
9	**Bone010** (2D trabecular bone) Model reduction problem		968K × 968K 47.85M	48.5 (56, 41, 25)%	552 MB
10	**Wiki-2007** (Wikipedia pages) Directed graph problem		3.56M × 3.56M 45.4M	12.6 (26, 7, 2)%	530 MB

Figure 6: Overview of sparse matrices used in evaluation study.

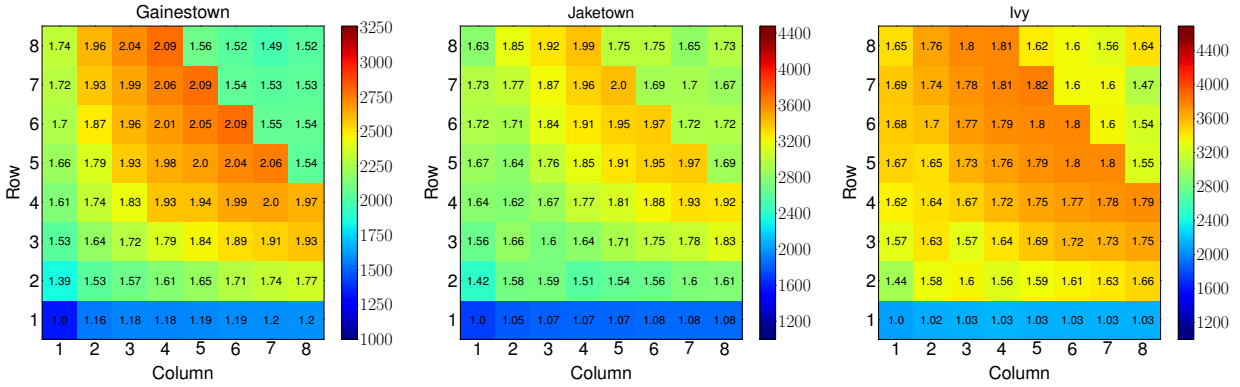
5 Experimental Results and Analysis

Here we present our measured SpMV performance on a variety of sparse matrices and multicore platforms. We begin by comparing performance of our off-line autotuning framework to OSKI and the roofline model (on dense matrices in BCSR format). We compare our auto-tuned implementations to reference serial SpMV with CSR, to OSKI, to the parallel Intel MKL Sparse BLAS Level 2 routine, and to a straightforward OpenMP

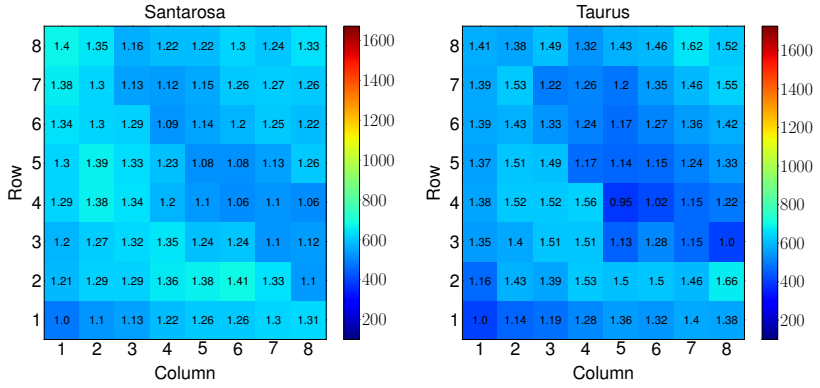
implementation. We also compare our performance to the predicted performance, and finally discuss future potential improvements of our autotuning system. Note that we use 8 byte double-precision floating-point for matrix and vector values, and 4 byte integers for column indices and row pointers in sparse formats. All implementations are compiled with Intel *icc* on Intel platforms and *gcc* on AMD platforms with the average results shown (average over 50 runs for off-line results and 100 runs for run-time results).

5.1 Off-line (compile-time) autotuning

Here we discuss the performance gain on a single core by our off-line autotuning for SpMV with a dense matrix in sparse matrix format, comparing results to OSKI and the roofline model. See Section 3.2 for a discussion of the tuning-parameters and code generation. For all timing measurements, we average the results over 50 runs.



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform



(d) Santarosa platform (e) Taurus platform

Figure 7: SpMV Performance Profiles for OSKI on a dense matrix on a single core. On each platform, each square is an $r \times c$ implementation, for $1 \leq r, c \leq 8$, colored by its performance in MFlops/s, and labeled by its speed up over the reference CSR implementation ($r = c = 1$). The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5.

5.1.1 OSKI baseline

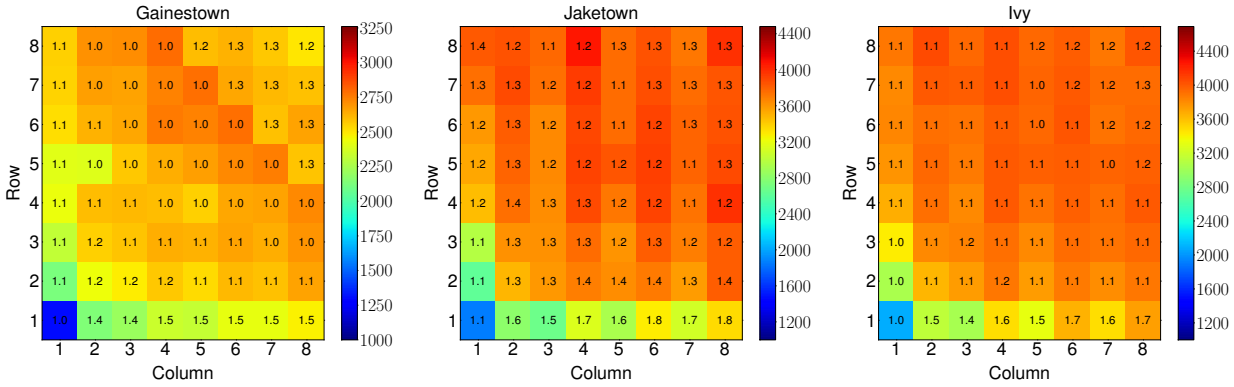
Figure 7 shows the SpMV performance gain by using BCSR format with register block size r and c compared to CSR, for a dense matrix in (B)CSR format. This implementation is the same as OSKI's (B)CSR

implementation, i.e. the code is fully unrolled, but there is no explicit software prefetching or use of SIMD intrinsics. Each plot in Figure 7 shows all 64 implementations, for $1 \leq r, c \leq 8$, each colored by its performance in MFlops/s, and labeled by its speedup over the reference CSR implementation ($r = c = 1$).

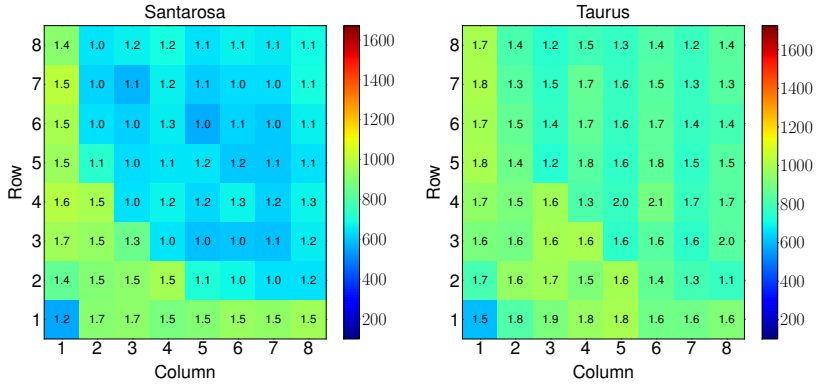
OSKI achieves up to 2.1x, 2x, 1.8x, 1.4x and 1.7x speedups over CSR for a dense $4K \times 4K$ matrix on the Gainestown, Jaketown, Ivy, Santarosa and Taurus platform, respectively. We observe that the peak performance is near $r \cdot c = 36$ for Intel's three platforms, 12 for Santarosa and 16 for Taurus. However, the performance of register blocks with a single row ($r = 1$) is not much improved in contrast to taller blocks ($r > 1$), largely because the register reuse is limited by having a single value of the output vector, resulting in a read-after-write dependency. We also observed that Intel platforms (up to 2.1x speedup over CSR) show slightly better performance gains by register blocking than AMD platforms (up to 1.7x speedup over CSR).

5.1.2 pOSKI's in-core optimizations

Figure 8 shows the SpMV performance gain by off-line autotuning of in-core optimizations (see Table 2) for each register block size $r \times c$. We refer to this implementation as optimized-BCSR. Each plot shows all 64



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform

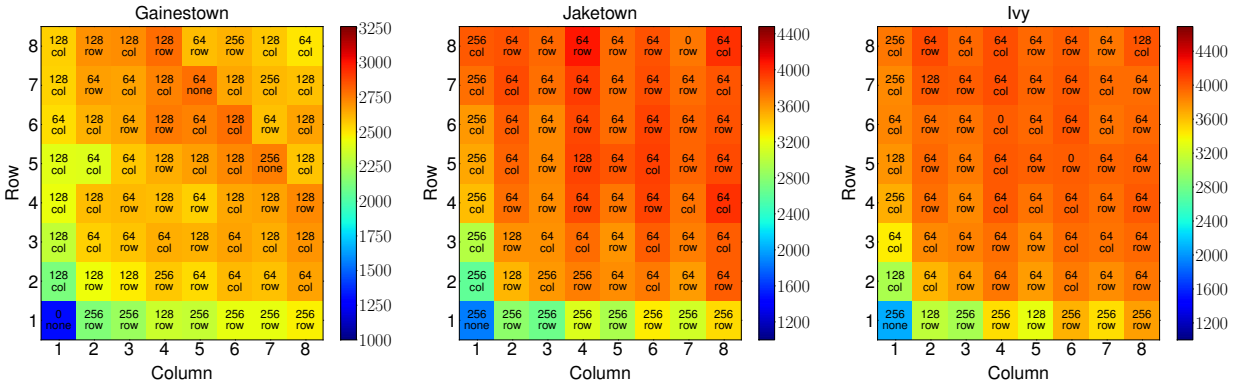


(d) Santarosa platform (e) Taurus platform

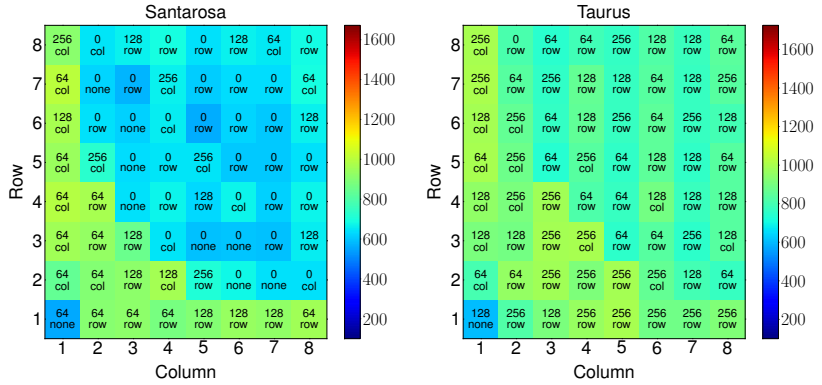
Figure 8: SpMV Performance Profiles for optimized-BCSR on a dense matrix on a single core. On each platform, each square is an $r \times c$ implementation, $1 \leq r, c \leq 8$, colored by its performance in MFlops/sec, and labeled by its speedup over OSKI. The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5.

implementations, each colored by its performance in MFlops/s, and each labeled by its speedup over OSKI's performance (from Figure 7).

By autotuning each register block size off-line, optimized-BCSR achieves up to 1.5x, 1.8x, 1.7x, 1.7x and 2.1x speedups over OSKI on Gainestown, Jaketown, Ivy, Santrarosa and Taurus, respectively. Although the x86 architectures support hardware prefetching to overcome memory latency from L2 to L1, our use of software prefetching still helps performance by improving locality in L2. For example, we observe on Taurus that optimized-BCSR with $r = c = 1$ is 1.5x faster than OSKI, by choosing the proper software prefetching distance. In-core optimizations are more helpful for blocks with a single row ($r = 1$) to overcome the read-after-write dependency problem with a single output. For example, we observe that optimized-BCSR improves the performance up to 90% over OSKI (on Taurus) for register blocks with $r = 1$. On Intel platforms the MFlop rate is still slightly lower for blocks with $r = 1$ than for taller blocks, with $r > 1$. In contrast, on AMD platforms the performance when $r = 1$ or $c = 1$ is often better than for other block sizes. The automatically selected prefetching distance and SIMDization schemes are shown in Figure 9. Each plot shows all 64 implementations, each colored by its performance in MFlops/s, and each labeled by its prefetching distance (upper label) and SIMDization scheme (lower label). In our experience, some decisions vary from run to run because some choices are very close in performance (see Figure 9(f)).



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform



(d) Santarosa platform (e) Taurus platform

$r \times c$		8×5		8×6	
imp		row	col	row	col
d	0	3733	3703	3826	3823
	64	3761	3663	3866	3796
	128	3522	3432	3600	3634
	256	3011	3025	3198	3097

$r \times c$		8×7		8×8	
imp		row	col	row	col
d	0	3722	3721	3641	3665
	64	3530	3573	3844	3998
	128	3408	3452	3677	3841
	256	3110	3066	3185	3174

(f) Jaketown platform

Figure 9: The selected software prefetching distance (d) and SIMDization scheme (imp) for optimized-BCSR on a dense matrix on a single core. On each platform, each square is an $r \times c$ implementation, $1 \leq r, c \leq 8$, colored by its performance in MFlops/sec, and labeled by its d (upper label in each square) and imp (lower label in each square). Note *row* indicates SIMD_{row} (row-wise) and *col* indicates SIMD_{col} (column-wise). The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5. Figure 9(f) shows examples of possible choices in performance (MFlops/s) on Jaketown.

5.1.3 Summary of off-line autotuning

Table 5 summarizes our off-line autotuning performance. pOSKI's optimized-BCSR achieves up to 2.1x, 2.4x, 1.9x, 2.1x and 2.4x speedups over the reference CSR implementation on Gainestown, Jaketown, Ivy, Santarosa and Taurus, respectively. It also improves average performance for all register block sizes with a better (smaller) standard deviation than OSKI (except for Santarosa, where the performance gains mostly occur for small register block sizes, $r \times c \leq 8$). This implies that it will give a better choice when run-time autotuning selects the proper register block size for a sparse matrix. However, optimized-BCSR does not improve the overall peak performance on Gainestown, whereas it improves peak performance up to 40% on other platforms. This shows that some but not all compilers can achieve good optimization for dense register blocks without explicit software prefetching or use of SIMD intrinsics. Comparing against the expected peak performance from the Roofline model, optimized-BCSR on a dense matrix achieves up to 91% of the single core peak performance (stream peak performance with a large data set, as shown in Figure 5 and the rightmost column of Table 5). The peak performance is an empirical upper-bound on SpMV performance with optimized-BCSR on single core, since a dense matrix maximizes data reuse and minimizes irregular memory access. We repeat that off-line tuning can be expensive since it searches over all possible in-core optimizations for each register block size $r \times c$, but occurs only once for each platform.

	CSR	OSKI-BCSR				optimized-BCSR				Expected peak GFlops/s by Roofline (optimized-BCSR peak GFlops/s in %)		
		average		peak		average		peak		system peak	stream peak	
Platform	GF	GF	stdev	$r \times c$	GF	GF	stdev	$r \times c$	GF		small	large
Gainestown	1.3	2.3	17%	8×4	2.8	2.5	8%	6×4	2.8	10.7 (26%)	6.7 (41%)	3.2 (86%)
Jaketown	1.7	2.8	16%	7×5	3.4	3.6	11%	8×4	4.1	12.8 (32%)	9.8 (42%)	4.5 (91%)
Ivy	2.1	3.3	15%	7×5	3.8	3.8	8%	8×4	4.1	6.4 (63%)	4.9 (83%)	4.7 (87%)
Santarosa	0.48	0.6	8%	2×6	0.7	0.7	20%	7×1	1.0	5.3 (19%)	3.0 (33%)	1.7 (60%)
Taurus	0.41	0.6	12%	2×8	0.7	0.9	10%	7×1	1.0	5.3 (19%)	4.1 (24%)	1.7 (58%)

Table 5: Summary of optimized-BCSR single core performance for SpMV on dense matrices, compared to peak performance from the Roofline model. The peak performances of OSKI and optimized-BCSR are taken from Figures 7 and 8. The three Roofline performance bounds are computed using system peak bandwidth (system peak), stream bandwidth with a small data set (small stream peak), and stream bandwidth with a large data set (large stream peak) as shown in Figure 5. GF is performance in GFlops/sec. Averages and standard deviations (stdev, shown as the percent of average) for OSKI and optimized-BCSR are computed from the performance data for all 64 values of $1 \leq r, c \leq 8$. The Roofline performance bound computed from the large stream peak (rightmost column) is the same as the expected single core peak performance. The percentages in the three rightmost columns show peak GF of optimized-BCSR divided by the Roofline performance bound.

5.2 Run-time (on-line) autotuning

Here we measure the SpMV performance gain by the run-time autotuning techniques presented in Section 3.3, as well as the run-time tuning cost, measured as a multiple of the cost of one SpMV on the same matrix. For all timing measurements, we average the results over 100 runs (Note that SpMV performance in GFlops/s does not include run-time tuning cost).

5.2.1 Thread-level Parallelization

To partition a matrix we must choose the number of submatrices into which to divide it, i.e. the number of threads (or cores) to use, since each thread processes one submatrix (where each core runs one thread). Since SpMV is memory bound on all of multicore platforms in our test suite, using too many cores may oversubscribe some memory resources (like total bandwidth) and cause slowdowns. This will be seen below in Figure 10, which shows performance data for varying numbers of cores. (Using too many cores may also

Gainestown	NUMA id	0								1							
	Core id	0		1		2		3		4		5		6		7	
	HW thread id	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15
	Mapping order	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
Jaketown	Channel id	0						1						-			
	Core id	0		1		2		3		4		5					
	HW thread id	0	1	2	3	4	5	6	7	8	9	10	11				
	Mapping order	0	6	2	8	4	10	1	7	3	9	5	11				
Ivy	Channel id	0						-									
	Core id	0		1		2		3									
	Mapping order	0		2		1		3									
Santarosa	NUMA id	0				1				-							
	Core id	0		1		2		3									
	Mapping order	0		2		1		3									
Taurus	NUMA id	0								1							
	Core id	0		1		2		3		4		5		6		7	
	Mapping order	0		4		2		6		1		5		3		7	

Table 6: Thread mapping order for efficient thread-level parallelism. *NUMA id* denotes a group of cores which share the NUMA node. *Channel id* denotes a group of cores which share memory channels. *Core id* denotes a group of HW threads (hyperthreading) which share cache. *HW thread id* denotes a physical core (logical processor) id on our platforms. *Mapping order* denotes our NUMA-aware mapping order when increasing the number of threads.

waste energy, a future autotuning topic). After presenting the data, we will suggest a natural simple formula for the optimal number of cores, based on measured performance data, but show that it only predicts the right number of cores for some test matrices and some platforms; quickly and accurately choosing the optimal number of cores remains future work.

Having selected the number of threads, we need to decide how to map each thread to an available core, because not every subset of cores has the same performance: cores may or may not share critical resources like cache or memory bandwidth. This is important when we only want to use fewer threads than the available number of cores; which cores should we use?

We choose cores using the platform-dependent NUMA-aware mapping shown in Table 6. For example, consider an 8-core Gainestown: The first row, labeled "NUMA id", divides the 8 cores into two groups of 4, according to which fast memory they share. The second row, labeled "core id", gives a unique number to each core in each group of 4. The third row, labeled "HW thread id", give a unique number to each of the two hyperthreads that can run on each core, so numbered from 0 to $2 \cdot 8 - 1 = 15$; the HW thread id is specified by the Gainestown system. Finally, the last row, labeled "Mapping order", tells us in what order to assign submatrices to hyperthreads.

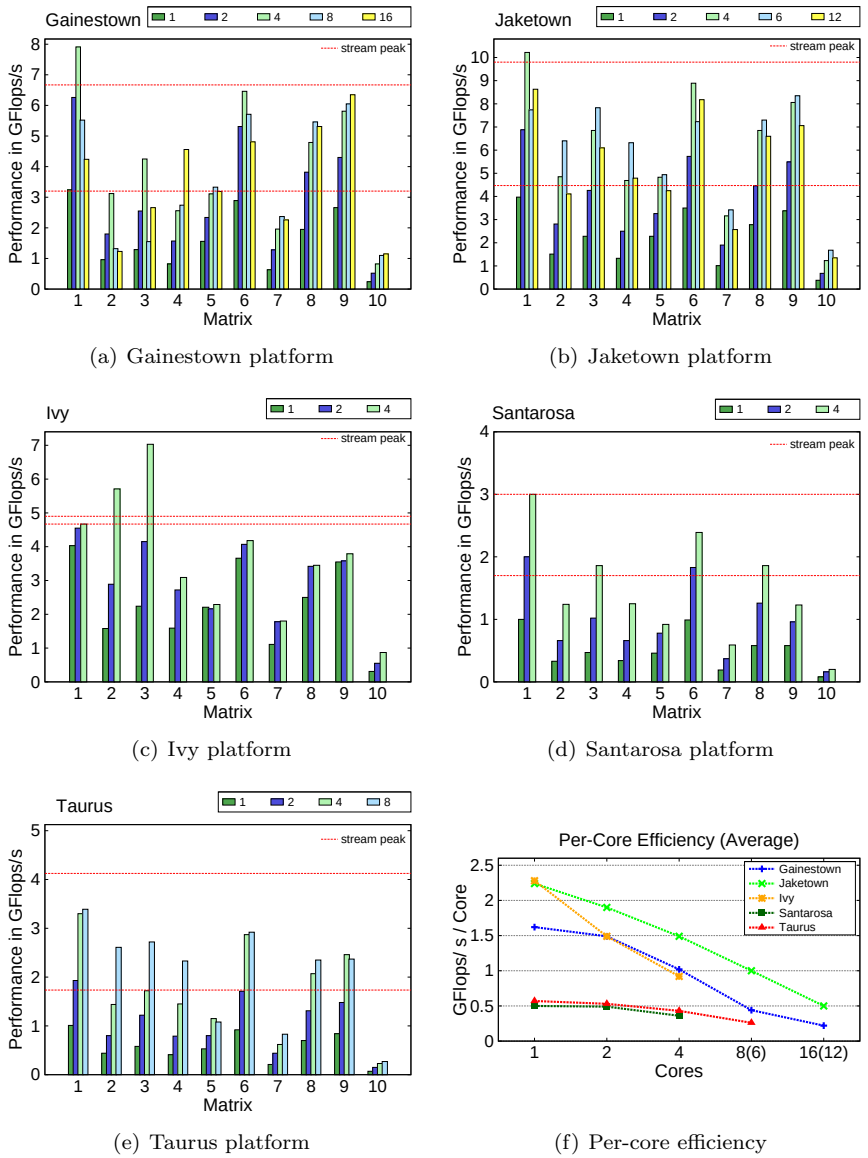
For example, with two submatrices on Gainestown, the first submatrix/thread (Mapping order = 0) pins to the first HW thread (id = 0) in the first NUMA region (id = 0), and the second submatrix/thread (Mapping order = 1) pins to the first HW thread (id = 4) in the second NUMA region (id = 1). This doubles the total hardware resources available, since each submatrix/thread utilizes 21.3GB/s memory bandwidth of each socket, and the 8MB L3, 256KB L2, and 64KB L1 caches on two 2.66GHz cores, so we would expect the largest possible speedup. With, say, seven submatrices on Gainestown, we would use hyperthreads from Mapping order = 0 to Mapping order = 6.

Similar scaling behavior for two submatrices/threads is expected for other NUMA platforms (Jaketown, Santarosa and Taurus). Note that Jaketown is a single socket but the two memory channels have NUMA behavior. However, using two threads on Ivy cannot double memory bandwidth and L3 cache, though they do double the L2 and L1 caches. Thus, we expect that scaling beyond two threads on Ivy may not help for matrices that do not fit in L2; this is borne out in Figure 10.

Figure 10 shows measured SpMV performance of all our test matrices on all platforms, with different numbers of threads. We observe that scaling with two or four threads shows good scalability on most matrices (except on Ivy). We also observe that using too many cores degrades the performance in some cases, in particular Gainestown and Jaketown. Finally, for a number of matrices pOSKI's SpMV exceeds the peak performance predicted by the Roofline model; this occurs frequently when using the Roofline model

with bandwidth measured by the Stream benchmark with a large data set (the lower red dashed lines in Figure 10), and occasionally even using Stream bandwidth with a small data set (the upper red dashed lines). Apparently Stream underestimates the bandwidth of the different memory access patterns accessed by SpMV, and it incurs no data reuse in caches. Also, we recall that small matrices 2 and 3 fit in L3 cache on Gainestown, Jaketown, and Ivy platforms.

Finally, we describe a simple performance model for predicting the optimal number of threads/submatrices to use, and compare it to the results in Figure 10: Take the fraction of system peak performance attained by optimized-BCSR, shown as a percentage in the third column from the right in Table 5; call it x (for example $x = 26\%$ for Gainestown). Then ideally using $1/x$ cores should attain the peak performance



(a) Gainestown platform

(b) Jaketown platform

(c) Ivy platform

(d) Santarosa platform

(e) Taurus platform

(f) Per-core efficiency

Figure 10: The performance of pOSKI on 10 test matrices, on 5 multicore platforms, with varying numbers of cores. Each color on a single matrix denotes the number of threads used to perform SpMV. Two Roofline-based performance bounds are shown, one using bandwidth measured using the Stream benchmark with a large data set (lower red dashed line) and one with a small data set (upper red dashed line).

permitted by the peak system bandwidth on the memory-bound SpMV. Rounding $1/x$ to the nearest integer give the optimal number of cores as $1/26\% \approx 4$ for Gainestown, $1/32\% \approx 3$ for Jaketown, $1/63\% \approx 2$ for Ivy, $1/19\% \approx 5$ for Santarosa, and $1/19\% \approx 5$ for Taurus. However, comparing to the data in Figure 10, we see the optimal number of cores is slightly different than expected due to lower per-core efficiency (shown in Figure 10(f)) by scaling with the number of threads; 4, 8 or 16 for Gainestown, 4 or 6 for Jaketown, 4 for Ivy, 4 for Santarosa, and 4 or 8 for Taurus. Clearly, a better predictor of the optimal number of cores is needed by considering other factors, such as memory latency and size of shared caches, which can affect the per-core performance by scaling with the number of threads.

5.2.2 Results from the Heuristic performance model

Run-time autotuning partitions a sparse matrix into submatrices, and then uses the heuristic performance model from Section 3.3.2 to select the best data structure and SpMV implementation for each submatrix. Thus, each submatrix may have a different data structure, different SpMV implementation, and different performance, possibly upsetting the load balance (improving the load balance is future work).

Table 7 shows in detail the results when we partition a single matrix, Matrix 6 (*Tsopf*), into 4 submatrices, for all 5 platforms. For each submatrix and platform, we show (1) the optimal values of the tuning parameters (r, c, d and imp), (2) the expected performance from the heuristic model $\hat{P}_{rc}(A_s) = P_{rc}(dense)/\hat{f}_{rc}(A_s)$, and (3) the measured performance for each submatrix, both running alone on a single core, and while running in parallel with all cores. As can be seen, different optimal parameters may be chosen for different submatrices on the same platform.

Platform	Id	Tunable parameters				Heuristic model			GFlops/s	
		nnz	$r \times c$	d	imp	$P_{rc}(dense)$	$\hat{f}_{rc}(A_s)$	$\hat{P}_{rc}(A_s)$	serial	parallel
Gainestown	1	2195751	8×2	128	*row*	2.7	1.29	2.09	2.29	1.76
	2	2195456	6×4	128	*row*	2.75	1.05	2.62	3.07	2.02
	3	2195798	6×4	128	*row*	2.75	1.05	2.62	3.07	2.01
	4	2194944	4×4	128	*row*	2.6	1.05	2.48	3.07	2.00
	overall								2.83	6.46
Jaketown	1	2195751	8×4	64	*row*	4.07	1.39	2.93	2.8	2.20
	2	2195456	8×4	64	*row*	4.07	1.07	3.8	3.84	2.36
	3	2195798	8×4	64	*row*	4.07	1.07	3.8	3.81	2.40
	4	2194944	5×4	128	*row*	3.91	1.05	3.72	3.83	2.37
	overall								3.5	8.9
Ivy	1	2195751	8×2	64	*row*	4.06	1.29	3.15	3.11	1.05
	2	2195456	7×2	128	*row*	3.99	1.05	3.8	3.9	1.10
	3	2195798	8×2	64	*row*	4.06	1.05	3.87	3.99	1.10
	4	2194944	7×2	128	*row*	3.99	1.05	3.8	3.89	1.10
	overall								3.68	4.18
Santarosa	1	2195751	3×1	64	*col*	0.96	1.08	0.89	0.86	0.59
	2	2195456	7×1	64	*col*	1.00	1.05	0.95	0.97	0.68
	3	2195798	4×1	64	*col*	0.99	1.05	0.94	0.96	0.68
	4	2194944	7×1	64	*col*	1.00	1.05	0.95	0.96	0.69
	overall								0.98	2.39
Taurus	1	2195751	2×2	64	*row*	0.95	1.05	0.90	0.85	0.71
	2	2195456	5×1	64	*col*	1.00	1.05	0.95	1.02	0.59
	3	2195798	5×1	64	*col*	1.00	1.04	0.96	1.06	0.71
	4	2194944	5×1	64	*col*	1.00	1.04	0.96	1.07	0.59
	overall								0.92	2.92

Table 7: Example of the selected tuning parameters with 4 threads for Matrix 6 (*Tsopf*). *Id* denotes the submatrix, nnz is the number of non-zeros for each submatrix, $r \times c$ is the selected block size, d is the selected software prefetching distance, imp is the selected SIMD implementation, $P_{rc}(dense)$ is measured SpMV performance for a dense matrix with $r \times c$, $\hat{f}_{rc}(A_s)$ is the fill ratio, $\hat{P}_{rc}(A_s)$ is the expected SpMV performance, and GFlops/s is the measured per-core performance in serial and in parallel.

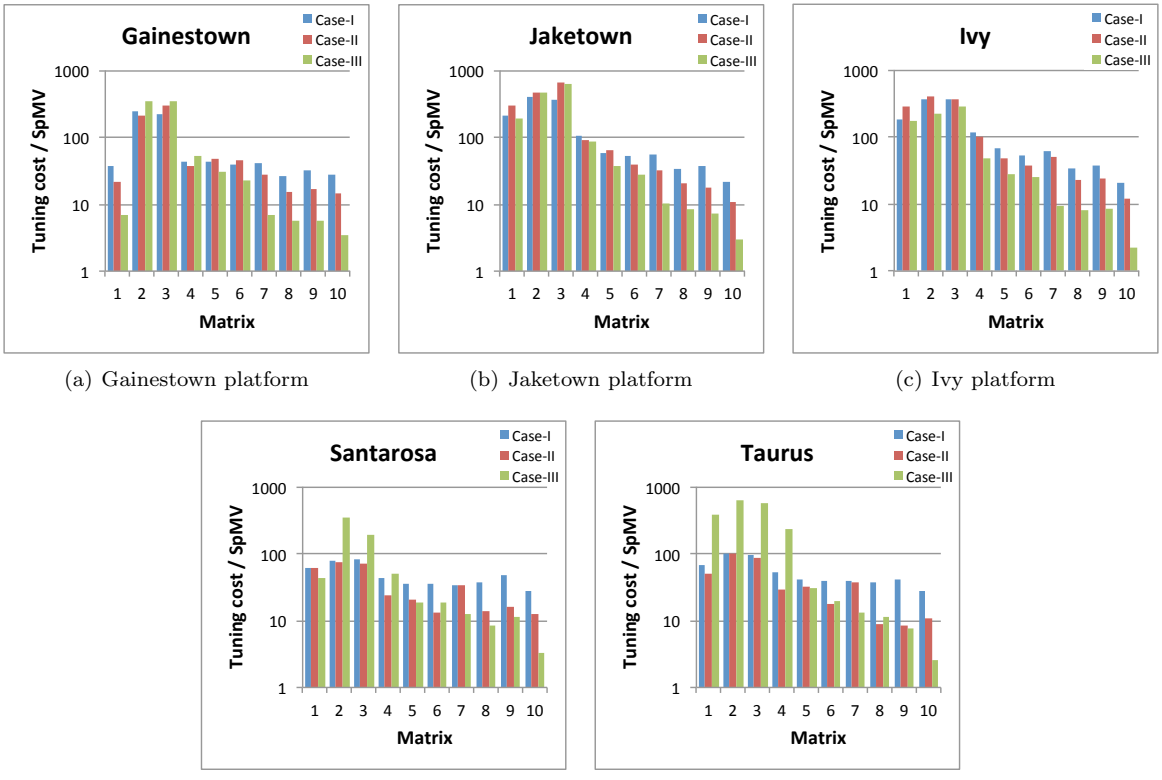
First, to confirm the accuracy of the heuristic model, we compare the measured serial performance (second to last column of Table 7) with the predicted performance (third to last column, $\hat{P}_{rc}(A_s)$); there is reasonable agreement, with some over- and some underestimates. Second, to understand the impact of parallelism, we compare the measured per-core performance in the last column to the measure serial performance; performance drops, sometimes significantly, because of resource conflicts. In particular, it drops almost 4x on Ivy, because 1 core is enough to saturate the memory bandwidth for this matrix.

5.2.3 Run-time tuning cost

The cost of run-time tuning depends on whether we use our history database or not: If we do not use it, we need to apply the heuristic performance model to choose the optimal data structure and implementation. If we do use it, we incur the (lesser) cost of accessing the database. Either way, we pay the cost of copying the input matrix from CSR format to the optimal format.

Figure 11 shows the run-time tuning costs in three cases, measured as a multiple of the time to perform a single sequential SpMV on the matrix in CSR format (i.e. without optimizations): (1) Case-I: autotuning with the heuristic performance model in serial, (2) Case-II: autotuning with the heuristic performance model in parallel, and (3) Case-III: autotuning with the history database in parallel. In Cases II and III, the number of cores is the optimal number as shown in Figure 10.

The dense matrix in sparse format (Matrix 1) is special case we discuss separately. We also consider the smaller matrices (Matrices 2-4) separately from the larger ones (Matrices 5-10), because the short running time for the small matrices makes the relative cost of tuning much more expensive.



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform

(d) Santarosa platform (e) Taurus platform

Figure 11: The run-time autotuning cost, relative to unoptimized SpMV in CSR format. Case-I denotes autotuning with the heuristic model in serial. Case-II denotes autotuning with the heuristic model in parallel. Case-III denotes autotuning with history data in parallel. Note that the vertical axises are log-scale.

We see that for a dense matrix (Matrix 1) and for all platforms, the average run-time tuning cost is 113, 143 and 160 unoptimized SpMVs in Case I, II and III, respectively. Case-II costs 1.7x, 1.0x and 1.3x less than Case-I on Gainestown, Santarosa and Taurus, however, it costs 1.3x and 1.3x more than Case-I on Jaketown and Ivy. Case-III costs 5.2x, 1.1x, 1.1x and 1.4x less than Case-I on Gainestown, Jaketown, Ivy and Santarosa, respectively, however, it costs 1.8x more than Case-I on Taurus.

We see that for small matrices (Matrices 2-4) and for all platforms, the average run-time tuning cost is 179, 201 and 300 unoptimized SpMVs in Case I, II and III, respectively. Case-II costs up to 1.1x, 1.2x, 1.2x, 1.8x and 1.8x less than Case-I on Gainestown, Jaketown, Ivy, Santarosa and Taurus, respectively. However, for a few matrices, Case-II costs at most 1.3x, 1.4x and 1.1x more than Case-I on Gainestown, Jaketown and Ivy. Case-III costs up to 1.2x and 2.4x less than Case-I on Jaketown and Ivy. However, for a few matrices, it costs at most 1.4x, 1.4x, 1.8x and 1.8x more than Case-I on Gainestown, Jaketown, Santarosa and Taurus, respectively.

We see that for large matrices (Matrices 5-10) and for all platforms, the average run-time tuning cost is 39, 26 and 14 unoptimized SpMVs in Cases I, II and III, respectively. Case-II costs up to 1.9x, 2x, 1.8x, 2.9x and 5.0x (average 1.5x, 1.6x, 1.5x, 2.2x and 2.7x) less than Case-I on Gainestown, Jaketown, Ivy, Santarosa and Taurus, respectively. However, for a few matrices, it costs at most 1.2x and 1.1x more than Case-I on Gainestown and Jaketown. Case-III costs up to 8.2x, 7.4x, 9.5x, 8.2x, and 10.5x (average 4.5x, 4.2x, 4.9x, 3.9x and 4.3x) less than Case-I on Gainestown, Jaketown, Ivy, Santarosa and Taurus, respectively.

We observe that parallel tuning (Case-II) and using history data (Case-III) give larger speedups (less run-time tuning cost) than Case-I on the largest sparse matrices (Matrices 8-10). However, we also observe that parallel tuning (Case-II) or using history data (Case-III) can result in more cost than Case-I on some cases (specially on small matrices). Further reducing run-time tuning costs is also future work.

5.2.4 Summary of run-time autotuning

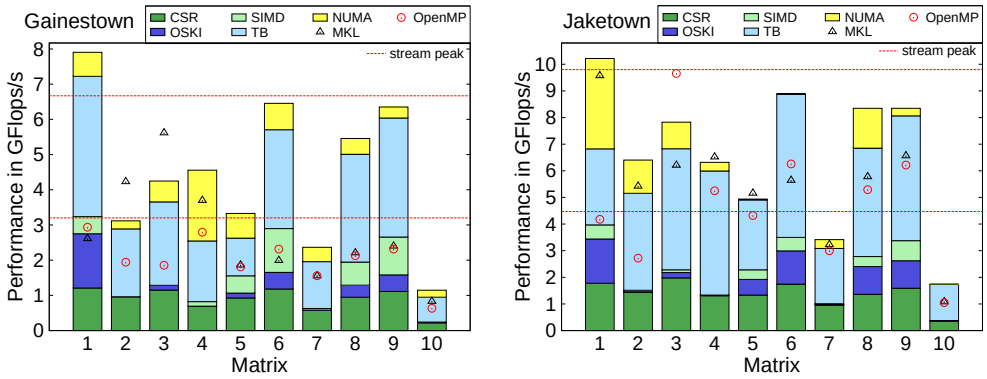
For clarity, we present the overall performance of each autotuning optimization condensed into a stacked bar format as seen in Figure 12, in order to see the contribution made by each optimization. We also include the performance of parallel Intel MKL Sparse BLAS Level 2 routine *mkl_dcsrmv*() and straightforward parallel implementation with OpenMP and CSR for comparison.

The colored bars have the following meanings. The bottommost, dark green, bar (labeled CSR) indicates the performance of the naive implementation, i.e. serial CSR. The second, dark blue, bar (labeled OSKI) indicates the performance of OSKI, i.e. serial performance using OSKI's optimization (register blocking). The third, light green, bar (labeled SIMD) indicates the performance of optimized-BCSR, i.e. serial performance using pOSKI's in-core optimizations (register blocking, software prefetching and SIMD intrinsics). The fourth, light blue, bar (labeled TB) shows the performance of thread blocking, running in parallel with the optimal number of cores shown in Figure 10, but without NUMA-aware mapping (pinning threads to cores by OS scheduler). Finally, the topmost, yellow-green, bar (labeled NUMA) indicates the performance of pOSKI with all of pOSKI's optimizations including NUMA-aware mapping, again running with the optimal number of cores. In addition, we compare our autotuning SpMV performance to the performance bounds from the Roofline model as in Figure 10 (dashed red lines).

In Figure 12, some matrices (Matrices 2, 4, 7, 10) do not or slightly improve performance by optimized-BCSR (labeled SIMD), since the matrices are too sparse. Although other matrices (Matrices 3, 5, 6, 8, 9) do get benefits from optimized-BCSR, the gains are less than in the dense matrix (Matrix 1) mainly due to overheads from filling in explicit zeros within each register block and irregular memory accesses of the input vector between register blocks. This further suggests that reordering methods for creating larger dense blocks will be worthwhile for optimized-BCSR for other sparse matrices.

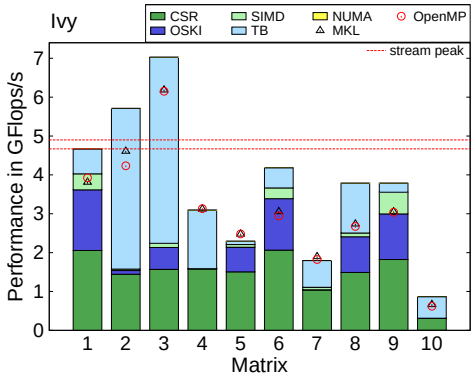
All matrices (Matrices 1 to 10) show performance improvement by thread blocking (labeled TB) on all platforms. Ivy shows the least performance improvement (4% over optimized-BCSR) by thread blocking on Largebasis (Matrix 5). NUMA-aware mapping (labeled NUMA) improves some matrices on NUMA architectures, however it also depends on the matrix and the underlying architecture.

As seen in Figure 12, the overall our autotuning for SpMV for the dense matrix (Matrix 1) in sparse

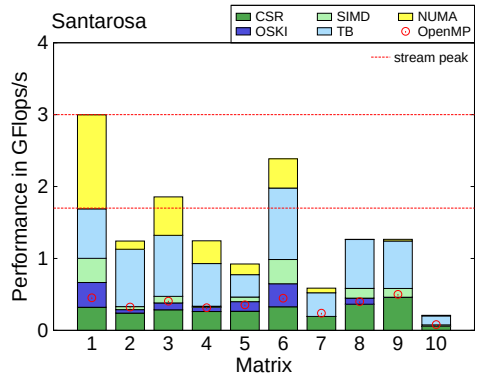


(a) Gainestown

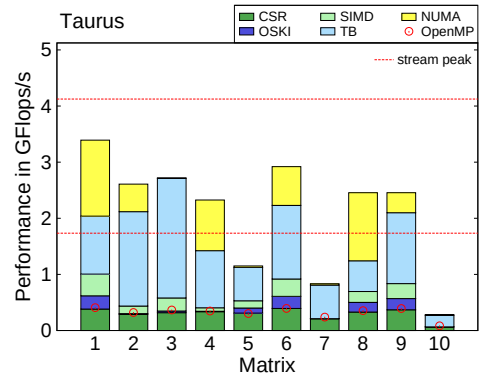
(b) Jaketown



(c) Ivy



(d) Santarosa



(e) Taurus

		pOSKI speedup over			
		CSR	OSKI	OpenMP	MKL
Dense Matrix 1	min	2.3	1.3	1.2	1.1
	ave	6.6	3.4	4.3	1.8
	max	9.3	5.5	8.3	3
Small Matrices 2-4	min	2	2	0.8	0.7
	ave	5.2	4.8	3.2	1
	max	9	8.6	8.1	1.3
Large Matrices 5-10	min	1.5	1.1	0.9	0.9
	ave	4.3	3.3	2.7	1.6
	max	7.5	5.5	7.4	3.2
All	min	1.5	1.1	0.8	0.7
	ave	4.8	3.7	3.0	1.4
	max	9.3	8.6	8.3	3.2

(f) Speedups on all platforms

Figure 12: Overall performance of autotuned SpMV. Each colored bar corresponds to performance from a different set of optimizations, as described in the text. Intel MKL results on Intel platforms are denoted by black triangles. OpenMP results are denoted by red circles. Two Roofline-based performance bounds are shown, one using bandwidth measured using the Stream benchmark with a large data set (lower red dashed line) and one with a small data set (upper red dashed line). Minimum, average and maximum speedups for each subset of matrices are shown in Figure 12(f). Note that speedup less than 1 means slowdown factor.

format achieves up to 74%, 80%, 73%, 57% and 64% of the system peak performance of SpMV (the third-from-rightmost column of the table in Figure 5(f)), and up to 118%, 104%, 95%, 100% and 83% of the

Roofline-based performance bound using bandwidth measured with the Stream benchmark with a small data set (upper red line) on Gainestown, Jaketown, Ivy, Santarosa and Taurus platform, respectively. Since the dense matrix (Matrix 1) in sparse format contains clear dense structure and large dimensions, the overall performance on the dense matrix is a kind of empirical upper bound on large matrices (Matrix 5-10).

Among large sparse matrices (Matrices 5-10) in our test suite, Tsopf (Matrix 6) shows the highest performance on all platforms because it has the highest density, the highest number of nonzeros per rows and the smallest dimensions (see Figure 6) among Matrices 5-10. Tsopf achieves at least 80% of the performance with the dense matrix. Wiki-2007 (Matrix 10) shows the lowest performance on all platforms because the matrix is very sparse with the largest dimensions resulting in poor cache locality of input vector. Wiki-2007 shows less than 20% of the performance with the dense matrix. Cache and TLB blocking optimizations may improve data reuse for large dimensional matrices.

As shown in Figure 12(f), for the dense matrix (Matrix 1) and for all platforms, pOSKI is up to 9.3x faster than the reference serial CSR implementation, up to 5.5x faster than the optimized serial OSKI implementation, up to 8.3x faster than a straightforward parallel implementation using OpenMP and CSR, and up to 3x faster than the parallel Intel MKL implementation.

For the small matrices (Matrices 2-4) and for all platforms, pOSKI is up to 9x faster than the reference serial CSR implementation, up to 8.6x faster than the optimized serial OSKI implementation, up to 8.1x faster than a straightforward parallel implementation using OpenMP and CSR, and up to 1.3x faster than the parallel Intel MKL implementation. However, for a few matrices, pOSKI performs at most 1.3x and 1.2x slower than Intel MKL and OpenMP implementations, respectively.

For the large matrices (Matrices 5-10) and for all platforms, pOSKI is up to 7.5x faster than the reference serial CSR implementation, up to 5.5x faster than the optimized serial OSKI implementation, up to 7.4x faster than a straightforward parallel implementation using OpenMP and CSR, and up to 3.2x faster than the parallel Intel MKL implementation. However, for a few matrices, pOSKI performs at most 1.1x and 1.1x slower than Intel MKL and OpenMP implementations, respectively.

In summary for each platform, pOSKI achieves up to 6.6x, 6.1x, 4.5x, 9.3x and 9x speedups over naive CSR, up to 6.6x, 4.9x, 3.7x, 4.9x and 8.6x speedups over OSKI, and up to 2.8x, 2.4x, 1.4x, 6.6x and 8.3x speedups over OpenMP+CSR on Gainestown, Jaketown, Ivy, Santarosa and Taurus, respectively. It achieves up to 3.2x, 1.6x and 1.4x speedups over MKL on Gainestown, Jaketown and Ivy platform, respectively. Additionally, pOSKI shows better performance than MKL or OpenMP implementations on most cases (42 out of 50 cases), while MKL or OpenMP implementations show better performance than on a few cases (Matrix 2 and 3 on Gainestown, Matrix 3-5 on Jaketown, Matrix 4, 5 and 7 on Ivy platform).

6 Conclusions

We have presented pOKSI, an autotuning framework for Sparse Matrix-Vector Multiplication (SpMV) on a diverse set of multicore platforms. We demonstrate significant performance gains over prior implementations: pOSKI is up to 9.3x faster than the reference serial CSR implementation, up to 8.6x faster than the optimized serial OSKI implementation, up to 8.3x faster than a straightforward parallel implementation using OpenMP and CSR, and up to 3.2x faster than the parallel Intel MKL implementation. However, for a few matrices, pOSKI performs at most 1.3x and 1.2x slower than Intel MKL and OpenMP implementations, respectively. We also show the how the performance of SpMV and run-time tuning cost strongly depends on structural properties of the sparse matrix and features of the underlying architecture.

Our autotuning depends on the following components: (1) Off-line optimization of in-core architectural features can thoroughly explore a large design space of possible implementations, without impacting run-time costs. (2) Run-time optimization of NUMA-aware thread-level parallelism can be done very quickly, and includes the possibility of using fewer than the maximum number of cores. (3) Run-time optimization can be done even more quickly by using a history database to reuse prior autotuning results. Using the history database depends on user hints saying "this matrix has the same structure as matrix XYZ in the database"; see [6] for more details about the pOSKI user interface.

Future work includes extending the scope of both off-line and run-time optimizations. These include (1) investigating reordering methods to reduce the fill ratio in BCSR format, (2) exploiting index compression, cache-blocking and TLB blocking to reduce memory traffic or to improve locality, (3) implementing other matrix partitioning schemes, (4) improving load balance when there are different data structures for each submatrix, (5) reducing data structure conversion costs at run-time, and (6) determining the most efficient number of cores for parallelism. We also plan to expand our autotuning framework to other, higher level sparse kernels, such as A^TAx and the matrix powers kernel $[Ax, A^2x, ..., A^kx]$.

Acknowledgements

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C.Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Also supported by U.S. DOE grants DE-SC0003959, DE-SC0004938, DE-SC0005136, DE-SC0003959, DE-AC02-05-CH11231, DE-FC02-06ER25753, DE-FC02-07ER25799, and DE-FC03-01ER25509.

References

[1] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886 – 905, 1996.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[4] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. *Proceedings of the International Conference on Supercomputing*, July 1997.

[5] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[6] J.-H. Byun, R. Lin, J. W. Demmel, and K. A. Yelick. poski: Parallel optimized sparse kernel interface library user's guide for version 1.0.0, 2012. http://bebop.cs.berkeley.edu/poski.

[7] U. Catalyurek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '96, pages 75–86, London, UK, UK, 1996. Springer-Verlag.

[8] U. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, Feb. 2010.

[9] E. Cuthill and J. Mckee. Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the 24th National Conference of the Association for Computing Machinery, ACM Publication P-69*, 1969.

[10] T. A. Davis. University of florida sparse matrix collection. *NA Digest, 92*, 1994. http://www.cise.ufl.edu/research/sparse/matrices/.

[11] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *Proceedings of the 5th international conference on Computational Science - Volume Part I*, ICCS'05, pages 99–106, Berlin, Heidelberg, 2005. Springer-Verlag.

[12] O. A. Fagerlund. Multi-core programming with opencl: performance and portability : Opencl in a memory bound scenario, 2010.

[13] M. Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, May 1999.

[14] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse cholesky factorization. *SIAM J. Matrix Analysis and Applications*, 15:1075–1091, 1994.

[15] P. Guo and L. Wang. Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus. In *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pages 1154 –1157, dec. 2010.

[16] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, UC Berkeley, Jun 2000.

[17] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136. Springer, may 2001.

[18] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135 – 158, 2004.

[19] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, and E. W. Bethel. A generalized framework for auto-tuning stencil computations. In *In Proceedings of the Cray User Group Conference, 2009*.

[20] V. Karakasis, G. Goumas, and N. Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 1, pages 247 –256, aug. 2009.

[21] I. P. King. An automatic reordering scheme for simultaneous equations derived from network systems. *Int. J. num. Meth. Engng*, 2:523–533, 1970.

[22] Lapack: Linear algebra package. http://www.netlib.org/lapack/.

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

[24] J. Pichel, D. Singh, and J. Carretero. Reordering algorithms for increasing locality on multicore processors. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, pages 123 –130, sept. 2008.

[25] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing, ACM/IEEE 1999 Conference*, page 30, nov. 1999.

[26] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[27] R. Shahnaz, A. Usman, and I. Chughtai. Review of storage techniques for sparse matrices. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1 –7, dec. 2005.

[28] P. Stathis, S. Vassiliadis, and S. Cotofana. A hierarchical sparse matrix storage format for vector processors. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., april 2003.

[29] Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream.

[30] F. Tavakoli. Parallel sparse matrix-vector multiplication. Master's thesis, Royal Institute of Technology (KTH), 1997.

[31] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, UC Berkeley, 2003.

[32] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[33] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(12):3 – 35, 2001.

[34] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, UC Berkeley, 2008.

[35] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[36] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[37] M. M. Wolf, E. G. Boman, and B. A. Hendrickson. Optimizing parallel sparse matrix-vector multiplication by corner partitioning. In *Proceedings of PARA08*, Trondheim, Norway, may 2008.