Structure-based Optimizations for Sparse Matrix-Vector Multiply

Mehmet Belgin

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Godmar Back, co-chair
Calvin J. Ribbens, co-chair
Kirk W. Cameron
Serkan Gugercin
Adrian Sandu

December 14, 2010
Blacksburg, Virginia

Keywords: Sparse, Matrix Vector Multiply, SpMV, SMVM, Code Generators, Vectorization, parallel SpMV, thread pool, OSF, PBR

Structure-based Optimizations for Sparse Matrix-Vector Multiply

Mehmet Belgin

(ABSTRACT)

This dissertation introduces two novel techniques, OSF and PBR, to improve the performance of Sparse Matrix-vector Multiply (SMVM) kernels, which dominate the runtime of iterative solvers for systems of linear equations. SMVM computations that use sparse formats typically achieve only a small fraction of peak CPU speeds because they are memory bound due to their low flops:byte ratio, they access memory irregularly, and exhibit poor ILP due to inefficient pipelining. We particularly focus on improving the flops:byte ratio, which is the main limiter on performance, by exploiting recurring structures or sub-structures in matrices. Our techniques also support micro-architecture level optimizations to further improve performance.

Operation Stacking Framework (OSF) stacks problems in large ensemble computations, which run the same sparse kernel using an identical matrix structure, such that they share a single copy of the indexing information to significantly reduce memory bandwidth usage. OSF provides performance improvements of up to 1.94× on an AMD Opteron compared to the CSR method. We validate performance results using hardware event counters, which demonstrate significantly improved cache and pipeline utilization.

Pattern-based Representation (PBR) exploits recurring block nonzero patterns by generating custom code for each recurring block pattern. In this way, no indexing data for individual nonzero elements are read from memory, reducing the overall size of the indices by up to 98%. Our code generator emits highly tuned codes that utilize SSE vectorization and software prefetching. PBR accurately identifies a block size that achieves optimal or near-optimal performance using a linear multiple regression performance model. On recent multicore machines, PBR provides performance improvements of up to 3.4× sequentially and 5× in parallel, compared to the CSR method. The PBR library we provide converts matrices at runtime, allowing our method to be used as a drop-in replacement for existing methods. We compare PBR's overhead relative to its benefits and show that PBR is beneficial for many applications that repetitively call the SMVM kernel for the same matrix structure.

Dedication

To *Müzeyyen Ergün...*

A loving grandma and exceptional teacher

who taught me, and many others, the art of learning

through her infinite love and patience.

Acknowledgments

I have had the privilege of having Dr. Calvin Ribbens and Dr. Godmar Back as my co-advisors. Dr. Ribbens had more confidence in me than I had for myself from the very beginning. Without his exceptional mentorship, both in scientific and personal matters, I would not be able to complete this long and difficult journey. Dr. Back introduced me to a whole new level of perfection, which I did not know even existed. He not only challenged me to always reach higher, but also provided me with everything I needed, down to the very last detail. I am truly grateful and proud to be their student. I would also like to thank my committee, Dr. Cameron, Dr. Sandu and Dr. Gugercin for their constructive criticism and valuable suggestions. I would like to thank Dr. Kevin Shinpaugh for providing me with continuous funding during my Ph.D. study. Thank you Campbell family, Petie, Bharath, Rajesh, Hari, and all other special friends, for many unforgettable Blacksburg memories.

Last but not least, I am grateful to my family for their exceptional love, support and patience. Perhaps the most difficult part of obtaining this Ph.D. was being away from them. Thank you Ebru for being my best friend, for your unconditional love, and for keeping your cool while helping me gather up my "lost goats" when I thought they were gone forever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation presents two structure-based techniques to improve the performance of sparse matrix-vector multiply (SMVM), an important computational kernel that multiplies a sparse matrix A by a dense vector x [BBC+94, Saa96]. SMVM kernels are heavily used in iterative solvers that solve sparse systems of linear equations, which arise in a wide range of scientific and engineering applications, including simulations, machine learning, and graphics [ABC+06]. A 2006 report titled "The landscape of parallel computing research: A view from Berkeley" lists SMVM as one of the 13 most common algorithms, or *dwarfs*, which capture the computation/communication patterns of today's applications [ABC+06].

Sparse matrices are matrices that are dominated by zero elements, which pose no numerical significance for matrix multiply. To avoid storage of, and computation on these zeros, sparse matrices are usually represented using specialized storage formats, which greatly reduce memory and CPU usage. Despite these benefits, however, sparse algorithms use computa-

tional resources very inefficiently, typically achieving only 10% or less of peak CPU speeds for untuned implementations [Vud03].

The performance of SMVM is primarily limited by the sustainable memory bandwidth of architectures [GKKS99, WOV+07]. Sparse matrix formats keep indexing information along with nonzero values, which increases the amount of data read from memory per matrix nonzero. Unfortunately, each nonzero element is used in only a single multiply-add floating point operation by the SMVM algorithm. This lack of data reuse significantly reduces performance because sustainable memory speeds on modern architectures are orders of magnitude slower than floating point speeds of CPUs. In addition, current architectural design trends increase the number of cores without proportionally increasing their memory bandwidth, which causes even greater contention for memory bandwidth, suggesting that SMVM will be even more limited by memory speeds in the future.

Williams *et al.* quantify the disparity between the CPU and memory with a *flop:byte* ratio [WOV+07, WOV+09]. SMVM algorithms perform two floating point operations (multiply and add) for each 8-byte double precision number read from the memory. Therefore, the upper bound for this ratio is 2:8 (0.25) if no indexing information were needed. The techniques we present in this dissertation reduce the size of the accessed indexing information to approach this theoretical bound.

A slow SMVM kernel with low flop:byte ratio directly and adversely affects the overall performance of applications that include iterative solvers of any type, because iterative solvers spend a majority of their runtime in the SMVM kernel. Figure 1.1 illustrates the percentage of time spent in SMVM for two widely used sparse iterative solvers, namely Conjugate Gradi-

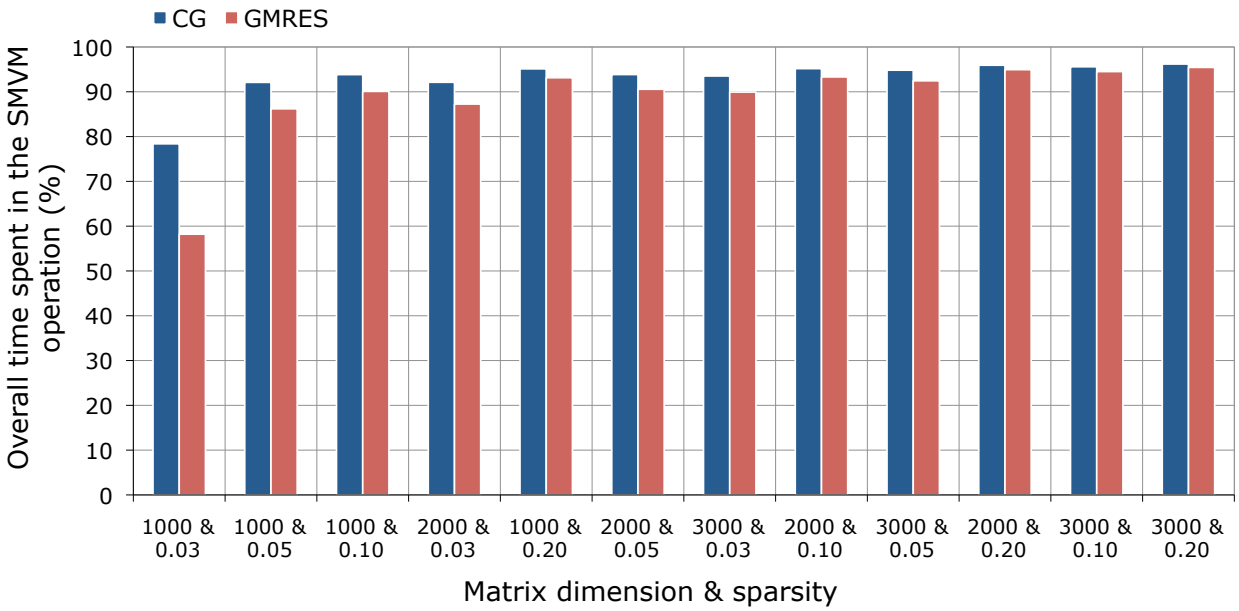


Figure 1.1: **Runtime spent in SMVM operations for two common iterative solvers.** Percent of time spent in SMVM for 30 iterations of CG and GMRES solvers on random matrices of varying dimension and sparsity (percent of nonzero elements). Matrices are represented using CSR format.

ent (CG) [GV89, BBC+94] and Generalized Minimal Residual (GMRES) [GV89, BBC+94]. For large problems, these methods spend 90% or more of their runtime in the SMVM operation, making it the primary performance bottleneck.

The two techniques we present in this dissertation augment the arsenal of SMVM methods by providing significant performance improvements (by up to 3.4× sequentially and 5× in parallel) over the fastest alternatives available in the literature. Both methods focus on the primary performance bottleneck, the memory boundedness, by identifying and reducing redundancies in sparse matrix representations. In addition, our methods take advantage of micro-level optimizations, such as vectorization, loop unrolling, and software prefetching. Our main motivation comes from our observation that an important subset of matrices share

identical structures or sub-structures, which can be exploited to reduce indexing overhead, thus leaving more available memory bandwidth for streaming nonzero values.

We consider two cases in which redundant matrix structure occurs. The first is when matrices from multiple problems share an identical structure. To improve performance, we solve these problems simultaneously, such that all problems share only one copy of indexing information. Since we stack several problems to reduce bandwidth usage, we refer to this approach as *operation stacking*.

The second case is when a single matrix can be decomposed into blocks that include recurring nonzero patterns. We generate custom SMVM kernel code that corresponds to each recurring block pattern, so that no index information is needed for individual nonzero elements within each block. We refer to this method as *pattern-based representation (PBR)*.

In the remainder of this chapter, we introduce these two novel techniques and provide an overview of the rest of the dissertation.

1.1 Operation Stacking

Operation stacking is motivated by *ensemble computations*, in which multiple problems need to be solved and the nonzero matrix structure of each problem is identical. The conventional approach is to run these problems separately, reading the same set of indexing information from memory repeatedly. We avoid this overhead by solving multiple problems in a simultaneous, collective operation. The benefit of this approach is twofold. First, only one copy

of the indexing information for the identical matrix structure is read from memory, which is then reused by all participating problems, thereby greatly reducing the memory bandwidth usage and improving the flops:byte ratio. Second, by solving multiple problems simultaneously, we permit some micro-level optimizations that would otherwise be ineffective. For example, operation stacking allows interleaved placement of data of multiple problems in memory, which improves spatial locality. Similarly, since the data from multiple problems are stacked, the inner loop of the SMVM algorithm contains more independent instructions. This reduces the relative cost of loop overhead and allows for more efficient pipelining by the CPU, thereby improving instruction level parallelism (ILP). We exploit this situation by expressing the stacked operations in unrolled form rather than using a loop. The resulting optimized code is specific to a fixed number of stacked problems, which we refer to as *stack depth*.

When problems converge, the stack depth changes at runtime. Therefore, stacked operation requires multiple versions of the code, each with an unrolled loop for a different stack depth. To shield users from the burden of manually writing multiple versions of stacked code, we developed the *Operation Stacking Framework (OSF)*. The OSF provides end users with *stacked solvers,* which are ready-to-use stacked versions of widely used iterative solvers, such as CG and GMRES. These stacked solvers can dynamically handle varying stack depths, since they already include a replicated version for each possible stack depth, and provide an effective mechanism for ejecting converged problems from the stack. The integration of these stacked solvers into existing codes requires minimal effort, because their interfaces are similar to other common numerical libraries. OSF keeps each problem in its own separate process without any changes to their data or computation patterns. For the caller, a stacked

function is no different than a regular function, as OSF does not intervene with the program flow until a call into a stacked solver. After a stacked solver is called, OSF handles all tasks related to data re-arrangement, process synchronization, and convergence, transparent from the user.

To avoid restricting the benefits of operation stacking to only a limited set of solvers, we developed the *OSF Developer Library,* which facilitates the conversion of any given iterative solver into stacked form, by supporting the replication of code for each stack depth. OSF stacks solvers using an automated code generation and replication process at compile time, without exposing stack solver developers to low-level programming details. Developers provide only data and function templates, which are similar to non-stacked codes with additional annotations (macros) indicating sections that need replication. The OSF's preprocessor executes these macros to generate the final stacked solver code, requiring no further involvement from the developers. Hence, any iterative solver can easily be adapted to perform operation stacking with little or no in-depth knowledge of the operation stacking technique or system-level programming.

We evaluate OSF on a set of matrices and demonstrate that operation stacking can improve the performance of widely used sparse solvers, such as CG and GMRES, by up to 1.94× on an AMD Opteron processor. In addition, we show that the OSF implementation costs are amortized after as few as five solver iterations. To explain and validate the performance improvements provided by OSF we collect hardware event counter data using the PAPI [BDH+00] interface and demonstrate significant improvements in cache hit rates and instruction level parallelism (ILP), when compared to non-stacked algorithms. We explore multiple conversion scenarios and show that OSF continues to provide performance benefits

even for the worst case scenario in which the ensemble includes only a few problems during the majority of its runtime.

1.2 Pattern-based Representation

Our second technique, *Pattern-based Representation (PBR)*, is motivated by the observation that many sparse matrices include blocks that share identical nonzero patterns. Exploiting this redundancy, PBR reduces the indexing information that needs to be read from memory, thereby allowing a higher flop:byte ratio, which leads to improved SMVM performance. PBR performs a simple analysis of the matrix to identify blocks with recurring patterns, then generates custom codes that execute instructions specific to each recurring pattern, so there is no need to read any indexing information for nonzeros inside each block from memory. As an added benefit, the generated codes can efficiently implement several micro-level optimizations, namely explicit prefetching, vectorization, and unrolling, since the matrix analysis makes each block's nonzero structure known to PBR's code generator.

Once a generated code's instructions have been fetched from memory, they are repeatedly applied to nonzeros in all of the blocks that share the same pattern, thereby creating temporal locality via the instruction cache. To achieve this code reuse, PBR uses *matrix splitting*, placing all nonzeros that belong to the same pattern in a submatrix. Since these nonzeros and their block indices are stored contiguously and accessed by the generated code in order, PBR guarantees good spatial locality in accessing these arrays.

We define two criteria to exclude block patterns that yield no benefit from PBR. Nonzeros

that belong to blocks of excluded patterns are collected in a remainder submatrix, which is kept using the CSR format. Therefore, for PBR to be effective, a high percentage of nonzeros in matrices must be covered by the PBR portion of the representation. We explored the nonzero coverage by PBR for a large set of matrices that originated from a wide variety of existing applications. We found that matrices from problems with underlying physical geometry, such as structural problems, computational fluid dynamics, graphics problems, etc., are more likely to benefit from PBR because these problems yield structured matrices. On the other hand, problems with no underlying physical geometry, such as weighted graphs, circuit simulations, and power network problems, involve less structured matrices, which leads to less coverage in some cases.

To make PBR usable, we developed the *PBR library*, which automates several steps in using this sparse matrix representation. First, PBR subdivides the matrix into square blocks of a given block size. Since each choice of block size yields a different set of recurring patterns and nonzero coverage, the matrix needs to be analyzed for different candidate block sizes. The PBR library achieves this task by using an highly optimized matrix analyzer, which has an asymptotic time complexity linear in the number of nonzeros.

Second, we provide a linear multi-regression model to select a block size that is predicted to yield the best performance. This model takes different memory access characteristics for different block patterns as input and estimates the performance for a given block size without actually running any SMVM operations. We validated the accuracy of this model and showed that performance loss due to mispredicted block sizes is only 3% on average.

Third, PBR requires data re-arrangement to assign nonzero elements to their respective sub-

matrices. The PBR library can complete this task at a low cost with linear time complexity because the initial matrix analysis step records the final destinations of nonzero values and block indices, including the locations of remainder nonzeros in the CSR matrix. In this way, the final PBR representations can be constructed without any sorting. The final PBR representations are saved to disk for future use to avoid repeated analysis of the same matrix.

Finally, the PBR library generates and compiles tuned custom codes for all recurring patterns. Compiled codes are kept on disk as shared object files, which are dynamically linked into the calling process at runtime. The PBR code generator can efficiently vectorize codes using SSE intrinsics for architectures that support this optimization.

Since the PBR library performs all necessary steps to convert matrices, it allows us to evaluate the cost of PBR relative to its benefits. Our evaluation shows that the PBR costs due to matrix analysis and data rearrangement can be compensated for after a few hundred SMVM operations on average. Compilation of generated codes significantly adds to the overhead, increasing the average break even point to thousands of SMVM operations. However, the PBR library can largely eliminate the need for code compilation by maintaining a code cache. We show that this cache achieves high hit rates on a large set of problems from varying problem domains.

PBR is also amenable to parallelization, since matrix nonzeros can be evenly distributed to threads at a block-level granularity. We row-partition matrices, which is a common strategy also used in parallel CSR. We included an additional step in the matrix analyzer to assign blocks to threads for equally distributing the workload, i.e., the nonzeros. A custom thread pool we developed allows for efficient parallel execution of PBR-generated codes.

Our evaluation on two recent AMD and Intel architectures demonstrates sequential performance improvements by up to 3.4×, averaging at 1.4× on AMD and 1.25× on Intel, in comparison to the CSR method. We show that PBR can also improve performance when compared to the widely used OSKI [VDY05] library. In parallel, PBR scales equally well as CSR and improves the SMVM performance by up to 5× when compared to this method.

1.3 Contributions and Scope

Operation stacking and pattern-based representation techniques contribute to the area of computational linear algebra by significantly improving the performance of sparse matrix vector (SMVM) kernels. We developed two comprehensive frameworks, one for operation stacking and another for PBR, to make these methods not only efficient, but also usable. In this section, we summarize these contributions.

Operation Stacking

- We improve SMVM performance on ensemble computations by eliminating unnecessary use of multiple copies of identical indexing data, and also by allowing efficient micro-level optimizations.
- We present the Operation Stacking Framework (OSF), which facilitates operation stacking by providing:
  - Ready-to-use stacked implementations of CG and GMRES solvers.

  - A comprehensive API of macros and functions to convert any given solver to stacked form with minimal effort.
  - An algorithm independent *OSF Iteration Engine (OSFIE)* to manage stacked algorithm iterations and efficient ejection of converged problems from the stack.
  - Compile-time code generation/replication scripts that emit stacked codes for multiple number of stack depths.

- We evaluate and validate the performance improvements by OSF on existing matrices from real application areas, as well as on randomly created matrices, by using hardware event counters that provide evidence for better cache and FPU utilization. We also explore the impact of varying converge scenarios and show that OSF retains its performance advantage even in the most pessimistic scenarios.

Pattern-based Representation

- We improve the SMVM performance on matrices that include blocks with recurring patterns by generating custom codes, which reduce the memory bandwidth usage and allow efficient micro-level optimizations.
- We present a comprehensive library to efficiently implement and combine all necessary steps required to benefit from PBR. This library includes the following components:
  - A low-cost matrix analyzer to identify recurring patterns for all candidate block sizes in only one pass.
  - A linear multi-regression model to predict performance accurately for the selection of an optimal block size.

  - A CSR to PBR matrix converter that efficiently implements required data rearrangement and matrix splitting for a given (optimal) block size.
  - A code generator that emits optimized custom codes that utilize automatic vectorization, explicit prefetching, and efficient parallelism.
  - A code cache that stores pre-compiled binaries to avoid repeated generation and compilation of codes for the same pattern.
  - A custom thread pool for efficient thread-level parallelism on multicore systems.

- We present significant performance improvements by PBR, both sequentially and in parallel, on a selected set of matrices that come from varying application areas. We also analyze PBR's analysis overhead relative to its benefits to show how many SMVM operations are required to amortize the implementation costs. We explore the PBR coverage of matrices from a wide variety of problem domains and show that problems that involve an underlying 2D/3D physical structure are more likely to benefit from the PBR. We demonstrate that a code cache that stores precompiled codes can achieve high hit rates, hence largely eliminating the need for repeated code generation/compilation.

1.4 Organization

The remainder of the dissertation is organized as follows. Chapter 2 provides readers with general background information. Common sparse storage representations (COO and CSR) and their usage in a SMVM operation are presented to highlight the performance problems we address, which are categorized and explained in detail. Chapter 3 is devoted to the oper-

ation stacking framework and presents the idea, benefits, implementation, and evaluation of this method in detail. This chapter explains the end-user and developer components of the OSF library, its support for variable convergence, and presents performance improvements by OSF, along with their validation using hardware event counters. Chapter 4 is devoted to the pattern-based representation method. This chapter explains the fundamentals of PBR, demonstrates how recurring patterns are identified and represented to improve performance, and finally presents an extensive evaluation of PBR performance. Chapter 5 summarizes related work in the literature and explains its relationship to OSF and PBR. Finally, Chapter 7 provides a conclusion of the dissertation and provides short and long term future research directions.

Chapter 2

Background

Reaching a high fraction of peak processor speeds on a range of scientific applications has always been challenging, hence the marketing term "theoretical peak speed." The performance gap between achieved and peak performance has largely been closed for dense matrix computations with the introduction of block algorithms [DDSvdV98] and optimized cache-aware numerical libraries [ABB+90, DCHD90, GvdG02, WPD01]. These libraries easily achieve more than 80% of peak speeds when solving dense problems.

SMVM computations, on the other hand, are historically known to be slow; they typically perform at 10% of peak (theoretical) CPU speeds for naïve implementations [Vud03]. Although a number of numerical libraries offer direct or iterative solvers for sparse matrix computations [BB+08, RP96, Saa94, DLPR94, JKK+99, THHS99, VDY05], none attains performance comparable to dense operations. For instance, the *PETSc-FUN3D* code [AGK+99], which was awarded the Gordon Bell Prize in 1999, achieved only $\sim 25\%$ of the peak CPU

speed [DDD+06].

High memory bandwidth usage, poor data locality, and poor instruction level parallelism are three main reasons that cause the poor performance of the SMVM kernel. This section introduces sparse storage formats and then provides a detailed investigation of these three limiters of SMVM performance, each of which is addressed by the techniques we describe in this dissertation.

2.1 Representation of Sparse Matrices

If a matrix includes a sufficient number of zeros to be exploited, then it is considered *sparse* [WR71]. It is possible to store a sparse matrix as it were dense, by explicitly storing zeros. However, doing so puts significant pressure on the memory bandwidth (for reading/writing zeros) and the CPU (for floating point operations on zeros). As a solution, sparse matrices are represented using formats that record and store the location of each nonzero in the matrix along with its value, while excluding all zero elements. An example of such a format is *Coordinate format (COO)*, which keeps the row and column indices of each nonzero using two integers (Figure 2.1) [Saa96].

An alternative to COO is *Compressed Sparse Row (CSR)*, which is one of the most commonly used sparse storage formats. It improves COO by grouping nonzeros by row, thus eliminating the need to store and access their identical row index values. This compression of the row index array [Saa96, BBC+94] reduces storage requirements and memory bandwidth usage. CSR assumes that nonzeros are kept in row order, while its variation *Compressed Sparse*

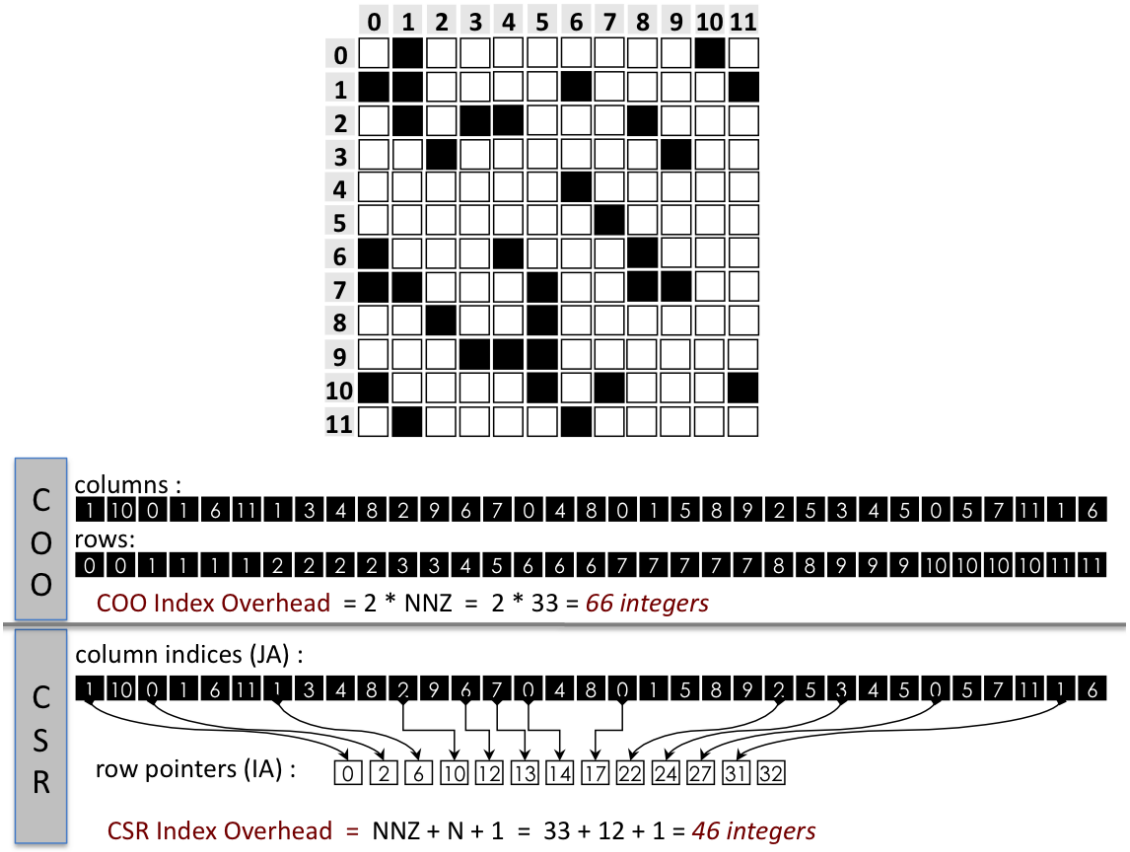


Figure 2.1: Representation of a 12×12 matrix with 33 nonzeros using the COO and CSR formats. COO keeps a pair of integers for each nonzero and uses 66 integers. CSR compresses the rows array, by removing recurring row indices, and reduces the indexing overhead to 46 integers. The array JA of CSR keeps the column index of each nonzero and the array IA points to the first nonzero on each row.

Column (CSC) assumes column-order storage of nonzeros. Figure 2.1 compares the COO format to CSR for a 12×12 matrix with 33 nonzeros. The column index array, JA, keeps column indices of each nonzero. The compressed rows array, IA, points to the first nonzero on each row. In this example, CSR (and similarly CSC, which incurs identical cost) saves 20 integers by eliminating the repeated indices in the row array of COO. With this reduction in indexing overhead, CSR achieves a higher flop:byte ratio by reading less indexing data from memory for the same number of floating point operations.

2.2 Performance Limitations for SMVM Computations

2.2.1 Memory Boundedness

Each sparse matrix format requires a matching SMVM algorithm that implements format-specific access patterns and operations. The SMVM algorithm for CSR representation is shown in Figure 2.2. Matrix nonzero values are kept in the AA array, and the algorithm computes their logical row and column indices at runtime by using JA and IA. The algorithm reads elements of AA, JA, and IA from memory, with $AA(k)$ and $JA(k)$ used only once per each iteration of the inner loop. Because of the lack of re-use of these elements, SMVM performance is limited by sustained memory bandwidth of the architecture. Compounding the problem, such memory-bound operations cannot be improved by micro-level optimizations that target better CPU utilization, because the CPU would already be idle while waiting for the data. For these reasons, reducing memory bandwidth usage, which is the primary goal of two techniques we present in this dissertation, is crucial for improving

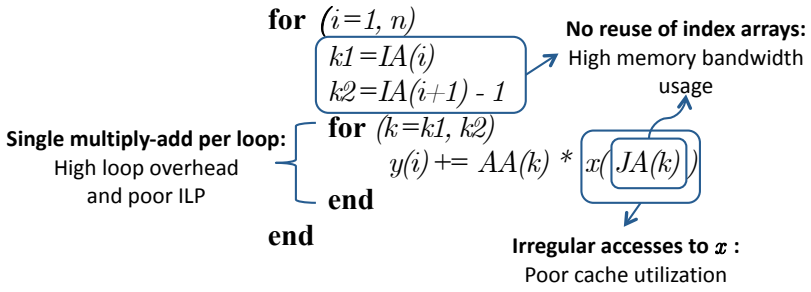


Figure 2.2: CSR SMVM algorithm.

the SMVM performance.

To illustrate the memory boundedness of CSR, we examined how much performance would improve if we could completely remove the floating point operations from the CSR SMVM algorithm. We refer to this synthetic benchmark as '*no-ops CSR*'. To avoid perturbing the sequence of memory accesses, we use inline `asm` statements to trick the compiler into generating code that performs exactly the same memory accesses as SMVM, but omits the actual floating point operations. By replacing the assignment:

```
y(i) += AA[k] * x[JA[k]];
```

in Figure 2.2 with the statement:

```
asm("nop":"=x" (y):"x" (AA[k]), "x" (x[JA[k]]));
```

we instruct the compiler to provide `y`, `AA[k]`, and `x[JA[k]]` in three SSE floating point registers, insert a `nop` instruction in the emitted code, and to act subsequently as if a new value for `y` had been produced in the first register. Figure 2.3 depicts the (theoretical) improvements in performance provided by no-ops CSR, which marks the ceiling for CSR

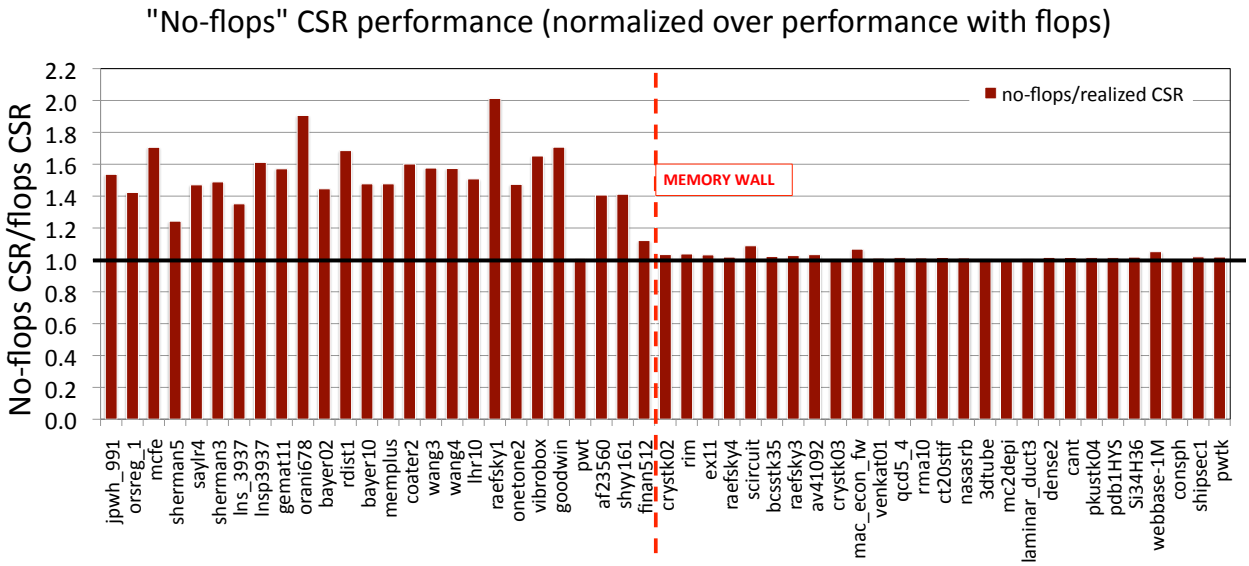


Figure 2.3: **Memory boundedness of CSR.** Comparison of CSR SMVM performance with (baseline) and without floating point operations. Matrices are sorted by size (number of nonzeros), growing from left to right.

performance. Matrices in this figure are sorted by the number of nonzeros they include, from small (left) to large (right). This figure clearly shows the point at which CSR hits the *memory wall* [WM95], after which CSR is completely memory bound and no micro-level optimizations can further increase its performance. After this point is reached (i.e., for matrices in the right half of the figure), the only way to improve performance is by reducing the memory bandwidth usage.

2.2.2 Lack of Data Locality

Data locality can be defined in terms of the distance of between accesses [Den68, Den05]. There are two kinds of data locality: *temporal*, where an application is likely to frequently access the same data location within a short timespan, and *spatial*, where an application

is likely to access data at memory locations in close vicinity of each other. Both kinds of locality are exploited by hierarchical memory designs, which employ several tiers of memory from large but slow (such as hard drives) to small but fast (such as low-level caches and CPU registers). If an application exhibits locality, then most of the data fetched to higher tiers of memory can be used before it is evicted. Transferring data from a low memory level to higher levels incurs a cost. Computer architectures are historically designed around the assumption that applications exhibit temporal and/or spatial locality. An example of such applications is the *dense blocked matrix-matrix multiply* [DCHD90, KLvL98] which achieves near-peak CPU speeds and is often used as a benchmark [DMBS86] to measure sustainable peak performance of architectures.

Unlike dense matrix-matrix multiply, SMVM is a *streaming* operation with poor temporal locality, because SMVM algorithms stream nonzeros and index arrays from memory, but use them only once in floating point operations. SMVM is also an *irregular* application with poor spatial locality, because SMVM algorithms accesses memory irregularly, causing cache misses. For example, when $x(JA[k])$ in the CSR algorithm (Figure 2.2) is accessed, the processor incorrectly assumes that the following accesses will be to the neighboring elements $x(JA[k]+1), x(JA[k]+2), \ldots$. Hence, these elements are fetched into the cache in an attempt to exploit spatial locality. Unfortunately, the CSR SMVM algorithm may access the x array in an irregular order as governed by the $JA[]$ array (Figure 2.2), namely $x(JA[k+1]), x(JA[k+2]), \ldots$, which makes little or no use of cached data, unless nonzeros are closely located as in band matrices.

2.2.3 Inefficient ILP

Modern architectures can achieve parallelism at the instruction level by performing multiple operations simultaneously. One hardware supported method to achieve instruction level parallelism (ILP) is by overlapping the execution of instructions, which is referred to as *pipelining* [HP06]. SMVM algorithms cannot fully exploit this feature, because the amount of independent floating point operations in each inner loop is often too small. Due to lack of parallelism, floating point units stay idle during most of the runtime. Manual or static compiler optimization techniques to improve ILP for dense matrices, such as loop unrolling and vectorization, cannot easily be applied to SMVM because of the small and varying number of nonzero elements per row.

Consider the CSR SMVM algorithm in Figure 2.2. First, the inner loop includes only a single multiply-add operation, which prevents efficient pipelining and leads to high loop overhead. Second, the loop length is determined at run time from the IA array ($k1$ and $k2$), therefore compilers cannot determine an optimal unrolling depth for this loop at compile time. Third, single instruction multiple data (SIMD) programming, i.e., vectorization, assumes contiguous strides of data, whereas SMVM imposes irregular access to memory, making vectorization difficult to implement.

Chapter 3

Operation Stacking Framework

Repeatedly running the same algorithm on sparse problems, with identical instructions and memory access patterns, yet on different data sets, is a common practice in many scientific applications. Model reduction [GAB08, ASG01], weather modeling [GR05], and drug design [MHW04] are examples of such *ensemble computations*. Conventionally, ensemble computations are run independently either sequentially or in parallel, which requires repeatedly reading identical index information from memory. This causes inefficient use of memory bandwidth.

We propose *operation stacking* as a new technique to allow the simultaneous solution of multiple problems that are part of an ensemble in a single, collective operation. As such, only *one* copy of the shared index data is used to solve *all* of the problems in the ensemble, significantly reducing the overall memory bandwidth usage. We developed the *Operation Stacking Framework (OSF)* to provide users with *'stacked solvers,'* which are modified ver-

sions of standard iterative solvers, such as CG and GMRES, that operate on data from multiple problems. In addition to reducing the memory bandwidth usage, stacked solvers process more data per iteration, mitigating inefficiencies caused by the short inner loop of the non-stacked CSR SMVM kernel (i.e., single multiply add per iteration in Figure 2.2). This provides several additional performance advantages, such as reduced loop overhead, more efficient loop unrolling, and improved pipelining. Moreover, OSF can interleave stacked data to improve spatial locality in SMVM operations, which is not possible when solving single problems. Operation stacking is not a new algorithm; instead it is a new methodology for running existing algorithms to improve their overall throughput. The stacked solvers can be used as drop-in replacements in existing codes with minimal effort, requiring no changes to their computation or data patterns.

Stacked codes provided by OSF must handle several complex tasks, such as keeping track of multiple problems in the ensemble, synchronization and communication, stacking and unstacking data, and removing converged problems from the stack. Developing such stacked solvers manually is a non-trivial task. For this reason, we created a set of *developer tools*, which provide stacked code developers with a high level API of functions and macros to conveniently and efficiently convert existing iterative solvers into stacked form. Using this API, developers only need to provide solver-specific data structures and functions via standardized containers. The OSF tools generate the final stacked solver code, requiring no further involvement by developers.

The *OSF Iteration Engine (OSFIE)* controls iterations, detects convergence, and ejects problems as they converge. Though implemented in C, OSFIE uses an object-oriented approach that treats the individual solver algorithm as a polymorphic object, which must implement

an interface we designed. OSFIE uses an efficient algorithm to compress the non-converged problems' data to retain the potential for speedup until only one problem is left in the stack.

We evaluate OSF on the Opteron architecture and demonstrate an average speedup of 1.45× when stacking 4 problems, with an observed maximum of 1.94×. Our analysis using hardware performance counters shows that these improvements stem from better cache utilization, more efficient use of floating point units (FPUs), and significant reductions in memory bandwidth usage. Although ejection of converged problems incurs overhead and reduces the potential speedup, we show that operation stacking retains its performance advantage even for ensembles that contain problems with widely varying convergence characteristics.

This chapter first presents the operation stacking approach, then explains the implementation of the OSF library to develop and use stacked solvers, and concludes with a comprehensive evaluation, which demonstrates and validates OSF's significant performance improvements.

3.1 Operation Stacking

Operation stacking is a technique to reduce memory bandwidth usage by stacking multiple problems with identical matrix structures, so they can share a single set of the index arrays IA and JA, instead of using one per problem. We refer to the number of stacked problems as *stack depth*. For a stack depth of k, the memory bandwidth usage of the index arrays is reduced to $1/k$ of what the non-stacked implementation would use, regardless of the format being used. For a given $n \times n$ CSR matrix with nnz nonzeros, the memory bandwidth savings by operation stacking can be calculated as:

$$Savings\ in\ indexing\ data\ (bytes) = (stackdepth - 1) \times (nnz + n + 1) \times (size\ of\ integer)$$

In addition to reducing the indexing overhead, operation stacking increases SMVM performance by improving utilization of two micro-level hardware components: caches and floating-point units. Operation stacking utilizes *data interleaving*, which is a simple but effective optimization for improving data locality. This data reordering technique [IYV04, MCWK01, HT06] merges multiple *separate* arrays into one interleaved contiguous array, placing data that are accessed in the same iteration together, so they are more likely to fall on the same cache line. For example, if a sparse problem includes multiple right hand side (RHS) vectors, then these vectors can be interleaved to improve cache utilization, as is done in the SPARSITY framework [IYV04]. Operation stacking keeps floating-point units busier due to longer stacked loops, which include multiple independent operations that can be overlapped by pipelining.

OSF interleaves arrays that are accessed irregularly, such as the x array in the CSR-SMVM

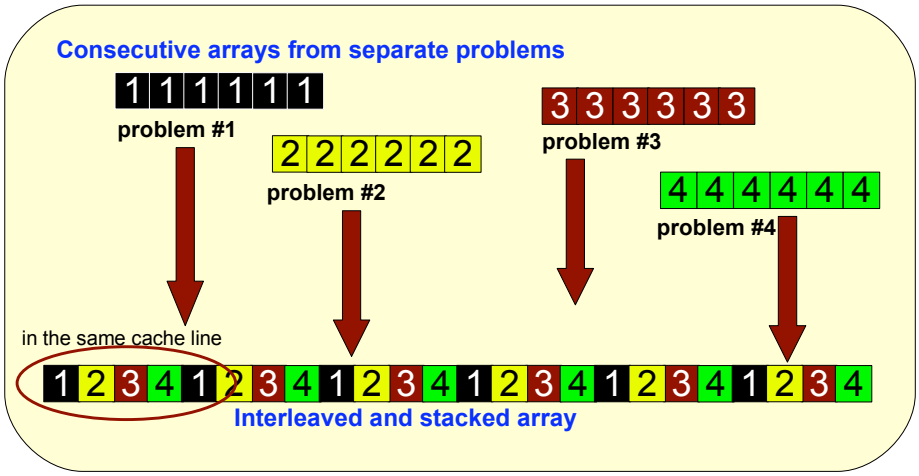


Figure 3.1: Interleaved stacking of data from multiple problems

algorithm given in Figure 2.2. An illustration of how we interleave the vector x for a stack depth of four is given in Figure 3.1. We do not interleave vectors that are being accessed sequentially, such as the array A. Stacked problems with single RHS benefit from interleaving, because interleaving *single* RHS vectors of *multiple* stacked problems works in the same way as interleaving *multiple* RHS vectors of a *single* problem. The current OSF implementation supports interleaving multiple RHS vectors as well, but our evaluation does not investigate this case.

We modified the non-stacked CSR-SMVM algorithm to operate on stacked, interleaved data as shown in Figure 3.2. This stacked algorithm accesses stacked arrays in the same sequence as the non-stacked algorithm, but the inner loop computes the dot product for all problems that are part of the stack. The stacked and interleaved array '*stc_x*' has better spatial locality than the non-stacked array x in Figure 2.2 because it ensures consecutive placement of consecutively accessed elements. In the figure, we denote the $JA(k)^{th}$ element of the stacked *stc_x* array from problem p as $stc\_x(JA(k), p)$, but our actual implementation uses

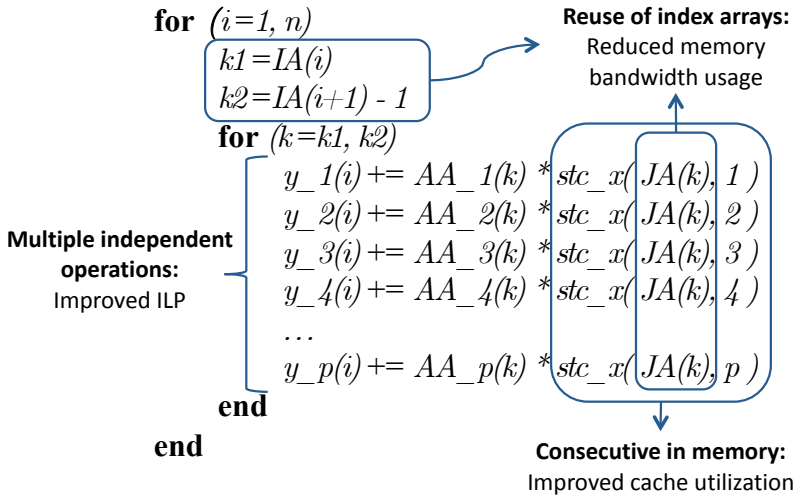


Figure 3.2: Stacked CSR SMVM algorithm

one-dimensional interleaved arrays in memory.

Another benefit of stacked algorithms comes from improved instruction level parallelism (ILP) due to the increased number of independent floating-point operations in the inner loop, which increases the potential of benefiting from compiler-guided loop unrolling and processor-specific instruction scheduling optimizations. To simplify the compiler's job, we manually unroll the innermost loop, which iterates over the problems in the stack, so the compiler can efficiently unroll the outer loop in Figure 3.2. By contrast, the standard CSR-SMVM algorithm in Figure 2.2 shows little benefit from loop unrolling, especially for matrices with a small number of nonzeros per row.

3.1.1 Stacked Solvers

Operation stacking improves the performance of the SMVM kernel, which is repeatedly invoked by iterative solver algorithms. Our experiments show that copying x arrays to/from a single interleaved array *stc_x* (Figure 3.1) for each invocation of SMVM incurs significant overhead. We avoid this excessive data movement by stacking data and operations not only for one kernel invocation, but for the entire iterative solver, turning them into *stacked solvers*. In this way, the data for the SMVM operations will retain their stacked and interleaved state throughout the entire solver run, and interleaving and de-interleaving are performed only once per solver invocation. OSF performs this operation in a transparent way, therefore stacked solvers can be called using conventional, non-interleaved form, requiring no changes to the original code.

An additional benefit of implementing stacked solvers is that the compiler is able to apply optimizations not only to the SMVM kernel itself, but also the code sections that surround this kernel in the solver, such as multiple vector updates that follow the stacked SMVM operations in the same iteration step.

3.1.2 Variable Convergence

Despite having identical matrix structures, stacked problems contain nonzeros with different values, which may cause them to converge at different iteration steps. Although it would be possible to keep already converged problems in the stack until the last problem converges, doing so would lead to unnecessary computations. Instead, we implemented an efficient

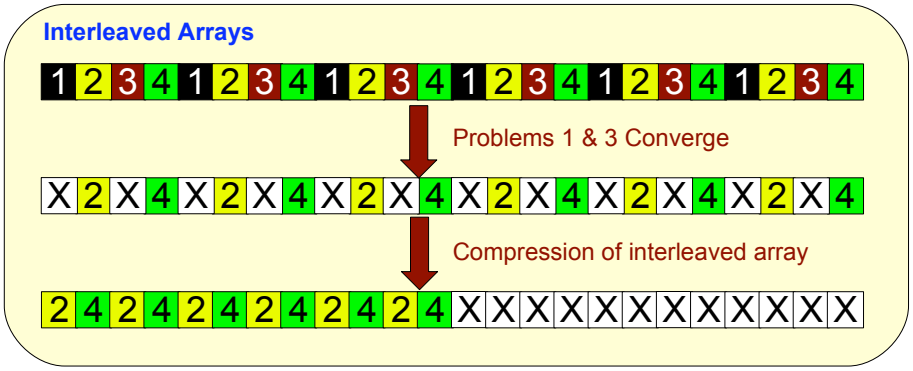


Figure 3.3: **In-place compression of interleaved stacked arrays after partial convergence:** This compression requires only a single pass through the array. Compression of data removes gaps of useless data (marked with ×), which would otherwise be fetched into the cache.

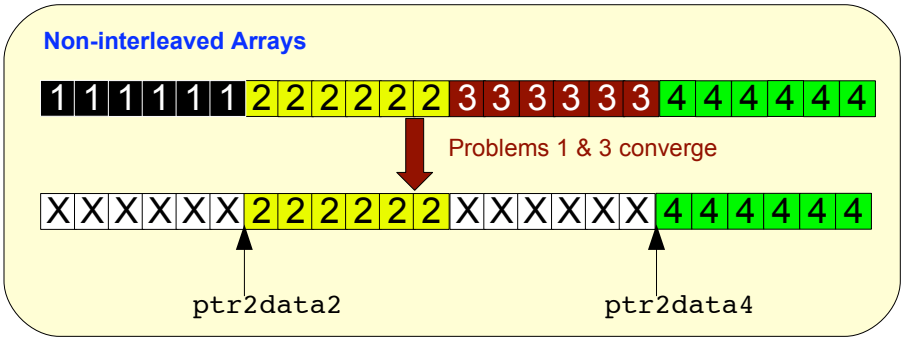


Figure 3.4: **Non-interleaved stacked array segments after partial convergence:** Unlike interleaved arrays, non-interleaved arrays do not require data compression, because their data are already placed consecutively in memory. OSF keeps track of data segments of non-converged problems using pointers.

mechanism to eject problems once they converge.

When problems are ejected from the stack, their data must be removed from the stack and their results must be returned. For interleaved arrays, OSF shifts the elements of the unconverged problems to the left, overwriting array entries previously occupied by converged problems as illustrated in Figure 3.3. This in-place compression requires only a single pass through each array, even when multiple problems converge simultaneously. This compression step is necessary to remove obsolete data (marked with X in Figure 3.3), which would reduce data locality otherwise. Unlike interleaved arrays, non-interleaved arrays need not to be compressed when problems converge. Instead, OSF maintains and updates pointers to those segments within a non-interleaved array that correspond to problems that are still part of the stack, as illustrated in Figure 3.4.

3.1.3 Algorithm Independence

Iterative solvers such as CG (Figure 3.5) and GMRES (Figure 3.6) share a similar structure in which an initialization step is followed by a variable number of iteration steps. OSF exploits this common structure by defining an abstract *"StackedSolver"* class, shown in Figure 3.7, which concrete solvers must implement.

A stacked solver must implement methods for the initialization and iteration steps for every stack depth within a range of (1..8), a method to identify and eject converged problems (`eject_converged_problems()`), and a method to retrieve the maximum number of iterations (`get_max_iterations()`). We use arrays of function pointers (`init[MAX_STACKDEPTH]`

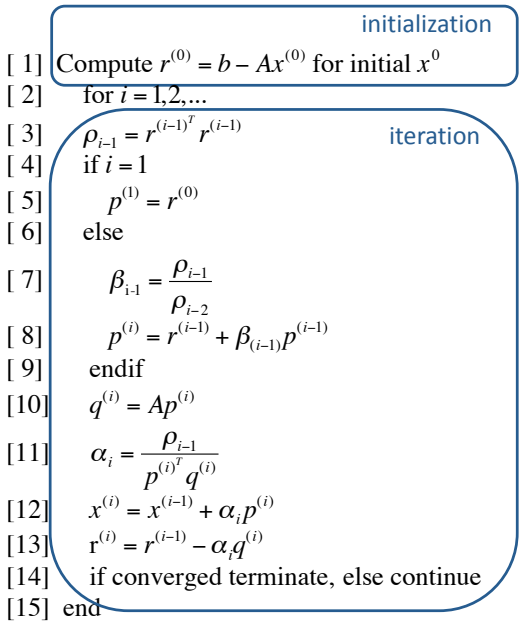


Figure 3.5: The Conjugate Gradient (CG) algorithm.

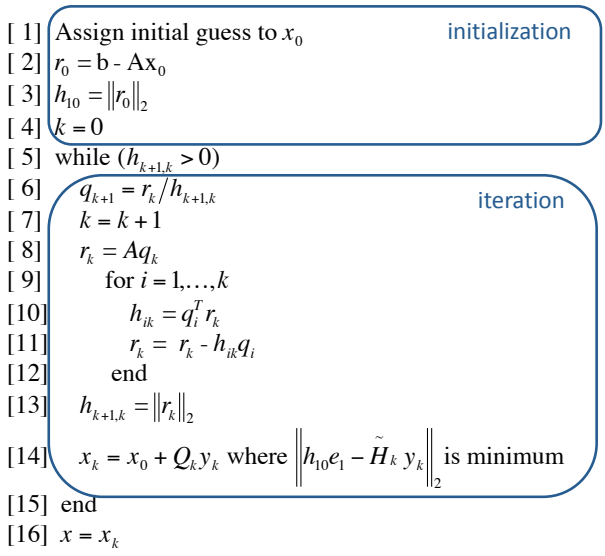


Figure 3.6: The Generalized Minimal Residual (GMRES) algorithm.

and `iterate[MAX_STACKDEPTH]`) to record the initialization and iteration functions for each possible stack depth. The C++-like syntax in Figure 3.7 is used only to illustrate the abstract class we define; the actual implementation uses C function pointer tables to describe each concrete solver implementation.

The OSF iteration engine (OSFIE) operates on abstract stacked solver instances, as shown in Figure 3.8. Starting with an initial stack depth *stackdepth*, the OSF iteration engine invokes the solver's `init[stackdepth]` method first. The engine will then repeatedly invoke the iteration method compiled for the current stack depth, which is the number of remaining non-converged problems in the stack. After each iteration step, the stacked solver may eject converged problems, if there are any, reducing the current stack depth. The iteration ends when all problems have converged or when a problem-specific maximum number of iteration

```
abstract class StackedSolver {
public:
        // Perform initialization
        void (*init[MAX_STACKDEPTH]) ();
        // Perform one iteration step and return
        // the number of converged problems
        int (*iterate[MAX_STACKDEPTH]) ();
        // Learn maximum number of iterations
        int get_max_iterations();
        // Eject converged problems
        void eject_converged_problems();
}
```

Figure 3.7: Abstract class StackedSolver, which stacked solver implementations must implement.

```
void osf_iterator_engine(StackedSolver *solver, int stackdepth) {
    solver->init[stackdepth]();
    int nr_iters = 0;
    while ((stackdepth > 0) && (nr_iters < solver->get_max_iterations())) {
        nr_iters++;
        int nr_converged = solver->iterate[stackdepth]();
        if (nr_converged > 0) {
            solver->eject_converged_problems();
            stackdepth -= nr_converged;
        }
    }
}
```

Figure 3.8: OSF iteration engine.

```
class CG:StackedSolver {
    private:
    int n, nnz;
    // Interleaved arrays
    interleaved_array<double> x[n * MAX_STACKDEPTH], p[n * MAX_STACKDEPTH];
    // Non-interleaved arrays
    double A[nnz * MAX_STACKDEPTH], r[n * MAX_STACKDEPTH];
    double b[n * MAX_STACKDEPTH], q[n * MAX_STACKDEPTH];
    double alpha[MAX_STACKDEPTH], ro[MAX_STACKDEPTH], beta[MAX_STACKDEPTH];
}
```

Figure 3.9: Data structures maintained by a stacked conjugate gradient (CG) solver.

has been exceeded.

Different stacked solvers need to keep track of different algorithm-specific arrays for the intermediate values they compute. For instance, Figure 3.9 shows the variables and arrays comprising the state of a stacked CG implementation, which correspond to the variables used to implement the algorithm in Figure 3.5.

The OSF iteration engine provides the necessary runtime support to keep track of interleaved and non-interleaved vectors. It records which problems had converged and which are still active, and maintains a map of their positions in each stacked vector, which is accessible to the stacked solver code. The stacked solver code can use this map to extract data of converged problems in interleaved vectors and to find the addresses of active segments in non-interleaved vectors. When problems converge and are ejected from the stack, the engine updates this map and automatically performs the necessary compression of all interleaved vectors. Consequently, developers of stacked solvers can focus on the iterative algorithm and do not need to reimplement suitable representations of their data.

3.2 Multi-Process OSF Implementation

Rewriting existing codes to solve multiple problems, for all possible stack depths, would require a significant amount of code modification, including changes to APIs and the representations of internal data structures. To avoid this burden, we implemented OSF as a multi-process framework in which multiple processes, each solving a single problem, combine to solve these problems in a stacked operation. This coordination takes place when each program calls into a stacked solver and is entirely transparent to the application code.

As most existing scientific codes solve a single problem at a time, integration of a stacked version of solvers into existing codes becomes straightforward, requiring only renaming the call sites to refer to the stacked version of a function. For instance, a call to a conjugate gradient solver, `cg`, with signature

```
cg(double *AA, double *b, double *x,...)
```

must be replaced with a call to its stacked counterpart, `osf_cg`, with identical signature

```
osf_cg(double *AA, double *b, double *x, ...)
```

and does not require changes to the representations of `AA`, `b`, or `x`.

3.2.1 Collective Operation by Multiple Processes

To initiate operation stacking users execute a driver program (*osfrun*), which simultaneously starts *stackdepth* number of participating processes for *stackdepth* individual problems. Each process inputs its own data set. When the participating processes call a stacked solver function, they cooperate and synchronize to perform a collective computation. The data segments contributed by each of the processes must be arranged and combined according to the requirements of the solver implementation. This arrangement requires data to be copied from the private memory of processes to a shared memory location, some in interleaved form. Since we implement stacked solvers, rather than stacked SMVM kernels, the cost of this data copying and synchronization is paid only once for each invocation of a stacked solver.

Though OSF involves multiple processes, all stacked computations are single-threaded. A *leader* process is elected to perform the stacked computation, e.g., the stacked CG algorithm in Figure 3.2, on behalf of the remaining processes. Before the stacked computation, each process contributes its data into a shared memory area on which the leader operates. Processes that are not the leader wait on a per-process semaphore for the leader to complete the computation of their problem, without consuming any computational resources themselves.

OSF starts operations by assigning ranks to all participating processes. The rank of a process determines the position of its data in each stacked array. When one or more problems converge, the leader wakes up the corresponding processes, which then copy the solutions to their private memory spaces. To ensure that the leader can safely continue with the remaining problems in the stack, the leader and the converged processes synchronize using an n-way barrier when the solutions have been copied out. At this point, the leader ejects

converged problems from the stack, compresses the stacked data as necessary, and continues iterations at the reduced stack depth.

If the leader's problem is amongst the converged, then a new leader must be selected to continue the stacked iterations before the outgoing leader ejects itself from the stack. The current OSF implementation chooses the non-converged process with the highest rank as the new leader. The outgoing leader signals the semaphore of this new leader, which then wakes up and takes over the stacked operation. OSF keeps problem-specific critical information, such as the iteration step and the current stack depth, in an internal struct, which is passed from one leader to the next.

Our implementation relies on the shared memory and semaphore capabilities described by the POSIX 1003.1b standard [Ame94]. We also use shared *pthread* barriers, which are an optional part of the same standard. Recent versions of Linux support this feature; on other platforms, we could implement barriers using shared semaphores or condition variables.

3.2.2 Example Scenario

Figure 3.10 illustrates an example OSF scenario for the stacked CG algorithm with four processes. At time step 1, all processes have entered into a stacked function. In this example, the participating processes' problems converge in the following order: first, process #1 and #2 converge at the same step, then #4, then #3.

The first process to arrive at time step 1 creates shared memory sections that are later filled with data from participating processes. All other processes copy their data into the shared

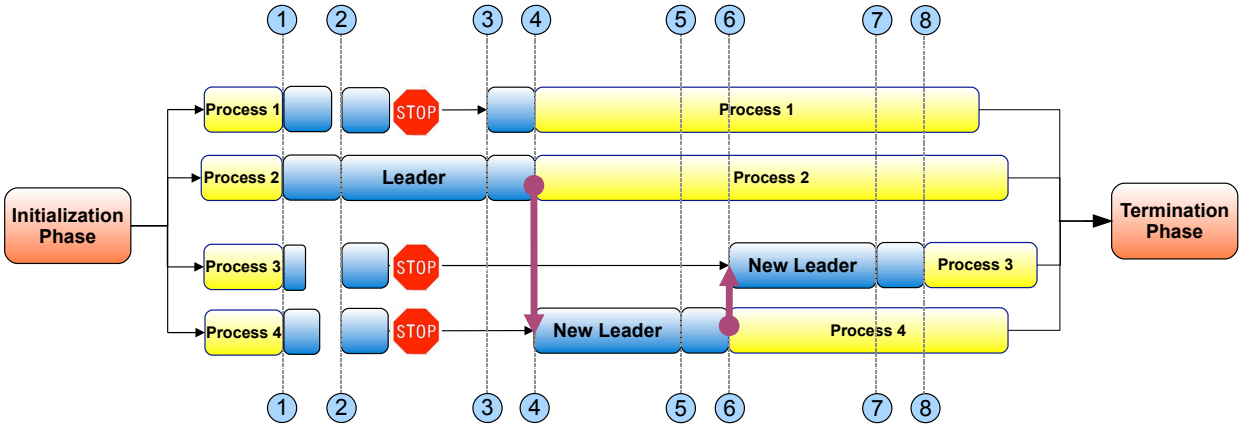


Figure 3.10: An example scenario for variable convergence.

memory section based on their rank, which is determined by the order of arrival at time step 1. Time step 2 represents a barrier, which is necessary to ensure that all processes have completed copying their data. To avoid additional context switches, the last process to arrive at the barrier becomes the initial leader (process #2 in this example). Between time steps 2 and 3, the leader implements the stacked algorithm, e.g., the stacked CG in Figure 3.2, while other processes wait on their semaphores.

At time step 3, problems #1 and #2 converge at the same iteration step. When this happens, the leader signals process #1, which then copies the results back to its local address space. Being one of the converged processes, the leader also copies its results to its private memory and compresses the stacked data in shared memory. Subsequently, processes #1 and #2 rendezvous at a 2-way barrier at time step 4. After the barrier, process #1 ejects and returns to its caller. Process #2 signals process #4, a non-converged process with the highest rank, to become the new leader, and ejects itself from the stack. Upon wake up, process #4 takes over the stacked computation on the remaining problems (#3 and itself) until time step 5,

when it finally converges. Subsequently, process #4 copies out the results, compresses the shared array, signals process #3 to become the new leader and ejects itself at time step 6. Being the only process left in the OSF stack, process #3 runs the regular (non-stacked) algorithm until it converges at time step 7. Upon convergence, it copies back its results and ejects itself in time step 8, concluding the stacked operation. After all of the processes have returned, they can call new stacked functions as needed.

3.2.3 Process and Resource Management

OSF's use of interprocess facilities such as shared memory requires prior setup and posterior cleanup. These tasks are performed by the *osfrun* driver script. This script, which is not specific to a particular solver implementation, user application, or problem size, initializes the system for the communication and synchronization of the processes participating in a stacked ensemble. It creates a shared memory area, shared semaphores, and locks used to bootstrap the rank assignment process at time step 1 of Figure 3.10.

To ensure that all shared memory areas are destroyed even in the event that some or all participating processes terminate prematurely on error, we placed a trap-on-exit bash handler in the OSF driver script. In Linux, all shared memory areas and semaphores are visible as files in the `/dev/shm` filesystem, allowing the exit handler to locate and remove all shared memory objects related to that particular stacked ensemble.

3.3 Developing Stacked Solvers Using OSF

OSF's runtime library supports the development of stacked solvers by addressing two main goals. First, it shields the stacked solver developer from the low-level details of stack management and ejection of converged problems. When implemented as a multi-process approach as described in Section 3.2.1, OSF's runtime support encapsulates the required process synchronization and shared memory management. Consequently, OSF separates the specific iterative algorithm being implemented from the concern of stacked operation. Second, this library separates these concerns without compromising stacking's performance benefits as described in Section 3.1. Because some of these benefits derive from enabling manual and compiler-driven optimizations, achieving this second goal necessitates compile-time support for the automatic generation of optimized versions of solvers for each stack depth. To that end, developers provide code templates, which are instantiated once for each stack depth.

3.3.1 Runtime Support

Stacked solver code templates must provide the `init()`, `iteration()`, `get_max_iterations()`, and `eject_converged_problems()` methods listed in Figure 3.7. To facilitate the implementation of these methods, developers use OSF's runtime library API, shown in Table 3.1. The API supports multi-process related functionality such as leader election, rank distribution, shared memory management, and process synchronization. Second, it supports the management of interleaved arrays and their compression during ejection. For example, the `init` method of a stacked implementation of CG uses this API to create and register the interleaved arrays for the variables shown in Figure 3.9. Third, the API provides a function to

Table 3.1: OSF Runtime Library API

Name	**Arguments**
`int osf_get_stackid`	`none`
Desc. Returns the ID number of an OSF instance (stackid), which is shared among all participating processes.	
`int osf_get_stackdepth`	`int stackid`
Desc. Returns the initial stackdepth of an OSF instance.	
`int osf_get_curstackdepth`	`int stackid`
Desc. Returns the current stackdepth, which is the number of actively participating processes when it is called.	
`int osf_get_rank`	`int stackid, int *rank, rank_callback_t cbfunc, void *cbdata`
Desc. Elects the initial leader and returns the OSF rank of the caller process. Invokes callback function in serialized execution.	
`void* osf_create_shared_array`	`int stackid, char *shr_name, char shr_type, int size`
Desc. Creates a shared memory array and returns a pointer to its virtual address. It uses shr_name and stackid to construct a unique name for this array.	
`void* osf_open_shared_array`	`int stackid, char *shr_name, char shr_type, int size`
Desc. Returns a pointer to the virtual address of an array, which must already be created by osf_create_shared_array using the same shr_name and stackid.	
`void osf_register_interleaved`	`int stackid, struct _stackstate *stackstate, void *array, char shr_type, int size`
Desc. Registers the input array as interleaved.	
`void osf_remove_from_stack`	`int stackid, struct _stackstate *stackstate, int process_no`
Desc. Finds the rank of the input process and marks it as ejected.	
`void osf_stacked_operation`	`int stackid, struct _functionstate *alg, void *data`
Desc. Initiates the OSFIE to execute stacked codes in the proper order.	

(a) Code generation template

```
#define DEFINE_SUM(_si) double sum##_si;
OSF_STACKED_OPERATION_SAME_BLOCK(DEFINE_SUM)
for (k=k1; k < k2 + 1; ++k) {
    #define MULT_AA_BY_X(_si) \
    sum##_si += aa_ptr2data##_si[k] *_IL_(oldx,(ja[k]-1), stackdepth, _si)
    OSF_STACKED_OPERATION(MULT_AA_BY_X);
}
```

(b) Generated SMVM code (for stackdepth=4)

```
double sum0, sum1, sum2, sum3;
for (k=k1; k < k2 + 1; ++k) {
    sum0 += aa_ptr2data0[k] * oldx[((ja[k]-1) * stackdepth)+0];
    sum1 += aa_ptr2data1[k] * oldx[((ja[k]-1) * stackdepth)+1];
    sum2 += aa_ptr2data2[k] * oldx[((ja[k]-1) * stackdepth)+2];
    sum3 += aa_ptr2data3[k] * oldx[((ja[k]-1) * stackdepth)+3];
}
```

Figure 3.11: Code generation using OSF macros.

start the execution of the OSF iteration engine.

3.3.2 Compile Time Code Generation

The key reason for using compile time code generation is to facilitate manual and compiler optimizations that depend on knowing the stack depth at compile time. To achieve this goal, we make creative use of the C preprocessor's macro expansion facilities.

As an example, consider the template shown in Figure 3.11-a, which implements the body of the inner SMVM loop in Figure 3.2. When expanded for a stack depth of four, the code shown in Figure 3.11-b is generated by the C preprocessor, which is equivalent to a manually created version a developer would have provided for this stack depth. This template uses the `OSF_STACKED_OPERATION_SAME_BLOCK` preprocessor macro we provide. This macro expects as its argument a second preprocessor macro (here: `DEFINE_SUM`), which is instantiated *stack-*

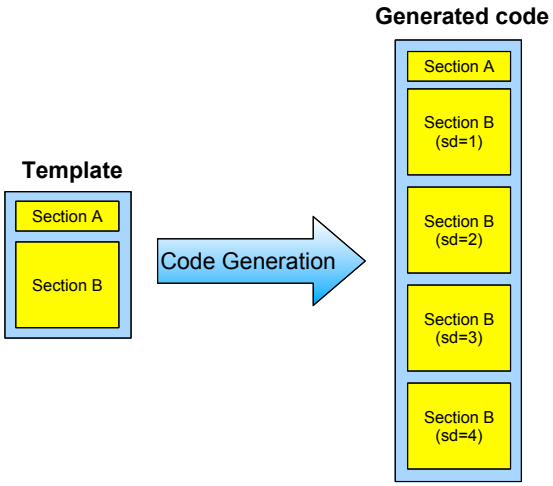


Figure 3.12: OSF code replication.

depth times for each of the values {0..3}. As a result, definitions for *stackdepth* temporary local variables { `sum0..sum3` } are generated, which hold the intermediate summation results for each problem. The `OSF_STACKED_OPERATION` macro functions in a similar way, except that the expanded form is placed in its own C block scope.

The macros being passed may use the stack index (_si) as an argument in any way desired, including to generate new identifiers via token concatenation using the `##` operator. For instance, the `MULT_AA_BY_X` macro creates references to identifiers '*aa_ptr2data0*', '*aa_ptr2data1*', etc., which represent sections of a non-interleaved array that contain the data of the problems left in the stack at this stack depth. In addition, the developer-provided macros may make use of convenience macros such as (`_IL_`), which expand to the arithmetic expression needed to index values in interleaved arrays. This approach to code generation manually unrolls an otherwise required loop, increasing the potential for compiler optimizations.

Since each stacked solver must provide definitions for the `init()` and `iteration()` functions for each stack depth, the code template is included multiple times. Between each inclusion, we suitably redefine all `OSF_*` C preprocessor macros. This multiple inclusion approach is illustrated in Figure 3.12. Code sections in the template that do not need duplication, depicted as 'Section A' in this figure, are protected by traditional `#include` guards.

In Appendix A, we demonstrate the conversion of a regular CG solver into a stacked solver using the OSF developer library in more detail. We use the resulting stacked CG algorithm extensively in our evaluation in Section 3.4.

3.4 Evaluation

Our evaluation pursues several goals. First, we show how operation stacking improves "time to solution," or overall performance, by measuring a stacked implementation's speedup over a non-stacked implementation that solves the same set of problems. Second, we exploit hardware event counters to validate our expectation that the observed performance improvements stem from increased cache utilization and increased floating-point pipeline utilization. Third, we investigate the relationship between the speedup and the number of iterations a solver performs, with particular emphasis on scenarios in which stacked problems converge at different steps.

We applied OSF's compile-time code generation support and the runtime library presented in Section 3.2 to convert a non-stacked implementation of a CG solver [Saa96], which uses the SMVM algorithm given in Figure 2.2, into stacked form (see Appendix A for more detail). We also used our OSF framework to create a stacked implementation of the GMRES algorithm [BBC+94]. We omit results for GMRES because they are substantially similar to CG, which is not surprising as both solvers are dominated by the same SMVM kernel. Our experiments show that, on average, 93% of CG's and 89% of GMRES's overall runtime are spent in the SMVM kernel, which operation stacking optimizes.

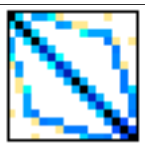
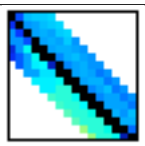
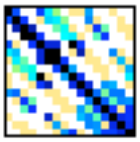
We consider both synthetic and real-world matrices, but restrict our analysis to matrices that do not fit into the cache. We compared our speedup results to the speedups obtained for a previous version of OSF that did not use the object-oriented framework [BRB07]. We found negligible differences, allowing us to conclude that the decoupling of the iteration engine and stack management described in Section 3.2 does not impose noticeable runtime overhead.

3.4.1 Methodology

For the synthetic portion of our benchmarks, we generated $n \times n$ random matrices for $n = 256,000$, $512,000$ and $1,024,000$. For each size, we create three versions by placing 5, 7, and 9 elements per row (epr). Such extremely sparse random matrices are highly challenging for existing blocking approaches. We also explored a set of matrices from the University of Florida Sparse Matrix Collection [Dav] that represents real world problems from a variety of application areas. Properties of these matrices are given in Table 3.2. We explore the effects of stacking for stackdepths 2, 4, and 8. The results presented here show only the time spent in the solver function, because the remainder of the code is identical for both stacked and non-stacked codes. We exclude the overhead of the OSF initialization process, because it is constant and minor when compared to the total execution time. For instance, the initialization overhead is only 0.12% of the total runtime of the smallest (256000 × 256000, 5 epr) and 0.02% of the largest (1024000 × 1024000, 9 epr) generated matrix in our set, assuming a stackdepth of four and 32 iterations. For evaluation purposes, we replace the convergence check of the CG algorithm with a check that stops iterations after a set number of iterations, which we vary based on the convergence scenario we wish to explore.

We use an AMD Opteron 240 with dual 1.4GHz CPUs as our test platform, running Linux kernel 2.6.18, with 2GB of RAM, a 64KB 2-way associative L1, and a 1024KB 16-way associative L2 cache. We use the 64 bit *gcc* compiler with the *-O2* and *-funroll-loops* optimizations, because this combination provided the best performance for both the stacked and the non-stacked code. We use IEEE-745 double precision floating-point arithmetic via the SSE instruction set. The *PAPI* library [BDH+00] provided access to the processor's hardware event counters. Appendix B includes supplementary experimental performance

Table 3.2: Selected matrices from the UF Matrix Collection.

Matrix Name	**N**	**Nonzeros**	**Sparsity (%)**	**Origin**	**Matrix Structure**
af23560	23560	460598	0.08298	CFD	
FEM_3D_thermal2	147900	3489300	0.01595	FEM 3D	
g7jac160sc	47430	564952	0.02511	Economy	
Pres_Poisson	14822	715804	0.32582	CFD	
srb1	54924	2962152	0.09819	Structural	
torso3	259156	4429042	0.00659	2D/3D	
troll	213453	11985111	0.02630	Structural	

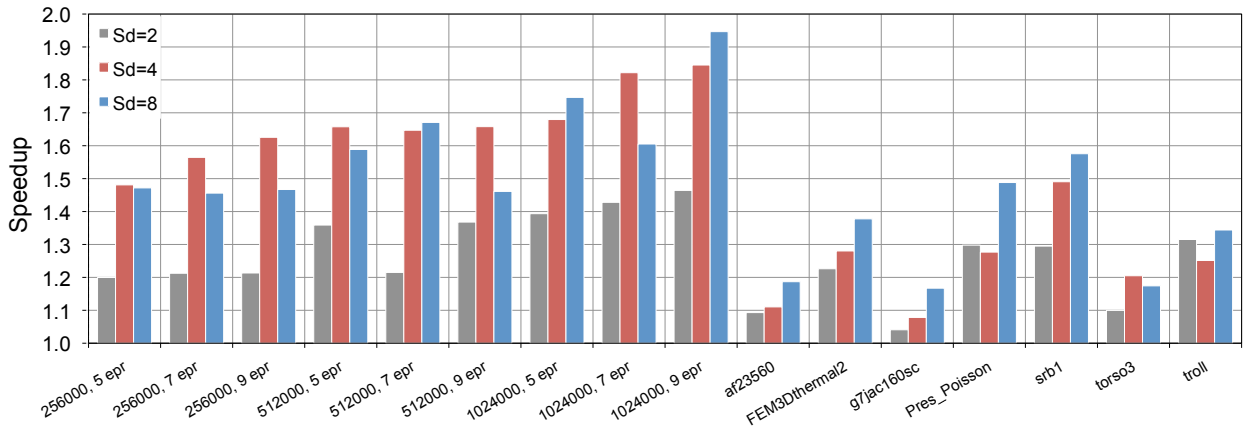


Figure 3.13: Performance of stacked CG solver.

results for a different set of matrices and a 1 Ghz AMD Athlon architecure.

3.4.2 Performance Analysis

Figure 3.13 depicts the performance improvements by OSF for the stacked CG algorithm for stackdepths 2, 4, and 8. The x-axis refers to the matrices we considered, where *epr* denotes the number of elements per row for the generated matrices. For these experiments, we performed 32 iterations for all problems in the stack. We calculate speedup as follows, where 'Time*pp*' denotes the time per problem, which is the time spent on one problem in the non-stacked case and the total time divided by the stackdepth in the stacked case:

$$\text{Speedup} = \frac{\text{Time}pp_{Non\text{-}Stacked}}{\text{Time}pp_{Stacked}}$$

Figure 3.13 shows that stacking can provide speedups of up to 1.94×, with an average

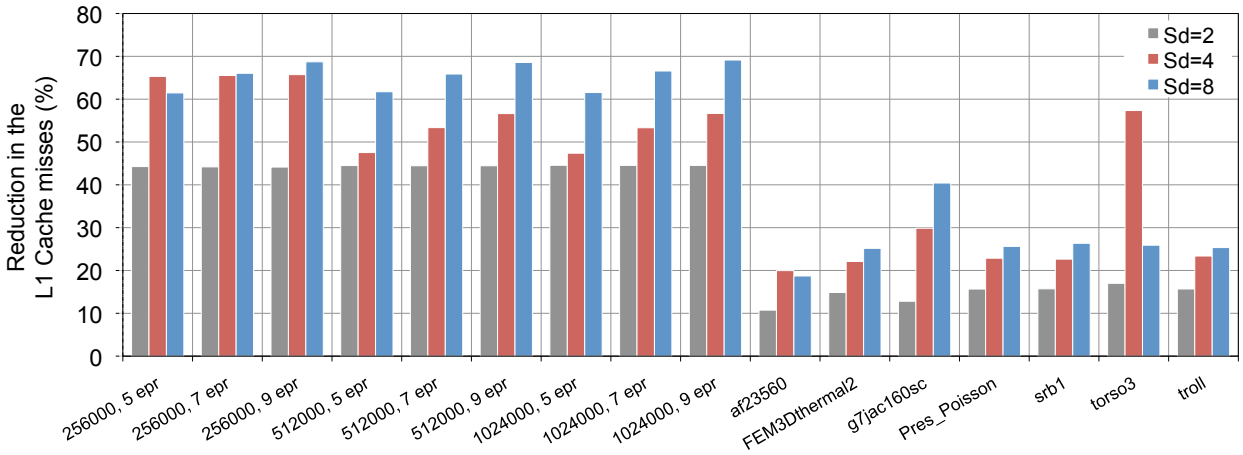


Figure 3.14: Reduction in the number of L1 Cache Misses by OSF

speedup of 1.24×, 1.45×, and 1.46× for stackdepths 2, 4, and 8 for the matrices considered. Speedups are higher for the synthetic matrices, which have a random structure, compared to the real-world matrices, many of which exhibit various degrees of bandedness. A banded structure tends to improve the locality of accesses to x in nonstacked implementations, therefore improvements by operation stacking are less pronounced for these cases.

For most matrices, stacking provides significant performance improvements for stacks containing at least two problems. Therefore, stacking can be used even in situations in which the amount of available memory does not permit larger stackdepths.

We measured both L1 and L2 cache misses to verify our assertion that operation stacking increases cache performance. Figure 3.14 shows that operation stacking reduces the number of L1 cache misses by up to 69% when compared to the non-stacked algorithm. Fewer L1 cache misses means that the L2 cache is accessed less often, decreasing the likelihood of a L2 miss. Figure 3.15 verifies this effect by showing an observed maximum of 80% reduction in the number of L2 cache misses.

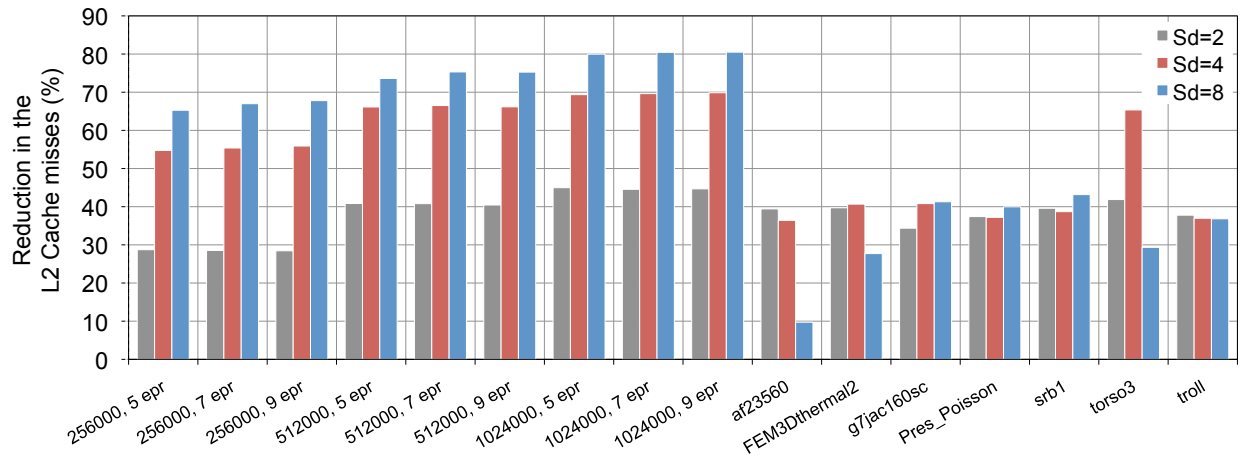


Figure 3.15: Reduction in the number of L2 Cache Misses by OSF

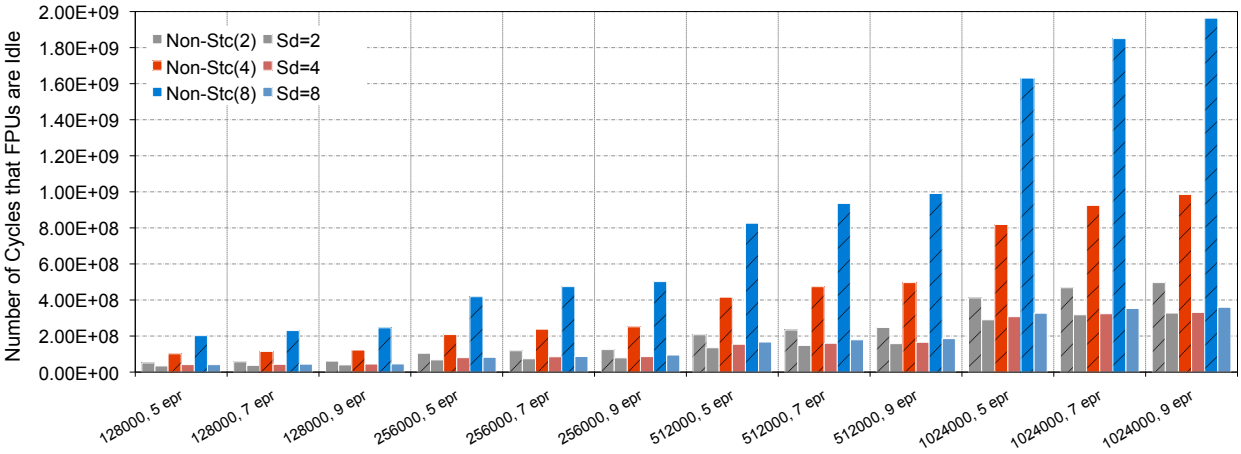


Figure 3.16: Number of cycles floating-point units are idle

In addition to improving cache utilization, we investigate if operation stacking improves floating-point utilization, which is an indicator of higher instruction-level parallelism, thus improved overall performance [HP06]. We use a hardware event counter to count the number of cycles during which the floating-point unit is idle, which is provided by PAPI's PAPI_FPU_IDL event. Figure 3.16 shows a significant and consistent reduction in the number of idle cycles due to stacking.

Finally, we compute the memory bandwidth savings operation stacking achieves because it reuses the index arrays IA and JA for multiple problems, as shown in Figure 3.2. For the matrices used in our experiments, we compute that operation stacking reduces the memory bandwidth usage by 15.3%, 23%, and 27% on average for stackdepths 2, 4, and 8, respectively.

3.4.3 Variable Convergence Scenarios

The overhead of combining problems into a stack and ejecting converged problems must be amortized before performance gains are realized. We investigated how many iterations it takes for this overhead to be amortized under the assumption that all problems converge simultaneously. Figure 3.17 shows the results for a sample synthetic matrix with $n = 512,000$ and 5 elements per row. For this matrix, the overhead is amortized after only 3 iterations; substantial speedup is seen for as few as 10 iterations. We note that stacking provides this speedup for ensemble computations without the tuning step required by specialized SMVM libraries such as OSKI, which often is compensated only after hundreds or thousands of repetitions.

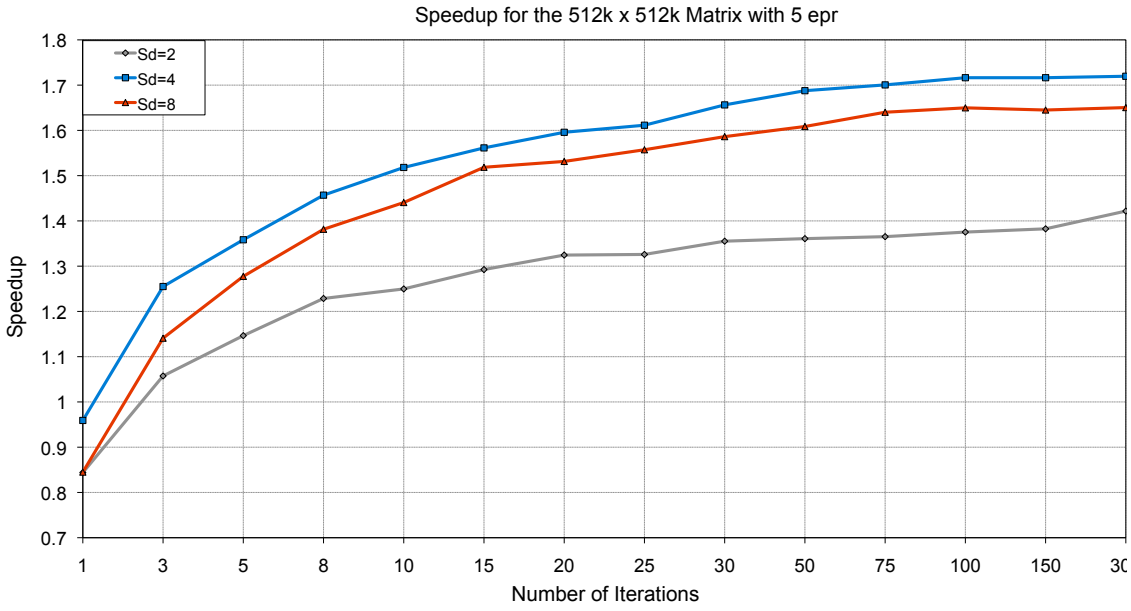


Figure 3.17: The impact of number of iterations on performance.

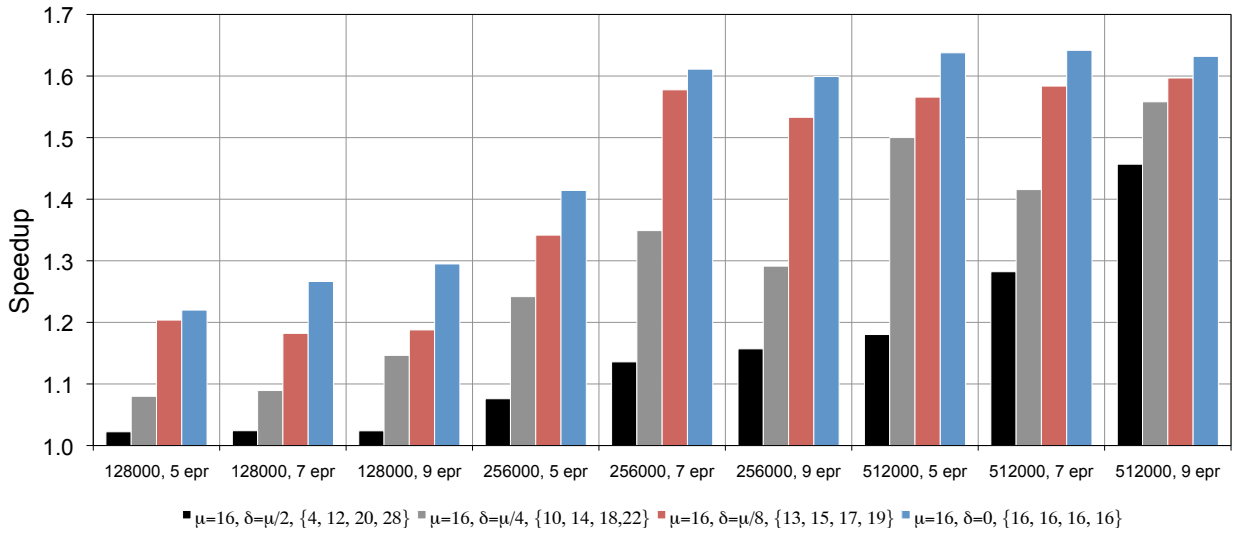


Figure 3.18: Comparison of convergence scenarios for $\mu = 16$

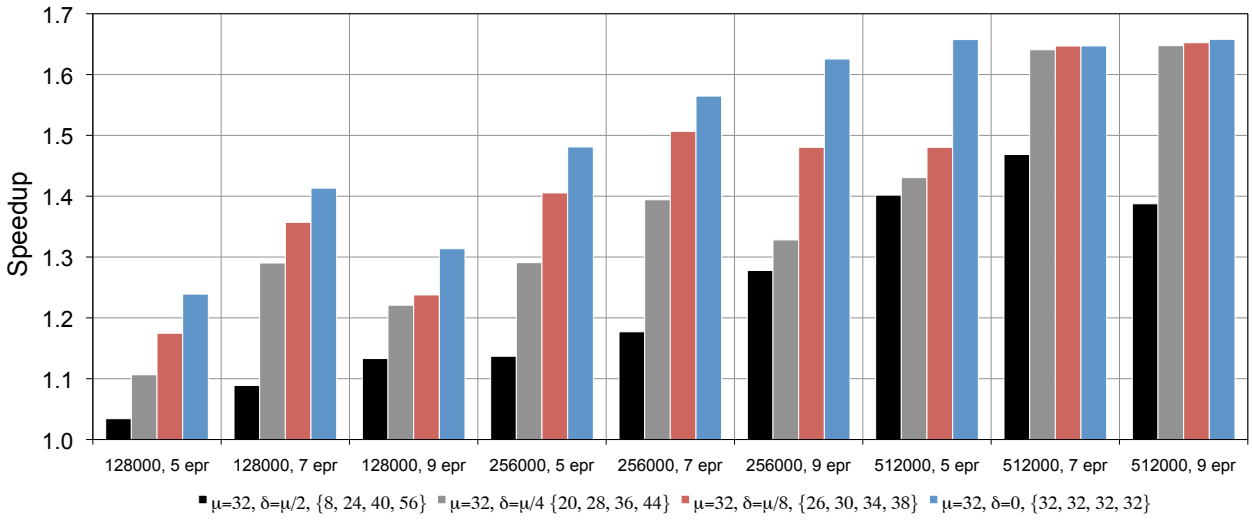


Figure 3.19: Comparison of convergence scenarios for $\mu = 32$

Ejection of converged problems affects OSF's performance in two ways. First, there is a direct cost of ejecting one or more converged problems, which includes the cost of compressing interleaved arrays and of synchronization between the leader process and the converged processes, as described in Section 3.2.1. Second, ejection of converged problems incurs an indirect cost because the stackdepth of the ensemble, thus the efficiency of stacked operations, is decreased for the remaining iterations.

To investigate the performance impact of providing support for variable convergence, we benchmark a number of carefully choreographed ejection scenarios. In all scenarios, we assume the worst case for the direct costs of ejection, which arises when problems converge at different steps and there is a leader reassignment each time since the leader's problem converged. Convergence of the leader incurs an additional synchronization operation to wake up a successor to continue the computation, as described in Section 3.2.1.

Our scenarios consider an equally spaced distribution of the number of iterations. Given a

Table 3.3: Convergence Scenarios

μ, δ	$\mu = 16$	$\mu = 32$
$\delta = 0$	{16, 16, 16, 16}	{32, 32, 32, 32}
$\delta = \mu/8$	{13, 15, 17, 19}	{26, 30, 34, 38}
$\delta = \mu/4$	{10, 14, 18, 22}	{20, 28, 36, 44}
$\delta = \mu/2$	{4, 12, 20, 28}	{8, 24, 40, 56}

mean number of iterations μ and a range parameter δ, we explore a stack of initial depth 4 and eject problems after $\mu - 1.5\delta, \mu - 0.5\delta, \mu + 0.5\delta$, and $\mu = +1.5\delta$ iterations. The coefficients for δ were chosen such that the total number of CG iterations, and thus floating-point operations, is identical for all scenarios (4μ). Therefore, any observed performance difference between the scenarios is attributable to the indirect costs of ejection. Table 3.3 outlines the ejection scenarios we consider for $\mu = 16$ and 32 and $\delta = 0, \mu/8, \mu/4$, and $\mu/2$. We believe these convergence distributions are representative of many real-world ensemble computations, especially if one considers that practitioners commonly exploit preconditioning to avoid excessive numbers of iterations.

Figures 3.18 and 3.19 show our results for $\mu = 16$ and $\mu = 32$. As expected, the highest speedup is seen in the $\delta = 0$ case, in which all problems complete at the last step simultaneously, because OSF benefits from the full stackdepth during the entire stacked run. Larger values of δ, which cause some problems to be ejected earlier, reduce the speedup as the indirect cost of ejection increases. However, our results show significant speedup for all ejection scenarios, including the pessimistic $\delta = \mu/2$ case.

As an alternative to ejection, we consider the option of leaving already converged problems in the stack and performing a small number of redundant iterations on them, which could

pay off in scenarios in which all problems converge at about the same number of iterations. However, we find that even for scenarios such as {29, 30, 31, 32}, in which the number of redundant iterations would be relatively small, we obtain better performance if converged problems are ejected immediately.

Chapter 4

Pattern-based Representation

Pattern-based Representation (PBR) is a novel technique to improve the performance of SMVM operations by reducing memory bandwidth usage, and also by allowing efficient micro-level optimizations and parallelism. PBR identifies recurring block patterns in a given matrix, then generates a *custom code* for each, eliminating the need for reading location information for nonzeros in blocks from memory. As a result, PBR makes SMVM operation less memory bound compared to other existing methods.

Similar to PBR, several other blocking techniques in the literature [IYV04, VDY05, IY01, NVDY04, NVDY07, VM05] also keep indexing information only for blocks, but make the assumption that blocks are *dense*, i.e, they contain no zeros. There are several drawbacks of this assumption. First, the matrix may not contain regions that include dense blocks. Second, searching for dense blocks in a matrix is a non-trivial and costly task, which often requires heuristic approaches with non-guaranteed results [Vud03]. Third, these techniques

resort to zero filling to convert near-dense blocks into dense blocks, causing increased memory traffic and wasting CPU cycles. In contrast, PBR makes no assumptions about the density or structure of nonzeros in blocks, therefore requires neither searching for dense blocks nor zero-filling. Instead, PBR detects and records existing recurring patterns in the matrix using a linear, low-cost analysis. Then, tuned custom codes are generated for each block pattern that meets two criteria we set as efficiency threshold. The generated custom codes implement an operation that matches the corresponding nonzero pattern, thereby eliminating the need for indexing information for individual nonzeros inside the blocks. Once generated codes are fetched from memory, they are reused for all blocks that share the same pattern, creating locality via the processor's instruction cache to significantly reduce memory bandwidth usage.

Converting a matrix into PBR is a multi-step process, starting with a matrix analysis to identify recurring patterns for all candidate block sizes, from 2×2 to 8×8, and finding out which block size is likely to yield the best performance. This step includes two challenges, which we address in this dissertation. First, this analysis must incur low overhead because some applications need to convert matrices at runtime, for example, when the matrix structure is not known beforehand. Second, the performance for each candidate block size must be predicted accurately, so that the optimal block size can be identified without actually benchmarking the kernel. The step following the analysis is the conversion of the input matrix into the PBR format for the predicted optimal block size. This step requires re-arrangement of the matrix data, because PBR places nonzeros belonging to a shared pattern into consecutive memory locations to preserve locality. In this way, when the custom code is called at runtime, it is re-used to process all of the blocks that share the pattern for which the code was generated. Similar to the matrix analysis, this step should also incur low

overhead to make the on-the-fly conversion of matrices feasible. The third and final step is the generation and compilation of custom codes for each shared pattern. We fine tune these codes using several manual and compiler optimizations for improved performance.

We developed the *Pattern-based Representation (PBR)* library to handle all three steps efficiently and conveniently. This library provides users with drop-in SMVM kernel replacements, minimizing the number of modifications users need to make to their codes. The PBR library analyzes the matrix for all candidate block sizes in a single pass, thus incurring an asymptotic time complexity that is linear in the number of nonzeros in the matrix. Then, it uses a multi-regression linear model to predict the performance of PBR with different block sizes, taking into account the varying memory access patterns of different block patterns, to select an optimal block size. We validated the accuracy of this performance model and show that performance loss due to mispredicted block patterns does not exceed 3% on average.

PBR utilizes *matrix splitting* to express the matrix as a sum of several submatrices, each containing only the nonzeros coming from blocks with identical patterns. PBR represents these submatrices in block coordinate (BCOO) format, along with a *blockcode* bitmask that identifies the pattern. The PBR library can apply data reordering required for splitting in a single pass, since the previously completed matrix analysis recorded each nonzero's final destination. As a result, each nonzero is placed directly to its final memory location, needing neither sorting nor shuffling of matrix nonzeros, therefore the structural conversion incurs a linear time complexity. The PBR library also allows reuse of converted matrices by providing necessary I/O routines to store the PBR representation to disk.

The PBR library includes a custom code generator that emits codes for each shared pattern.

This code generator exploits the structural information gathered from the matrix analysis step to implement three efficient micro-level optimizations: explicit prefetching, vectorization, and loop unrolling. First, explicit prefetching ensures that data is brought directly into the L1 cache and tagged with the correct temporal locality. We exhaustively search for the optimal prefetch distance for each architecture. Second, we have adapted our code generator to emit vectorized codes using SSE-intrinsics [INT06, INT09a, AMD09], which is possible since the substructure for each block code is known at code generation time. Third, knowing the block structures also allows inlining independent operations for each nonzero element without introducing an inner loop, which improves pipelining efficiency and reduces loop overhead per operation. PBR-generated codes also implement thread-level parallelization using a custom thread pool we developed. Our evaluation shows that compilation of generated codes constitutes the largest share of the overall PBR cost by a large margin. However, the PBR library can greatly reduce the need for code compilation by maintaining a code cache, which allows storage and reuse of compiled codes for patterns that have been previously encountered.

PBR excludes patterns that include less than three nonzeros, or cover less than one thousand nonzeros overall, because they do not provide benefits when compared to existing techniques, e.g., CSR. Nonzeros of excluded patterns are placed in a remainder submatrix kept in CSR format, which we do not optimize. As stated by Amdahl's law [Amd67], the impact of an optimization is bounded by the fraction of time an algorithm spends in the optimized section, which is expressed by the nonzero coverage in the PBR case. We conducted a comprehensive study on a large set of matrices gathered from the University of Florida Sparse Matrix Repository [Dav] and showed that matrices derived from underlying problems with a 2D/3D

physical geometry are more likely to have high coverage, and therefore are more amenable to PBR. This study also revealed many examples of no-geometry matrices that yield high coverage by PBR, although high coverage is not as common as for those. Another promising result is that only 31 patterns on average (961 max) are sufficient to represent matrices with PBR, which suggests that generating code for each recurring pattern is a feasible task.

We evaluate PBR on two recent AMD and Intel architectures and demonstrate speedups by up to 3.4× over sequential CSR and 5× over parallel CSR. Since the PBR library supports online conversion of matrices, we also evaluate PBR's runtime costs relative to its benefits, and show that PBR costs can be compensated for after a few hundred SMVM operations. When code must be generated and compiled for all shared patterns, thousands of iterations may be required to break even. Fortunately, a modest-sized code cache maintained by the PBR library can exhibit a high hit rate in realistic settings, significantly reducing the need for code generation/compilation.

This chapter introduces PBR as an efficient and practical new method that significantly improves performance of sparse computations, especially problems that involve many repeated SMVM-based linear solves with structurally identical matrices (e.g., turbulent flow simulations [PCM98, HCK93]), where an overhead of hundreds or thousands of iterations can easily be compensated for.

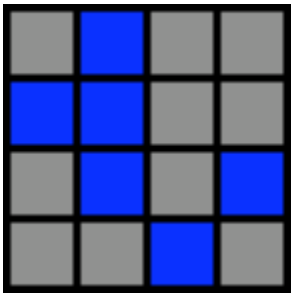
4.1 Fundamentals of PBR

In this section, we explain the fundamentals of PBR in detail. First, we explain how recurring patterns are identified and exploited to reduce memory bandwidth usage. Then, we explain the relationship between the block size and PBR nonzero coverage, and provide statistics for PBR coverages for different matrices and corresponding savings in their indexing overhead.

4.1.1 Exploiting Recurring Patterns

Since the primary factor limiting SMVM performance is its memory boundedness, PBR primarily focuses on reducing memory traffic, which is mainly created by arrays x and y and the sparse matrix itself. There is not much we can do to eliminate or reduce memory operations on arrays x and y, and also on the nonzero values in matrix, therefore we focus on reducing the indexing data used for representing the matrix.

We express each recurring pattern using a *blockcode*, which is a bit vector that encodes the nonzero micro-pattern using a bit for each nonzero. As an example, consider the top-left 4×4 block of the matrix given in Figure 2.1. The illustration of this block pattern, its blockcode representation, and the corresponding custom code as generated by PBR's code generator are depicted in Figure 4.1. For better readability, we show the non-vectorized version of the code. CSR needs more than six integers to represent the six nonzeros in this block. Existing blocking methods, such as the SPARSITY Framework [IYV04] or OSKI [VDY05], use less than 2 integers to represent the same block (not exactly 2 due to use of compressed representations), but they either fill in ten zeros (at a cost of increased memory bandwidth



0100110001010010

```
[ 1] __attribute__ ((noinline))
[ 2] void pbr_multiply_18994(double *y, double *x, double *Abase,
[ 3]                     int freq, BlockCoords_t *blockCoords, int pbrprfdist)
[ 4] {
[ 5]     int i;
[ 6]     register double y0;
[ 7]     register double y1;
[ 8]     register double y2;
[ 9]     register double y3;
[10]     for (i = 0; i < freq; i++) {
[11]         #ifdef DYNPREFETCH
[12]         __builtin_prefetch((void *) &Abase[0] + pbrprfdist, 0, 0);
[13]         #endif
[14]         int topleftrow = blockCoords[i].first;
[15]         int topleftcol = blockCoords[i].second;
[16]         y0 = y[topleftrow + 0];
[17]         y1 = y[topleftrow + 1];
[18]         y2 = y[topleftrow + 2];
[19]         y3 = y[topleftrow + 3];
[20]         y0 += Abase[0] * x[topleftcol + 1];
[21]         y1 += Abase[1] * x[topleftcol + 0];
[22]         y1 += Abase[2] * x[topleftcol + 1];
[23]         y2 += Abase[3] * x[topleftcol + 1];
[24]         y2 += Abase[4] * x[topleftcol + 3];
[25]         y3 += Abase[5] * x[topleftcol + 2];
[26]         y[topleftrow + 0] = y0;
[27]         y[topleftrow + 1] = y1;
[28]         y[topleftrow + 2] = y2;
[29]         y[topleftrow + 3] = y3;
[30]         Abase += 6;
[31]     }
[32]     return;
[33] }
```

Figure 4.1: An example 4×4 block pattern and corresponding PBR-generated code. Code is not vectorized for readability.

and CPU usage) to make this block dense, or simply disregard it because filling in that many zeros would not be beneficial. Being a blocking technique, PBR uses 2 integers to keep the block location in the matrix, namely *topleftrow* and *topleftcol* (lines 14 and 15), and requires no zero filling because PBR-generated codes include pattern-specific instructions (e.g., lines 16 to 29 in Figure 4.1) that perform the exact order of operations for a given pattern. In this way, PBR can take advantage of blocks that are not dense enough to be considered by other blocking techniques, and makes more efficient use of memory bandwidth and the CPU compared to the alternative of zero filling. A generated code cannot be used for other patterns, therefore a different code must be generated for each of the recurring patterns.

The pattern-specific instructions (lines from 16 to 29) include integer index constants integers to represent the block's nonzero structure. These indices appear as offsets in the machine instructions in the compiled binary, so they still consume memory bandwidth when fetched. However, this cost is easily compensated for, because once compiled codes are fetched from memory, they are repeatedly used to process *all* of the blocks that share this particular pattern before being evicted from the instruction cache.

4.1.2 Identification and Representation of Recurring Patterns

We use a simple analysis to identify repeating patterns. Given a block size $R \times C$, we divide a $m \times n$ matrix into a grid of $\lceil \frac{m}{R} \rceil \times \lceil \frac{n}{C} \rceil$ rectangular blocks and count how often each of the possible $2^{R \times C}$ patterns occurs. We do not consider block sizes larger than 8 because good coverage can be obtained from block sizes less than or equal to 8, and because the exponentially increasing number of possible patterns ($2^{R \times C}$) makes it less likely

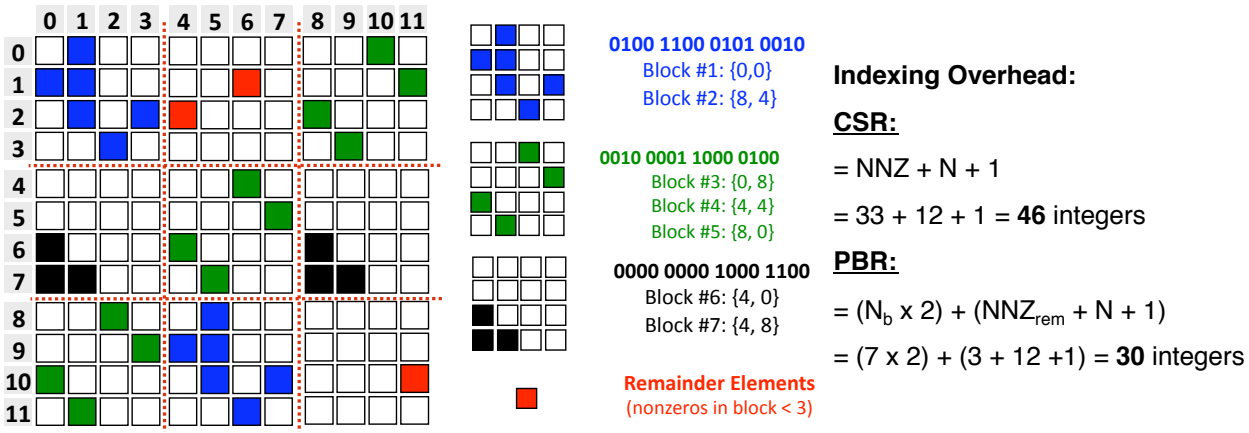


Figure 4.2: PBR representation of a square matrix of dimension $N = 12$ with $NNZ = 33$ nonzero elements. This matrix is composed of 3 recurring 4x4 block patterns and two remainder blocks with less than three nonzeros. $N_b = 7$ is the number of blocks with a shared pattern, $NNZ_{rem} = 3$ is the number of remainder nonzeros.

that individual patterns meet the cut-off criteria for larger block sizes. Our current PBR implementation considers only square blocksizes for simplicity.

We exclude two types of block patterns. First, we exclude blocks with patterns that include less than three nonzeros, because they yield little or no reduction in terms of reducing the index overhead in comparison to CSR, which already uses two or more integers for two nonzeros. Second, we exclude blocks whose patterns do not occur frequently enough to cover a significant number of nonzero elements, because the overhead of dispatching to the compiled code specific to the pattern may not be amortized otherwise. We empirically set this threshold as one thousand. Our analysis aggregates nonzeros that belong to excluded blocks in a remainder matrix, which is kept using conventional CSR representation.

Figure 4.2 depicts the PBR representation of the example 12×12 matrix, which was previously used in Figure 2.1 to explain the CSR and COO formats. For the sake of demonstration,

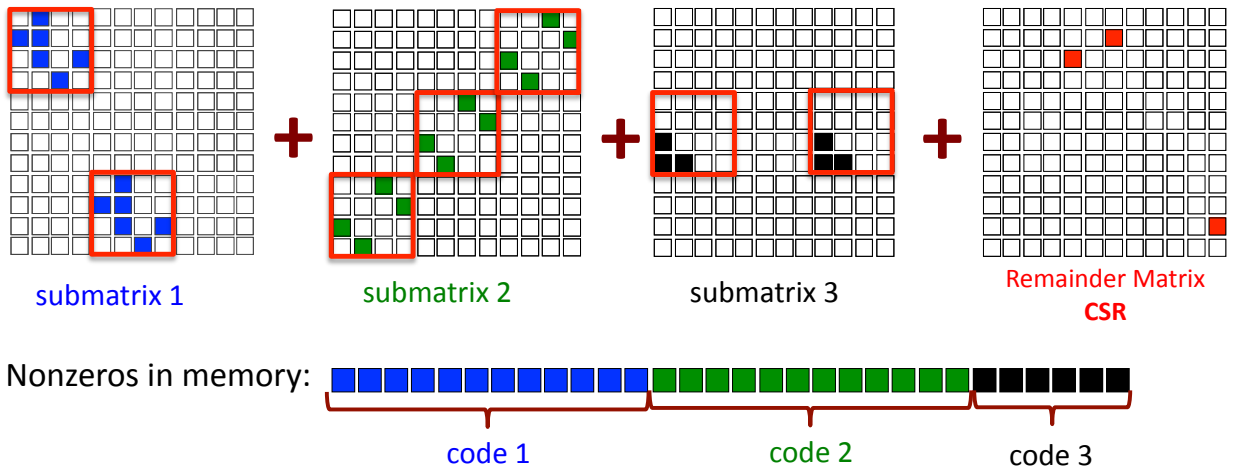


Figure 4.3: Splitting of the 12x12 matrix illustrated in Figure 4.2.

we ignore the one thousand overall nonzero coverage criterion (which is not met by such a small matrix), but we still exclude patterns with less than 3 nonzeros. An analysis that assumes a block size of 4×4 yields 3 recurring patterns with more than 2 nonzeros, which are highlighted as blue, green and black in the figure. These patterns cover 7 of the matrix's 9 blocks; the remaining 2 blocks containing less than 3 nonzeros (shown in red) are excluded from the PBR representation and their nonzeros are placed in a CSR remainder submatrix. In this example, the indexing overhead in a PBR representation is calculated as the sum of 2 integers per each valid block ($N_b \times 2$), plus the number of integers required by the CSR representation of the remainder matrix ($NNZ_{rem} + N + 1$). The total is 30 integers, as opposed to 46 integers required by the CSR representation as shown in Figure 2.1. This reduction corresponds to 35% savings in indexing overhead for this particular example.

We store coordinates for blocks of valid patterns in BCOO (Block Coordinate) format and place all blocks that share the same pattern consecutively in memory. This re-organization of nonzero values and indexing data expresses matrices as a collection of multiple submatrices,

similar to *matrix splitting* methods used in [Tol97, PH99, VM05]. The linearity of SMVM operations allows each of these submatrices to be processed separately. Figure 4.3 shows how the original matrix is split into a sum of submatrices and a remainder CSR matrix. This figure also illustrates how dispatched codes (code 1, code 2 and code 3) process nonzeros of corresponding patterns (blue, green and black) in a row to preserve their locality.

4.1.3 Block Size Selection and Nonzero Coverage

To benefit from PBR, the selected block size must yield sufficient nonzero coverage relative to the number of nonzeros placed in the remainder matrix. Depending on the structure of a matrix, different choices of R and C yield different degrees of coverage. In a preliminary study to investigate the impact of coverage on the PBR performance, we selected a set of 53 matrices that included all obtainable matrices from two previously studied sets by Im et al. [IYV04] and by Williams et al. [WOV+07]. We use these sets for two main reasons. First, these matrices are selected by their respective authors to cover a variety of problem domains, therefore they represent realistic usage scenarios. Second, the usage of sets with matrices selected by others avoids accidental bias, since PBR's efficiency is very sensitive to the nonzero structure of the selected matrices. Since many of these matrices are small relative to today's cache sizes, we also included some large matrices from the University of Florida repository [Dav] to reach a balanced matrix size distribution.

To find the optimal blocksize, we analyzed the matrices as described in Section 4.1.2 for all square block sizes $R = C = 2, 3, 4, \ldots, 8$, generated code for qualifying recurring patterns and measured the performance on two recent architectures: Intel Harpertown and

AMD Opteron. We summarize the results and properties of the matrices in Table 4.1. The "Opt. Blocksize" column lists the block size that yielded the best performance for the Intel Harpertown (HPT) and AMD Opteron (OPT) architectures. The columns "# of Patterns," "PBR Coverage," and "Index Savings" list the number of patterns, PBR coverage, and index overhead reduction relative to the CSR method for the optimal block size for the respective architecture. This table presents structural information for the block size that achieved the best performance, but not the performance results themselves, which are explored in detail later in Section 4.3.

These results reveal several interesting, and rather counter-intuitive, facts about PBR. Although a good coverage by PBR is a necessary condition for good performance, we find that optimal block sizes are not necessarily the ones that yield the largest nonzero coverage, nor the ones that provide the highest reduction in indexing overhead data. We also find that the optimal block sizes are architecture dependent. For example, on the Intel architecture, the best performance for the *saylr4* matrix was achieved using 6×6 blocks, covering 81.88% of the nonzeros to provide 43.8% savings in indexing overhead. On the AMD architecture, the best performance is achieved using 3×3 blocks, covering 85.87% of nonzeros to save 34.1% of the indexing overhead. An exhaustive search for the optimal block size, which we did for these particular experiments, would not be practical since it would require converting matrices for *all* candidate block sizes, followed by a performance benchmarking. As a solution, we describe a mechanism to *predict* optimal block sizes in Section 4.2.2, which utilizes a linear multi-regression model to approximate PBR performance on a given architecture.

Although we establish that nonzero coverage by PBR is not the sole factor to determine performance, it is still a prerequisite for performance improvements, therefore we now turn to

Table 4.1: Properties of matrices used in the evaluation of PBR. This table is based on the block size that yielded the best performance on the Intel Harpertown (HPT) and AMD Opteron (OPT).

			Opt. Blocksize		# of Patterns		PBR Cov.(%)		Index Sav.(%)	
Matrix Name	**N**	**NNZ**	**HPT**	**OPT**	**HPT**	**OPT**	**HPT**	**OPT**	**HPT**	**OPT**
orsreg_1	2205	14133	7	7	2	2	97.03	97.03	66.6	66.6
sherman3	5005	20033	5	5	2	2	77.34	77.34	42.9	42.9
sherman5	3312	20793	3	3	4	4	100	100	54.8	54.8
saylr4	3564	22316	6	3	5	2	81.88	85.87	43.8	34.1
mcfe	765	24382	4	4	1	1	37.8	37.8	32.1	32.1
lns_3937	3937	25407	8	2	5	3	40.86	38.38	29	14
lnsp3937	3937	25407	5	5	5	5	74.81	74.81	43.7	43.7
gemat11	4929	33185	2	2	1	1	13.38	13.38	5.82	5.82
bayer02	13935	63679	2	2	1	1	5.53	5.53	2.27	2.27
orani678	2529	90158	4	4	12	12	70.68	70.68	56.6	56.6
rdist1	4134	94408	6	6	8	8	71.02	71.02	63.4	63.4
bayer10	13436	94926	2	2	1	1	39.47	39.47	17.3	17.3
memplus	17758	126150	8	8	18	18	55.16	55.16	41.1	41.1
wang3	26064	177168	6	6	2	2	96.08	96.08	63	63
wang4	26068	177196	6	6	2	2	94.72	94.72	62.1	62.1
coater2	9540	207308	5	5	1	1	0.61	0.61	0.29	0.29
onetone2	36057	227628	3	8	18	3	45.72	3.5	20.1	2.15
lhr10	10672	232633	8	4	50	30	76.62	91.02	67.9	63
raefsky1	3242	294276	6	4	31	18	91.5	97.38	82	75.7
goodwin	7320	324784	4	4	44	44	75.42	75.42	52.6	52.6
pwt0	36519	326107	8	2	57	5	53.6	44.93	41.8	19.5
shyy161	76480	329762	4	5	6	6	86.33	88.89	45.1	52.2
vibrobox	12328	342828	2	2	1	1	6.42	6.42	3.1	3.1
af23560	23560	484256	8	8	7	7	100	100	87.4	87.4
finan512	74752	596992	6	2	74	4	79.66	31.39	44.9	10.8
scircuit	170998	958936	3	2	24	5	46.16	28.44	24	11.7
crystk02	13965	968583	3	6	1	10	99.91	99.64	76.6	88.8
rim	22560	1014951	4	6	111	220	88.95	87.62	61.2	72.9
ex11	16614	1096948	3	3	36	36	93.56	93.56	65	65
mac_econ_fwd500	206500	1273389	2	2	5	5	5.18	5.18	1.59	1.59
raefsky4	19779	1328611	3	3	28	28	95.53	95.53	66.9	66.9
bcsstk35	30237	1450163	3	3	17	17	98.56	98.56	74.2	74.2
raefsky3	21200	1488768	8	2	1	1	100	100	95.5	49.3
av41092	41092	1683902	4	6	59	236	84.81	82.61	55.6	60.1
venkat01	62424	1717792	8	2	15	1	100	100	85.8	48.2
crystk03	24696	1751178	3	3	1	1	99.95	99.95	76.7	76.7
qcd5_4	49152	1916928	3	3	1	1	100	100	75.8	75.8
mc2depi	525825	2100225	2	2	2	2	37.28	37.28	9.94	9.94
rma10	46835	2374001	3	3	33	33	95.84	95.84	63.7	63.7
nasasrb	54870	2677324	3	3	22	22	99.27	99.27	73.6	73.6
ct20stif	52329	2698463	6	3	227	47	84.1	93.39	72.7	63.8
webbase-1M	1000005	3105536	2	2	5	5	10.31	10.31	3.76	3.76
3dtube	45330	3213618	3	3	14	14	99.78	99.78	76.2	76.2
laminar_duct3D	67173	3833077	3	3	12	12	100	100	72.4	72.4
dense2	2000	4000000	6	3	3	3	100	100	94.4	77.7
cant	62451	4007383	3	3	19	19	99.97	99.97	74.9	74.9
pkustk04	55590	4218660	3	3	1	1	100	100	76.8	76.8
pdb1HYS	36417	4344765	6	3	15	1	99.87	99.98	90.7	77.1
Si34H36	97569	5156379	3	3	68	68	71.11	71.11	43	43
consph	83334	6010480	3	3	13	13	99.85	99.85	75.4	75.4
shipsec1	140874	7813404	6	3	1	1	100	100	92.8	76.4
pwtk	217918	11634424	3	3	34	34	97.65	97.65	71.6	71.6

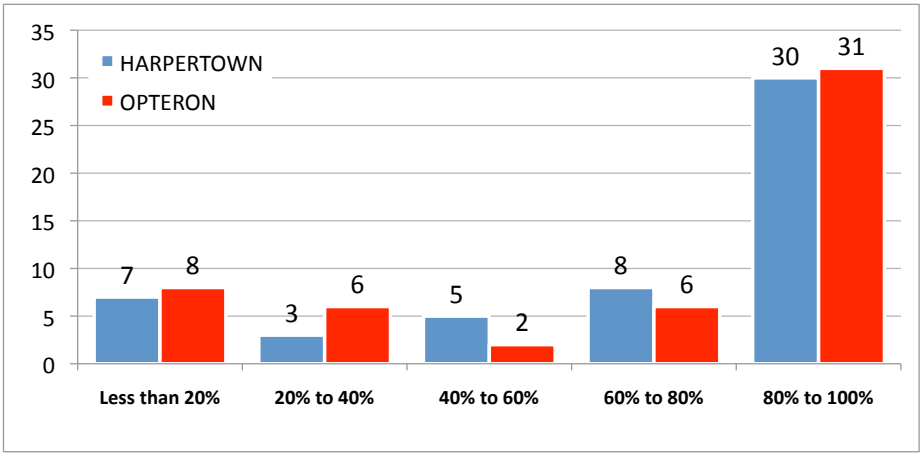


Figure 4.4: PBR nonzero coverage for matrix set shown in Table 4.1. Values are tabulated for the block size that yielded highest performance on each of the architectures shown.

Table 4.2: Frequency of the optimum block sizes that yield the maximum performance for the matrix set shown in Table 4.1.

	2×2	3×3	4×4	5×5	6×6	7×7	8×8
Harpertown	7	18	6	3	10	1	7
Opteron	13	20	5	4	6	1	3

Table 4.3: Distribution of the number of distinct block patterns. We consider matrices used in Table 4.1, aggregated for block sizes that yielded the highest performance.

	< 10	< 25	< 50	< 100	> 100
Harpertown	27	12	7	4	2
Opteron	33	9	7	1	2

this metric. Figure 4.4 provides a histogram that summarizes the achieved nonzero coverage, based on the block size that yielded the best performance on both architectures. For 56% (HPT) and 58% (OPT) of matrices, PBR encodes over 80% of nonzeros; coverage is 40% or less for only 19% (HPT) and 26% (OPT) of the matrices. Table 4.2 displays how often each block size yielded the best performance.

For this matrix set, our analysis appears to capture the underlying regularity in the nonzero structure of many types of matrices. Only matrices that are nearly random, such as those that model the relationship between hyperlinked web pages, tend to yield lower coverage.

Table 4.3 shows the distribution of the number of distinct patterns that satisfy our inclusion criteria. This table shows that the high degree of nonzero coverage we observe requires only a small subset of all theoretically possible $2^{R \times C}$ patterns (e.g., on both architectures only two matrices required more than 100 patterns), thus making it feasible to generate code for all qualifying patterns.

4.2 PBR Library

To facilitate pattern-based representation, we developed the *PBR library*, which provides a conversion routine that implements all necessary steps for matrix conversion. Users provide the input matrix in CSR format, thus making PBR drop-in compatible for any codes exploiting this widely used format. Our implementation can easily be modified to work with any other sparse format that provides matrix nonzeros in a row-wise order. We heavily optimized components of the PBR library to minimize the overhead required to use this method.

The conversion routine returns an opaque handle that is used in subsequent SMVM operations. The handle refers to an internal data structure that encapsulates matrix-specific information (such as dimension and sparsity), PBR-specific information (such as block size, number of recurring patters, nonzero coverage and names and paths of generated code and compiled object files), the data structures to hold the converted matrix itself (including nonzero values, block indices, list of patterns and their occurrence) and the remainder matrix in CSR format. An analyzed matrix structure can be saved to and restored from disk, allowing re-use of the analysis results for structurally identical matrices.

The library is callable from C code, but it is implemented in portable C++ using several high-performance container and utility classes provided by the STL [STL] and Boost [Boo] libraries. The custom SMVM kernels that implement the multiplication step are generated as C code and compiled with an optimizing C compiler.

PBR conversion involves the following steps: analyzing the matrix structure to find recurring patterns, prediction of the optimal block size, structural conversion, code generation, compilation (if necessary), and dynamic loading. The following sections describe these steps in detail and summarize the optimization techniques we used to reduce their runtime cost.

4.2.1 Matrix Analysis to Identify Recurring Patterns

The structure analysis determines which block patterns occur in the matrix and how often. To avoid reading the matrix index structure multiple times, we analyze all block sizes from 2×2 to 8×8 in a single pass. Figure 4.5 illustrates how matrix nonzeros are analyzed for

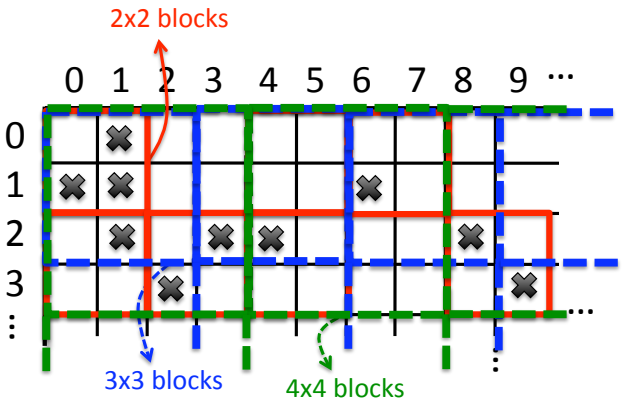


Figure 4.5: Single-pass analysis of block patterns: Each nonzero is read only once, and its contribution to all of the considered block sizes are simultaneously recorded into respective hash maps.

different block sizes at the same time. We divide the matrix into row stripes, each comprising $lcm(2, 3, ..., 8) = 840$ rows. This size ensures that row stripes contain an even number of block rows for each block size.

For each block size, we use a separate instance of Boost's unordered hash map to store the indices of those blocks that contain at least one nonzero element in the stripe. We cumulatively update these blocks as we process the input matrix index array. To provide for efficient iteration over the blocks, we additionally store their column indices in a linear STL vector. We optimized this step further by using a custom memory allocator based on GNU's obstack implementation, which allows for fast allocation and deallocation of these temporary containers, which are reset for each row-stripe. After all nonzeros within a stripe are accounted for, the encountered patterns and their frequency are tabulated in a separate hash map that is keyed by bitsets encoding each pattern. Since all lookups are performed with hash maps that provide $O(1)$ average lookup time, the asymptotic time complexity of this analysis step is linear in the number of nonzeros contained in the matrix. Some

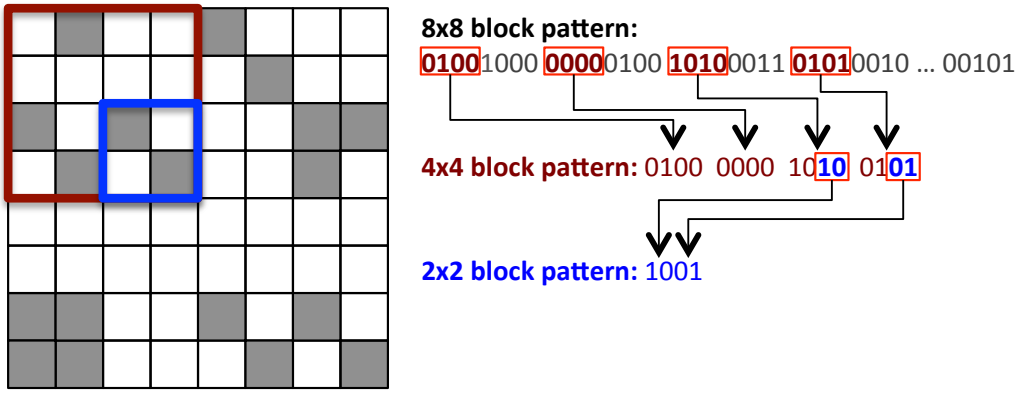


Figure 4.6: Derivation of statistics for small patterns using their parent patterns. 4×4 and 2×2 block patterns can be derived from the pattern of their parent 8×8 block, using simple bit operations.

operations, such as the initialization and teardown of the hash maps containing the indices of the nonzeros within each stripe, are performed once for each stripe. These operations incur an asymptotic complexity that is linear in the number of matrix rows N.

To further optimize the matrix analysis step, our implementation collects pattern statistics only for blocks larger than 4×4, which we refer to as *parent blocks*. The statistics for blocks of size 4×4 and 2×2 are derived from their parent 8×8 pattern by examining subsets of bit patterns, as illustrated in Figure 4.6. Similarly, the statistics for 3×3 patterns is derived from their encapsulating 6×6 blocks. This greatly reduces the cost of analysis because each nonzero block triggers a search for its pattern in the hashmap, thus deriving statistics from a 8×8 block potentially eliminates up to 16 hash map searches for 2×2 blocks, or 4 searches for 4×4 blocks. The analysis step provides only the number and kind of distinct recurring patterns for each block size, excluding patterns that do not meet the cutoff criteria described in Section 4.1.2. No matrix conversion or data re-arrangement is performed yet, since it requires detection of the optimal block size first, which we explore in the next section.

4.2.2 Block Size Selection

The preliminary results we summarized in Table 4.1 reveal that the optimal block size is not solely dependent on the nonzero coverage and indexing reduction. It is possible that two block sizes with similar nonzero coverage and indexing reduction can lead to completely different access patterns on arrays y and x, due to the differences in the micro-structures of patterns they include. For sufficiently large matrices, we expect that PBR SMVM's runtime is very sensitive to these memory access patterns.

With these considerations in mind, we developed a simple multiple linear regression model, which is composed of three variables that capture the memory accesses patterns performed during the multiplication:

- The number of bytes fetched from memory while accessing the covered nonzero values, computed as the product of $coverage \times NNZ \times$ `sizeof(double)`, plus the number of bytes comprising the block index array, which is $\sum_{i \in P} 2 \times freq(P_i) \times$ `sizeof(int)` for the set of qualifying patterns P.
- An approximation of the number of accesses to the x and y vectors while multiplying the submatrix for each included block pattern. Since each submatrix may cover all rows and columns, we approximate this number as the product of the matrix dimension and the number of patterns: $N \times |P|$.
- The number of writes to y, which can be derived for each block pattern from its structure. This variable is needed because unlike CSR, which writes each element of y exactly once, PBR may read and write each y element multiple times.

The pattern-specific values of these input variables are computed from the statistics collected for each of the block sizes during the matrix analysis step.

Our model must also predict the performance of the remainder matrix, which is kept in CSR format. We modeled CSR's performance with a separate multiple regression model that is based on matrix size and number of nonzeros. Finally, the block size that achieves the best predicted runtime, which is calculated as the sum of estimated runtimes for PBR and remainder CSR using their respective models, is selected for generating the PBR structure in the following step.

Our evaluation in Section 4.3.5 discusses the parameter estimates for these models for the training set and architectures we use, and shows that this model can select an optimal or near-optimal block size for the majority of the cases we considered.

4.2.3 Structure Conversion

The structure conversion step records the block indices and rearranges the nonzero values in PBR format for the selected block size. We keep the (row, col) block indices and the matrix nonzero values in two one-dimensional contiguous arrays, similar to the BCOO format. As depicted in Figure 4.3, the nonzeros and indices of all blocks belonging to the same pattern are stored in contiguous slices of these arrays, which guarantees spatial locality in the inner loop of each kernel. Because the number of blocks for each recurring pattern has been determined during the analysis step, the start and end locations of these slices can be precomputed in a single pass over all patterns. Thus, nonzeros and block indices can be copied directly to their

target destinations by maintaining pointers to the current nonzero and block index offsets within each pattern's slices.

4.2.4 Code Generation

The code generation step generates custom C functions for all qualifying patterns, stores them to disk, and invokes the C compiler to create shared object modules (.so for Linux or .dylib for MacOS). Each shared object module is dynamically linked into the process's address space. A pointer to the C function it contains is added to an array of function pointers. The outer loop of the SMVM routine iterates over this array and invokes the generated block multiplication functions via indirection.

The shared object modules are created on demand and stored in a code cache on disk. Since object modules are specific to only the block pattern and size, they can be reused both across multiple uses of the PBR library on the same matrix, as well as across multiple matrices that contain the same pattern. Optionally, the code cache can be primed by generating and compiling all possible patterns for block sizes 2×2, 3×3, and 4×4, which would pay for these code generation costs upfront and avoid further cost when these block sizes are chosen. Priming the cache for larger block sizes is infeasible due to the exponential growth of the number of possible patterns with block size ($2^{R \times C}$).

Our code cache is designed to hold modules for different target architectures and with different optimization options (e.g., SSE) simultaneously, thus allowing it to be shared by multiple machines on a network. If needed, the size of the code cache repository could be managed by

a periodically scheduled cronjob that deletes files that have not been accessed for a certain time period, which is similar to the strategy used by many Unix installations to purge stale files from the `/tmp` directory.

4.2.5 Explicit Software Prefetching

We optimized our baseline implementation by applying explicit prefetching for the matrix elements. Although the streaming pattern of these accesses would ordinarily prevent gains from prefetching, prior work on SMVM optimizations has shown [WOV+07, WOV+09] that explicit prefetching is beneficial on SSE-enabled x86-based architectures, which constitute 91% of the systems in the June 2010 Top-500 list [MSDS10]. Prefetching can place data directly into the L1 cache and label cache entries with the correct temporal locality, thus allowing eviction in preference to other data such as elements of the x and y vectors, which may be reused. Explicit prefetching is implemented via the GCC `__builtin_prefetch()` compiler intrinsic (line 12 in Figure 4.1), which results in the emission of `prefetch*` SSE instructions. The current PBR implementation performs an exhaustive search to obtain the best prefetch distance at runtime.

4.2.6 Vectorization

We adapted our code generator to emit vectorized codes that exploit SSE SIMD intrinsics. These SSE instruction set extensions allow simultaneous operations on a vector of two double values via the `__mm128d` data type, which is mapped to a 128 bit SSE register. For simplicity,

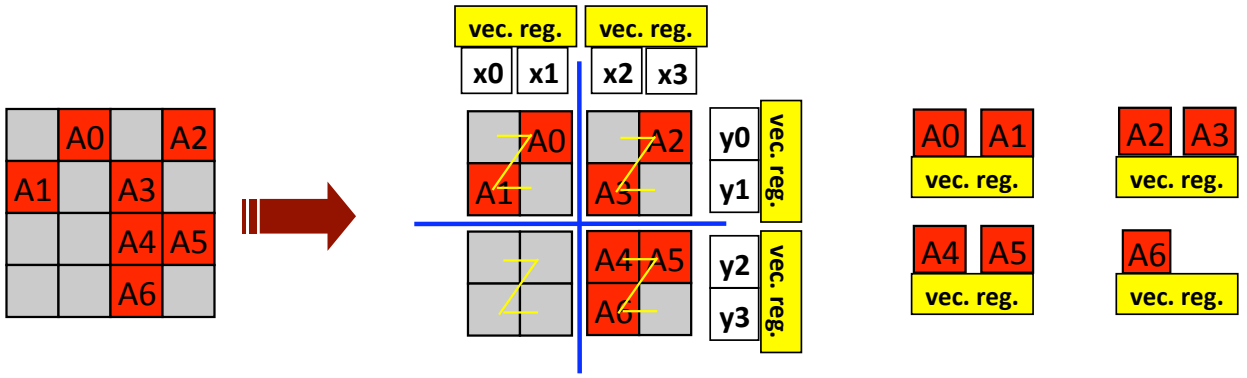


Figure 4.7: Assignment of x, y and matrix elements to 128-bit vector registers. Matrix elements are stored in zigzag order in memory.

we vectorize only blocks with even sizes. Each $R \times C$ block is divided into $(R/2 \times C/2)$ subblocks of size 2×2. We allocate $R/2$ and $C/2$ vector variables for each (x_i, x_{i+1}) and (y_j, y_{j+1}) pair. For aa matrix nonzeros, we allocate 1 vector variable if a subblock has 1 or 2 nonzeros, and 2 variables if it has 3 or 4 nonzero elements, independent of where those nonzeros are located within the subblock. To minimize the number of shuffle operations that are required if aa elements are not located in the order in which they are needed, we store the aa elements in zigzag order in memory. In this way, nonzeros within a 2×2 subblock are located consecutively in memory. Figure 4.7 illustrates the assignment of elements of arrays x and y, as well as the matrix nonzeros, to 128-bit vector registers for an example 4×4 block. Figure 4.8 depicts an example for generated vector code, for the pattern $\frac{\circ|\bullet}{\bullet|\circ}$. The comment lines in this code are also placed by the code generator to improve readability.

We maximize the use of 128-bit aligned load instructions by placing the beginning address of each block on a 16-byte boundary, using padding when necessary, as required by this SSE instruction. We implemented two different alignment strategies within a block. Our first strategy uses aligned loads only when it is known that the elements to be loaded have the

0110

```
[ 1] /*
[ 2]  * SSE custom kernel for 2x2 block code
[ 3] [ ][x]
[ 4] [x][ ]
[ 5] */
[ 6] __attribute__ ((noinline)) void
[ 7] pbr_multiply_6(
[ 8]     real_t *y, real_t *x, real_t *Abase,
[ 9]     int nblocks, struct _nnzblock *nnzblock)
[10] {
[11]   int i;
[12]   for (i = 0; i < nblocks; i++) {
[13]     int topleftrow = nnzblock[i].topR;
[14]     int topleftcol = nnzblock[i].topC;
[15]     // _r76 = [_a1, _a0]
[16]     __m128d _r76 = _mm_load_pd(Abase);
[17]     // _r75 = [_y1, _y0]
[18]     __m128d _r75 = _mm_load_pd(y + topleftrow);
[19]     // _r73 = [_x1, _x0]
[20]     __m128d _r73 = _mm_load_pd(x + topleftcol);
[21]     // _r77 = [_x0, _x1]
[22]     __m128d _r77 = _mm_shuffle_pd(_r73, _r73, 1);
[23]     // _r78 = [_a1 * _x0, _a0 * _x1]
[24]     __m128d _r78 = _mm_mul_pd(_r76, _r77);
[25]     // _r79 = [_y1 + _a1 * _x0, _y0 + _a0 * _x1]
[26]     __m128d _r79 = _mm_add_pd(_r75, _r78);
[27]     _mm_store_pd(y + topleftrow + 0, _r79);
[28]     Abase += 2;
[29]   }
[30] }
```

Figure 4.8: PBR-generated SSE custom kernel for the example 2×2 pattern $\begin{smallmatrix} \circ & \bullet \\ \bullet & \circ \end{smallmatrix}$.

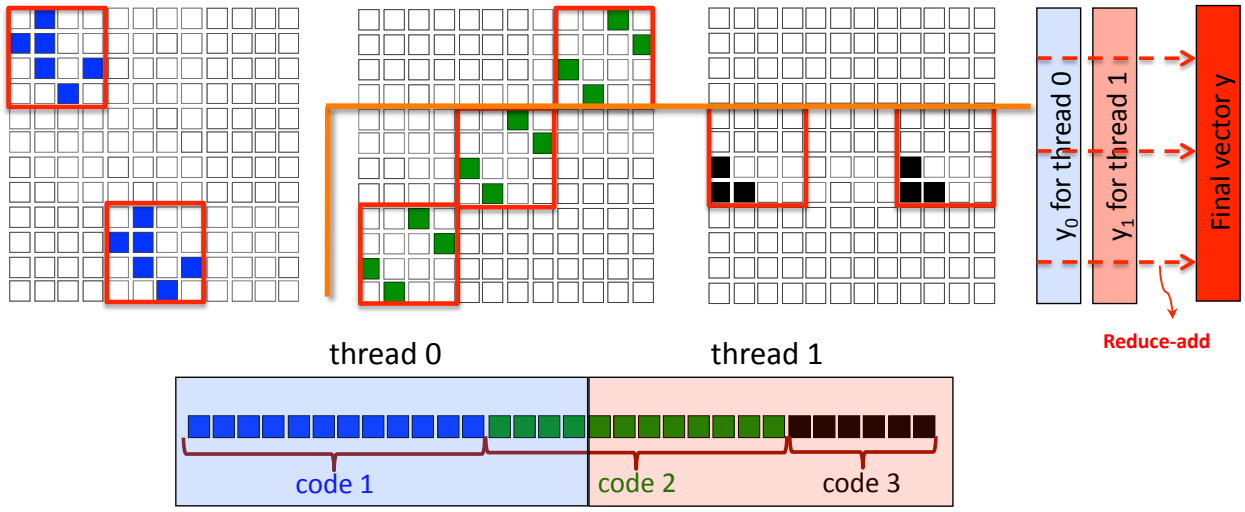


Figure 4.9: Block-level partitioning of PBR. Both threads need to maintain a local copy of the vector y, and a final reduce-add operation is needed to obtain the final vector y.

correct 16-byte alignment, based on the alignment of the initial nonzero within the block. If the number of nonzero elements is odd (1 or 3), we use 64-bit double load instructions and set the unused element within the vector to zero instead. Our second strategy stores an additional zero in memory whenever the number of nonzeros within a 2×2 subblock is odd, thus allowing the use of aligned load instructions throughout. Our performance evaluation reports the results for the strategy that yielded the best performance.

4.2.7 Parallelization

PBR-SMVM operations contain inherent data parallelism, since all floating point multiply operations are independent of each other. Moreover, the post analysis we perform to identify recurring patterns reveals the number of blocks for each pattern, as well as their distribution in the matrix, allowing us to finely balance the workload across threads.

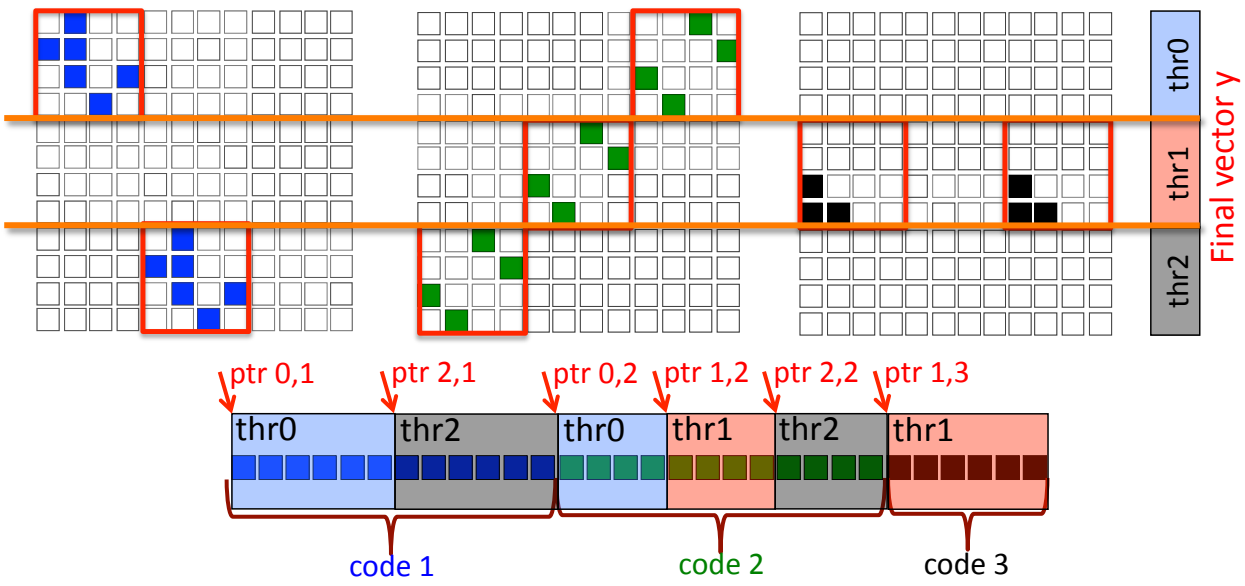


Figure 4.10: Row-wise partitioning of PBR. This approach does not need local y vectors and a reduce-add step, but requires identification of pointers to express data regions inside the contiguous nonzero and index arrays for each thread.

We implemented and evaluated PBR-SMVM in parallel using two different approaches. The first approach achieves parallelism by partitioning the contiguous arrays of nonzeros in the PBR submatrices (e.g, the nonzero array in Figure 4.3). The second approach uses a row-wise partitioning of the matrix and treats each row-stripe as if they were separate rectangular matrices.

Our evaluation of the former approach [BBR09] revealed significant performance and scalability problems, especially as the number of cores grows. Figure 4.9 depicts an illustration of this approach, which distributes the nonzeros to 2 threads at a block-level granularity. Since blocks can be located anywhere vertically in the matrix, each thread can potentially update any element in the entire y array, leading way to possible write conflicts. One solution to prevent write conflicts is to use locks, but they incur significant cost and sequentialize the

operation. A more efficient approach is to maintain a local y array for each thread, which requires an additional reduce-add step at the end of operations to obtain the final vector y. We found that neither of these solutions can achieve efficient parallelization, however.

As a solution, we row-partition the matrix, which is a common approach also used for parallelization of many other methods, including CSR. We set the boundaries of each row-stripe such that each thread is assigned approximately the same number of nonzeros across all patterns. Each partition is handled by a thread as if it were a separate PBR matrix. As a possible drawback, this approach does not not guarantee that each pattern covers at least 1000 nonzeros within a partition, which is one of PBR's coverage criteria as described in Section 4.1.2. In our experience, however, this drawback does not cause significant performance loss. Figure 4.10 depicts an illustration of row-partitioning a PBR matrix for 3 threads. Threads process only the blocks inside the stripe they are assigned to. Since PBR reordering places all blocks of the same pattern contiguously in memory, we keep an array of pointers to identify the data ranges each thread is responsible for. In Figure 4.10, for example, thread #0 accesses the first blue block (ptr 0,1), then skips the second blue block that is in another stripe, then accesses the first green block (ptr 0,2). We use the offsets from the beginning of arrays in the current implementation, rather than memory addresses, to be able to store and reuse this partitioning information.

We record offsets that mark data regions for each thread during the matrix analysis step for an assumed number of threads, which is set to 128 in current implementation. If the granularity level of a run is coarser than the assumed level, e.g., if only 8 threads are used instead of 128 threads, then threads take over multiple stripes (e.g., 16 for each of the 8 threads) to achieve efficient parallelism. On the other hand, the opposite it not possible, i.e.,

we cannot run the code with a finer granularity than what the matrix is analyzed for. In this case, the matrix needs to be analyzed once again to set pointers for the finer granularity levels, which incurs a low linear cost as described in Section 4.2.1.

4.3 Evaluation

Our evaluation contains multiple parts. First, we demonstrate how PBR-based SMVM shortens time to solution, or overall runtime, in both sequential and parallel environments, relative to readily available alternatives. Second, we demonstrate the relative impact of the prefetching and vectorization optimizations we applied to PBR. Finally, we evaluate the runtime costs of PBR relative to its benefits. For all experiments, we use the matrix set in Table 4.1, which is introduced in Section 4.1.3.

4.3.1 Methodology

We used two x86-based architectures for our evaluation. The first architecture is a 2.8Ghz 2-socket quadcore (8-core total) Intel Xeon Harpertown 5400 with 12 MB L2 cache per socket (each core pair sharing a 6MB cache,) and 8GB of RAM. The second architecture is a 2.5 GHz 2-socket quadcode (8-core total) AMD Opteron 2380 with 512KB L2 and 128KB L1 caches per core, a 6MB shared L3 cache per socket, and 8GB RAM per socket (NUMA, 16GB total). Diagrams of these architectures, as generated by the Portable Hardware Locality (hwloc) library [BCOM+10], are depicted in Figures 4.11 and 4.12. We use the GCC compiler version 4.1.2 on the Harpertown and version 4.3.2 on the Opteron with optimization flags:

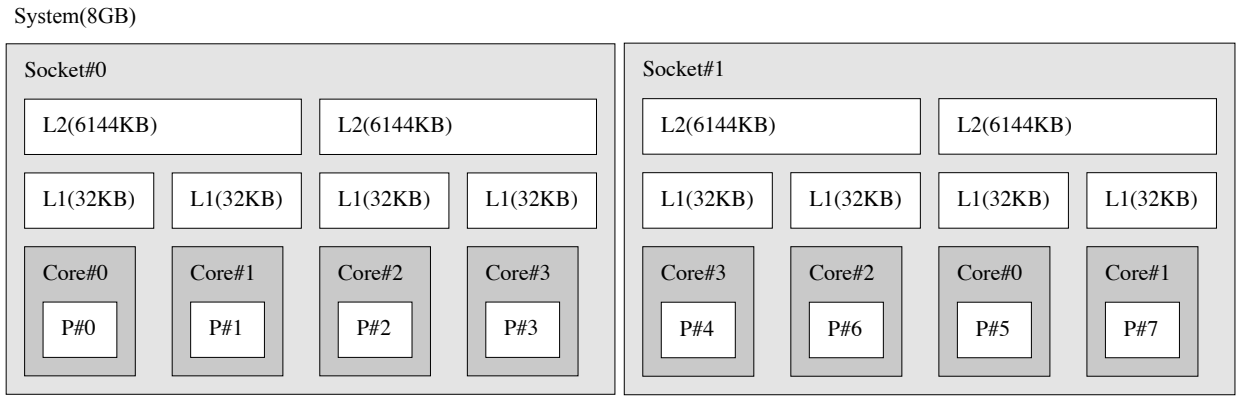


Figure 4.11: Illustration of the Intel Harpertown architecture. We used the Portable Hardware Locality (hwloc) library to draw this figure.

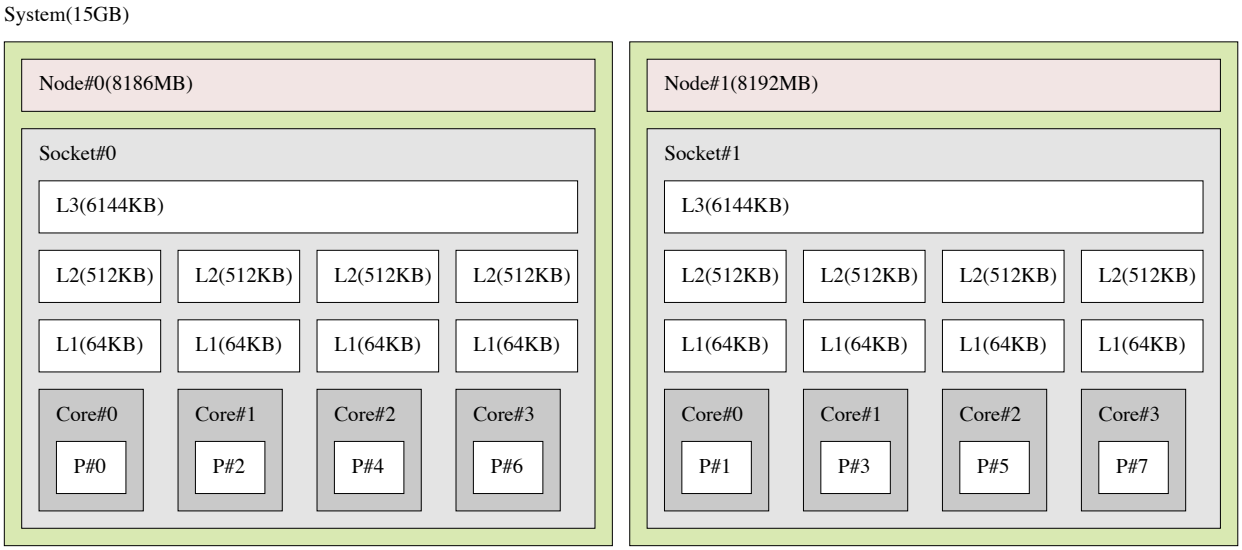


Figure 4.12: Illustration of the AMD Opteron architecture. We used the Portable Hardware Locality (hwloc) library to draw this figure. This system differs from the Intel Harpertown (Figure 4.11) with its NUMA architecture, additional shared L3 caches and per-core L2 caches.

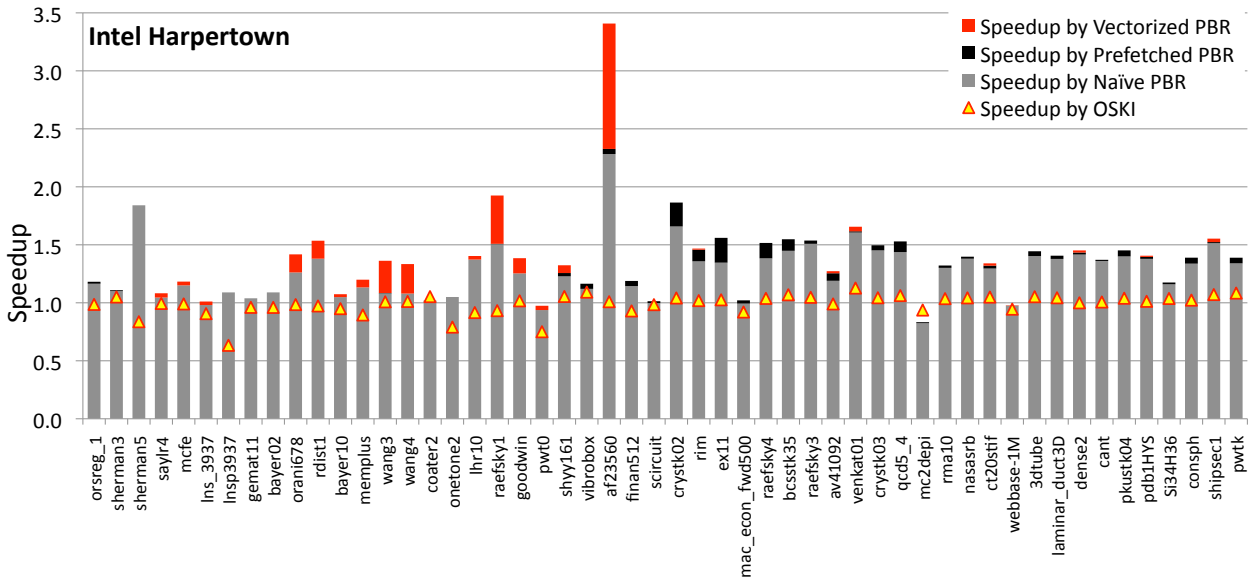


Figure 4.13: Sequential PBR speedup on the Intel Harpertown. This figure compares naïve CSR performance to (1) plain PBR, (2) prefetched PBR, (3) vectorized PBR and (4) OSKI.

`-O2, -funroll-loops, -mfpmath=sse, -msse, -mtune`. We also specified `-march` using proper values for each architecture.

4.3.2 Sequential Performance

Figures 4.13 and 4.14 compare the performance of unoptimized (plain) sequential PBR to the performance of PBR with prefetching and PBR with vectorization on Harpertown and Opteron, respectively. Speedup results are shown relative to naïve CSR performance.

For matrices with 80% or more nonzero coverage, which account for at least 30 of the 53 matrices on both architectures, PBR provides a maximum speedup of 3.41 with an average of 1.53 on the Harpertown and a maximum speedup of 2.32 with an average of 1.64 on the

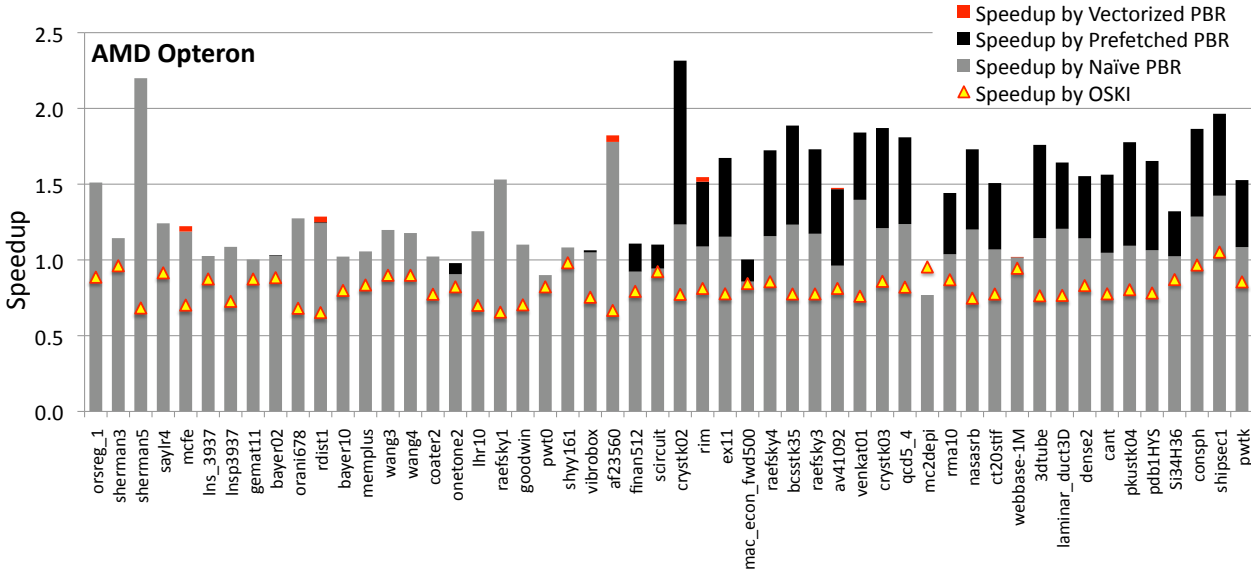


Figure 4.14: Sequential PBR speedup on the AMD Opteron. This figure compares naïve CSR performance to (1) plain PBR, (2) prefetched PBR, (3) vectorized PBR and (4) OSKI.

Opteron.

The relative contribution of prefetching and vectorization is shown as zero if prefetching or vectorization did not improve performance. For a few matrices, these optimizations yielded slight slowdown. On Harpertown, small matrices benefit particularly from vectorization, whereas the benefit of prefetching is more pronounced on the Opteron. This figure might lead to an incorrect deduction such that vectorization does not provide benefit on the Opteron architecture. However, although not shown in the figure, vectorization does improve the performance by 25% on average when applied *without* prefetching.

These charts also contain the results we obtained using OSKI 1.0.1h [VDY05], a widely used autotuning sparse kernel library. We used OSKI's aggressive tuning option without providing structural hints. OSKI is consistently slower than PBR, and even provides little

or no speedup compared to naïve CSR on the architectures considered, which is consistent with earlier results obtained by others [WOV+07].

4.3.3 Parallel Performance

We built a thread pool implementation on top of Linux's PThreads API. To avoid the potentially random placement of threads onto cores by the OS scheduler, we used the *Portable Linux Processor Affinity (PLPA)* interface from the OpenMPI project [GSB+06] to explicitly set a CPU affinity for each thread, which forces a fixed mapping of threads to cores. When using 2 threads, we placed threads on neighboring cores that share the same L2 cache on Harpertown and the same L3 cache on Opteron. When using 4 threads, we placed them onto the same socket, thus sharing access to a single front-side memory bus.

We compared our parallel implementation of PBR, which uses a row-partitioning strategy as described in Section 4.2.7, to a row-partitioning parallel implementation of naïve CSR. Both CSR and PBR use the same thread pool implementation.

Figure 4.15 shows the overall runtime of parallelized PBR relative to parallelized CSR for 2, 4, and 8 threads on Intel Harpertown (top) and AMD Opteron (bottom). For this discussion, we consider only the subset of matrices that provided more than 80% coverage and that yielded at least 1.1× sequential speedup. A value greater than 1.0 indicates that the use of PBR is beneficial for a given number of threads. These results show that parallel PBR significantly outperforms parallel CSR, providing a maximum speedup of 3.8× on Harpertown and 5.0× on Opteron using 8 cores.

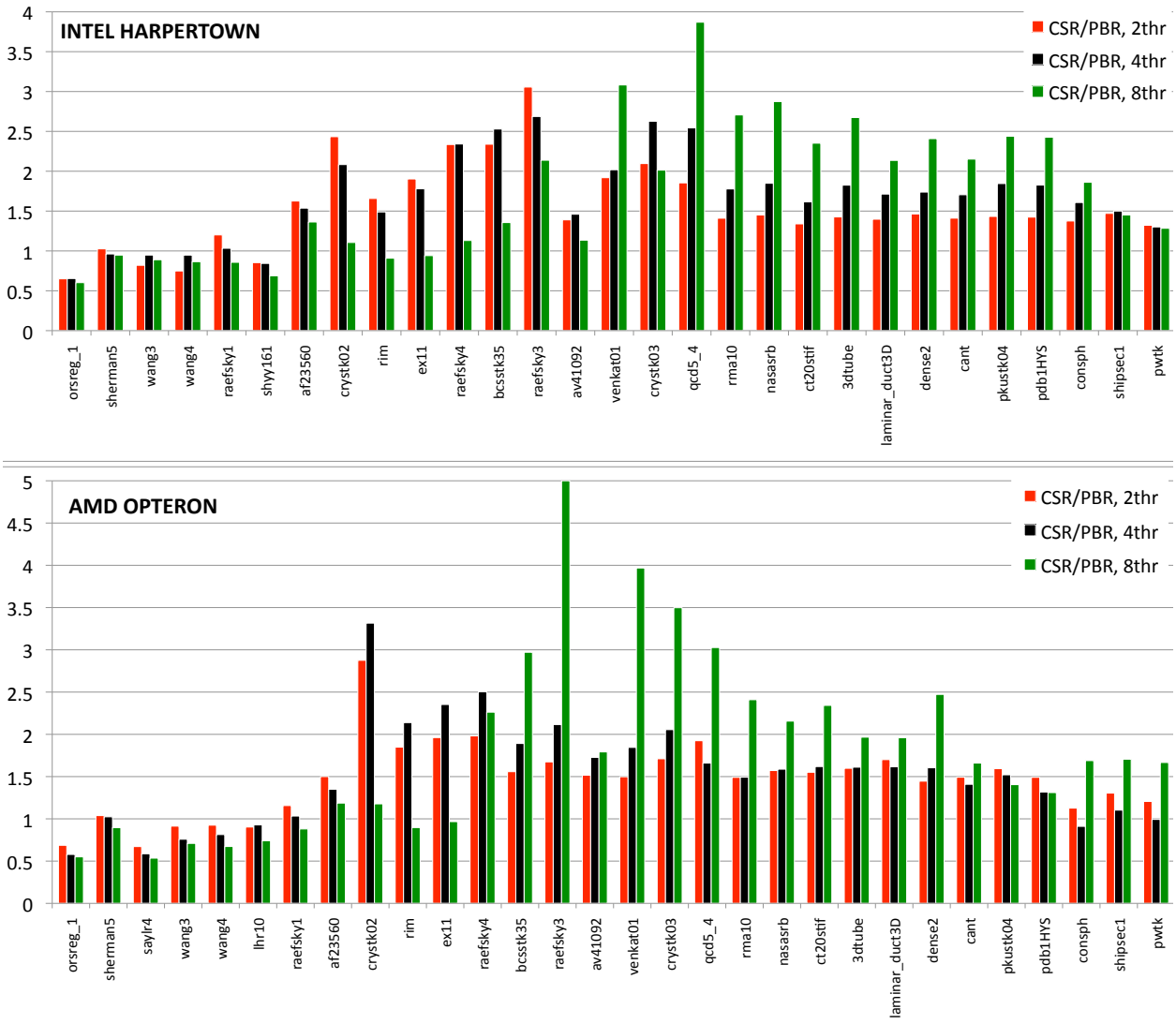


Figure 4.15: Parallel PBR performance relative to the parallel CSR. We run both methods using 2, 4, and 8 threads on Harpertown (top) and Opteron (bottom). Only matrices with more than 80% coverage and sequential PBR speedup greater than 1.1 are included. Matrices are sorted by their size, starting from the smallest on the left.

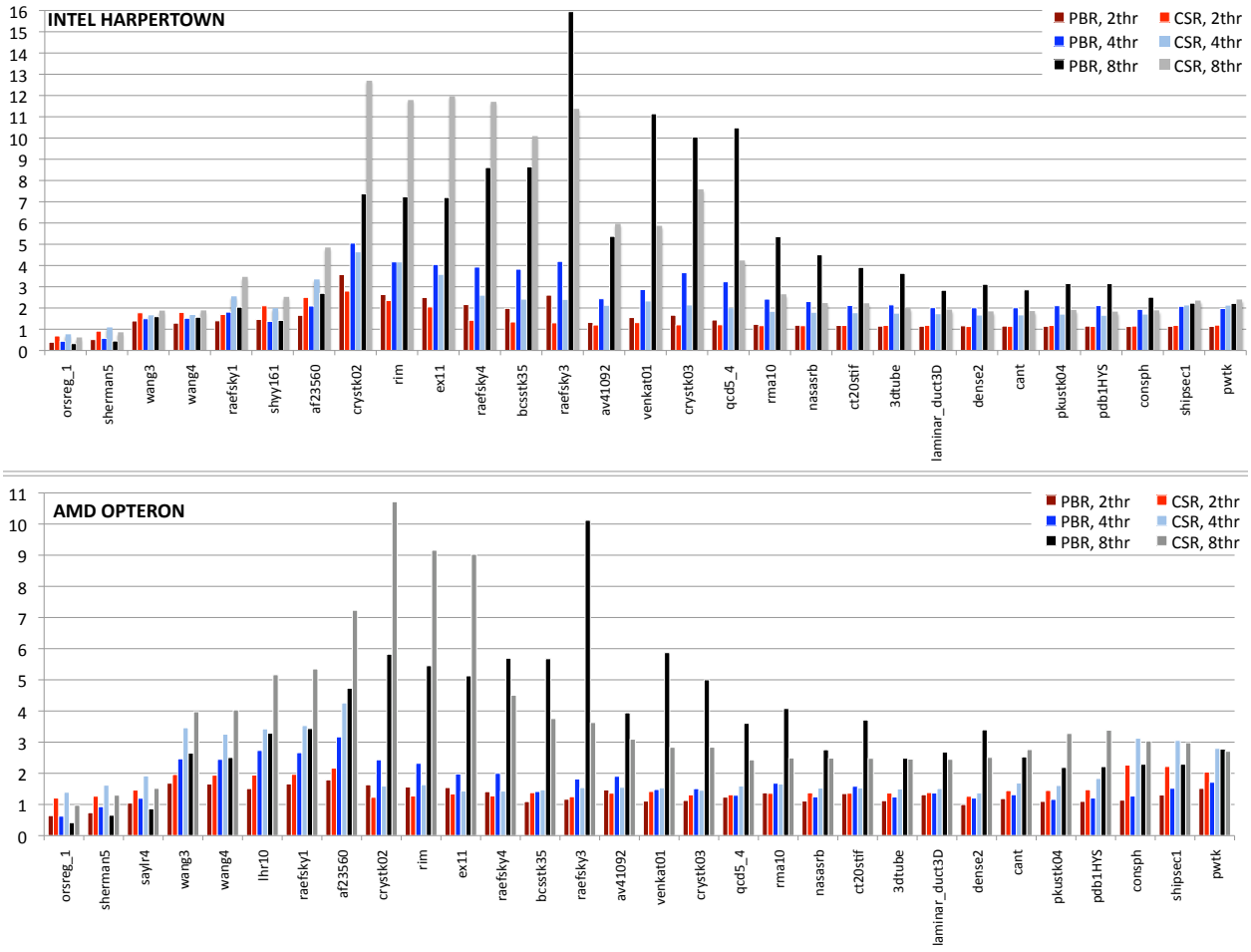


Figure 4.16: Parallel PBR and CSR speedup. We run both methods using 2, 4, and 8 threads on Harpertown (top) and Opteron (bottom). Performance data is shown for large matrices with more than 80% coverage and sequential PBR speedup greater than 1.1.

Table 4.4: Average of parallel speedups given in Figure 4.16.

Harpertown	**2 core**	**4 core**	**8 core**
PBR	1.29	1.70	3.61
CSR	1.54	2.04	3.82

Opteron	**2 core**	**4 core**	**8 core**
PBR	1.50	2.48	4.88
CSR	1.44	2.16	4.64

Figure 4.16 depicts parallel speedup by CSR and PBR relative to their respective sequential implementation for the same matrix set used in Figure 4.15. The averages of the speedups in this figure are summarized in Table 4.4. Both methods exhibit superlinear speedup with 8 threads for medium sized matrices due to doubled memory bandwidth and cache capacity from using two sockets. These results are in line with CSR superlinear speedups that were previously observed on similar architectures by others [WOV+07, WOV+09].

4.3.4 Overhead Analysis

As with any optimization method, PBR's overhead must be evaluated relative to the benefits it provides. We relate PBR's overhead to its performance benefits over the CSR method in the context of iterative solvers. The *break-even* column (*BEP*) in Table 4.5 shows how many iterations would be needed to compensate for PBR's overhead. Since the break-even point depends on the achieved speedup for a given matrix, we also report as a second point of comparison how many CSR iterations could be performed in the time it takes to analyze the matrix *(Analysis)*, convert it to PBR *(Struc.)* and link to kernel object files *(dynLink)* in the same table. These numbers provide a lower bound below which PBR cannot be amortized.

Table 4.5: PBR overhead for matrices that achieved at least 1.1× speedup. Matrices are sorted by NNZ. This table depicts cost of block size detection (Analysis), conversion of the matrix (Struc.) and dynamically linking to kernel object files (dynLink), expressed in number of CSR iterations for the given matrix and architecture. BEP denotes the break even point, which is the number of iterations needed to compensate PBR overhead. BBS is the block size that yielded the best performance.

	Intel Harpertown					AMD Opteron				
Matrix	**BBS**	**Analysis**	**Struc.**	**dynLink**	**BEP**	**BBS**	**Analysis**	**Struc.**	**dynLink**	**BEP**
orsreg_1	7	108	59.5	24.7	1252	7	128	65.8	3.2	583
sherman3	5	86.5	54.9	12.0	1567	5	115	67.3	2.02	1465
sherman5	3	85.8	60.2	23.4	370	3	115	70.3	3.79	347
saylr4	-	-	-	-	-	3	115	80.9	1.96	1015
mcfe	4	340	112	9.3	2979	4	371	114	1.87	2665
orani678	4	201	92.3	24	1075	4	215	93.6	2.82	1448
rdist1	6	84.7	77.1	14.2	503	6	107	82.3	1.94	8598
memplus	8	184	74.8	16.0	1657	-	-	-	-	
wang3	6	85	59.8	2.11	552	6	103	65.4	0.26	101
wang4	6	86.5	61.5	2.12	599	6	104	66	0.28	1127
lhr10	8	89	70.2	34	670	4	114	84.2	2.67	1268
raefsky1	6	84	75.6	28.2	391	4	99	79.6	1.3	5183
goodwin	4	95.3	91.9	25.2	768	4	117	95	2.79	234
shyy161	4	67.1	61.9	2.27	536	-	-	-	-	
vibrobox	2	138	178	0.612	2235	-	-	-	-	
af23560	8	54.8	39.5	1.99	136	8	88.4	55.9	0.29	3200
finan512	6	63.2	50.9	8.92	776	2	94	111	0.1	210
scircuit	-	-	-	-	-	2	135	82.2	0.05	2373
crystk02	3	27.8	27.9	0.118	120	6	57.9	42.1	0.14	1751
rim	4	35.5	36.8	7.9	250	6	67.7	52.1	4.57	351
ex11	3	39.8	35.3	2.75	216	3	74.4	57.6	0.39	329
raefsky4	3	29.7	33	1.44	188	3	62	55.7	0.27	280
bcsstk35	3	28.9	27.4	0.816	161	3	61.7	48.9	0.15	235
raefsky3	8	24.3	20.7	0.0737	128	2	55.5	63.5	0.02	282
av41092	4	55.8	44.6	3.05	483	6	93.6	67.6	3.12	508
venkat01	8	29.8	21.4	0.737	131	2	65.2	60.6	0.02	274
crystk03	3	27.6	27.9	0.0643	167	3	58.1	48.9	0.02	230
qcd5_4	3	29.4	27.5	0.0564	164	3	63.4	49.4	0.02	252
rma10	3	37.4	33.7	1.14	296	3	72.8	56	0.15	421
nasasrb	3	28.1	28.6	0.692	202	3	59.2	50.3	0.27	260
ct20stif	6	39.8	29.3	7.99	303	3	77.2	57	0.24	400
3dtube	3	27.5	31.4	0.3	192	3	60.1	59.6	0.06	277
laminarduct3D	3	28.1	30.8	0.228	204	3	60.4	53	0.04	289
dense2	6	23.7	21.5	0.0618	145	3	52.8	52.7	0.01	296
cant	3	28.8	30	0.34	218	3	60.1	50.9	0.07	308
pkustk04	3	29.9	29.1	0.0209	189	3	62.1	52.6	0.01	262
pdb1HYS	6	29.9	23.8	0.321	186	3	61	48.5	0.01	277
Si34H36	3	47.1	54.1	0.948	685	3	76.6	77.7	0.17	634
consph	3	28.2	28.1	0.199	201	3	49.6	41.7	0.02	196
shipsec1	6	26.1	21.2	0.0138	132	3	48.5	40	0.003	180
pwtk	3	24.3	27.9	0.263	186	3	55.3	50.4	0.04	306

Table 4.6: Multi-regression parameters for the matrix analysis costs. This model approximates the 'Analysis' column of Table 4.5 for the Harpertown architecture.

	Model Parameters		
	Intercept	**N**	**NNZ**
Parameters:	0.032	6.15E-07	9.07E-08
Std Error:	0.015	7.98E-08	5.47E-09
	Adjusted $R^2 = 0.90$		

The *best block size (BBS)* column depicts the block sizes that yielded the best performance. This table excludes the code generation/compilation cost, which is shown separately in Table 4.8. Matrices in this table are sorted by their size, with smaller matrices in the top half. Since small matrices benefit less from PBR, their break-even point is generally higher. The break-even point reduces significantly with increasing matrix size.

To ensure that the overhead measured by our evaluation accurately reflects the inherent complexity of our method, we validated that our implementation in fact exhibits the asymptotic complexity we presumed. During this validation process, we eliminated several lurking quadratic complexities in our implementation by applying the techniques described in Section 4.2.1. We fit a multiple linear regression model using the variables N and NNZ to the benchmarked analysis times. This model yielded an adjusted R^2 value of 0.90. Parameter estimates and standard errors for the model are shown in Table 4.6.

We performed the same validation for the structure conversion step as well. Table 4.7 presents the results of this multiregression analysis, which includes a separate model for each block size. The adjusted R^2 values range from 0.77 to 0.88, confirming that the structure creation step incurs a linear cost in matrix size and number of nonzeros, similar to the matrix analysis step.

Table 4.7: Multi-regression parameters for PBR structure creation cost.

	2 × 2			**3 × 3**		
	Intercept	**N**	**NNZ**	**Intercept**	**N**	**NNZ**
Parameters:	-0.19	1.0016E-05	4.0002E-08	-0.040644	3.1825E-06	6.6172E-08
Std Error:	0.158165	8.165E-07	5.634E-08	0.048073	2.482E-07	1.712E-08
	Adjusted $R^2 = 0.773681$			Adjusted $R^2 = 0.820198$		
	4 × 4			**5 × 5**		
	Intercept	**N**	**NNZ**	**Intercept**	**N**	**NNZ**
Parameters:	-0.031686	2.5434E-06	6.8687E-08	-0.011349	1.7177E-06	7.2905E-08
Std Error:	0.037268	1.924E-07	1.327E-08	0.025303	1.306E-07	9.012E-09
	Adjusted $R^2 = 0.841013$			Adjusted $R^2 = 0.86814$		
	6 × 6			**7 × 7**		
	Intercept	**N**	**NNZ**	**Intercept**	**N**	**NNZ**
Parameters:	-0.008991	1.5042E-06	6.6265E-08	-0.001651	1.2776E-06	7.483E-08
Std Error:	0.021985	1.135E-07	7.831E-09	0.019354	9.99E-08	6.893E-09
	Adjusted $R^2 = 0.872665$			Adjusted $R^2 = 0.888206$		
	8 × 8					
	Intercept	**N**	**NNZ**			
Parameters:	0.0023503	1.1936E-06	7.322E-08			
Std Error:	0.01947	1.005E-07	6.935E-09			
	Adjusted $R^2 = 0.877092$					

Table 4.8: Cost of code generation/compilation. We only consider matrices, which yielded the best performance with a block size $> 4 \times 4$, thereby triggering code generation/compilation. ΔBEP and ΔCSR itr columns show the number of extra iterations incurred by code generation/compilation, to be added on values given in BEP and CSR itr columns in Table 4.5.

	Intel Harpertown				AMD Opteron			
Matrix	**Blocksize**	**# Patterns**	**Δ BEP**	**Δ CSR itr**	**Blocksize**	**# Patterns**	**Δ BEP**	**Δ CSR itr**
orsreg_1	7	2	25128	3860	7	2	10167	3444
sherman3	5	2	17845	1745	5	2	13534	1702
rdist1	6	8	8964	3125	6	8	12606	2806
memplus	8	18	20695	3439	-	-	-	-
wang3	6	2	1156	308	6	2	1551	256
wang4	6	2	1220	305	6	2	1704	257
lhr10	8	50	27774	7998	-	-	-	-
raefsky1	6	31	9408	4509	-	-	-	-
af23560	8	7	561	397	8	7	979	441
crystk02	-	-	-	-	6	10	379	216
rim	-	-	-	-	6	220	12434	4398
av41092	-	-	-	-	6	236	8660	2794
finan512	6	74	11438	1812	-	-	-	-
raefsky3	8	1	42	14	-	-	-	-
venkat01	8	15	362	143	-	-	-	-
ct20stif	6	227	5398	1371	-	-	-	-
dense2	6	3	44	13	-	-	-	-
pdb1HYS	6	15	208	60	-	-	-	-
shipsec1	6	1	8	3	-	-	-	-

Table 4.5 assumed that no code generation is necessary, which is true if the code cache contains already compiled modules for all encountered patterns. Table 4.8 shows the additional cost incurred by the code generation/compilation step for the subset of matrices that yielded the best performance for a block size greater than 4×4. We report numbers only for these matrices because we assume that the code cache is primed with all of the precompiled modules for smaller block sizes.

4.3.5 Predictor Model for Blocksize Detection

In Section 4.2.2, we described a method for estimating the PBR execution time in order to choose a block size that yields optimal or near-optimal performance. For each candidate block

Table 4.9: Predictor model R^2 values for each block size and the remainder CSR.

2 × 2	**3 × 3**	**4 × 4**	**5 × 5**
0.9948	0.9887	0.9754	0.9612
6 × 6	**7 × 7**	**8 × 8**	**Rem. CSR**
0.9524	0.9562	0.9707	0.9806

size, we first estimate the PBR and remainder CSR execution time using separate models; the overall predicted execution time is the sum of these two components. The model for the PBR part uses the three estimation factors (the memory reads due to the nonzero and index arrays, the approximation to the number of accesses to the x and y vectors, and the number of writes to y), as described in Section 4.2.2. The model for the remainder CSR uses matrix size and the number of remainder nonzeros as two estimation factors. Then, we pick the block size that leads to the smallest predicted execution time.

Since our model is based on the assumption that PBR performance is dominated by memory accesses, we chose the 29 largest matrices of our set to train the model. We verified that the overall memory consumption of these matrices exceeds the cache size, making PBR memory bound. The resulting adjusted R^2 values for these models on the Intel Harpertown architecture are given in Table 4.9; these high values strongly indicate a good fit. For these experiments, we do not consider the impact of vectorization and assume a constant prefetching distance.

We summarize the optimal and predicted block sizes and corresponding model error and performance loss in Table 4.10. This table shows that our model correctly selects the optimal block size for 12 of 29 matrices. Since the performance approximations for multiple block sizes can be very close to one other, incorrect selection of the optimal block size does not necessarily

Table 4.10: Accuracy of the predictor model. *Perf. Loss* column shows the slowdown caused by misprediction of the optimal block size. *Model Error* column shows the runtime estimation error relative to the best realized performance.

Matrix	**Opt. Blocksize**	**Pred. Blocksize**	**Model Error%**	**Perf. Loss%**
af23560	8	8	2.84	**None**
finan512	2	7	5.26	12.3
scircuit	2	8	28.35	2.05
crystk02	6	7	14.35	2.00
rim	6	7	8.82	2.43
ex11	4	4	15.12	**None**
mac_econ_fwd500	2	2	2.71	**None**
raefsky4	6	7	10.92	0.19
bcsstk35	6	6	14.29	**None**
raefsky3	8	8	16.16	**None**
av41092	6	4	-3.53	0.86
venkat01	8	4	17.48	2.99
crystk03	8	3	7.01	5.17
qcd5_4	8	3	9.68	0.68
mc2depi	2	2	12.10	**None**
rma10	5	5	-1.90	**None**
nasasrb	6	6	8.47	**None**
ct20stif	6	5	-1.67	1.27
webbase-1M	2	2	2.55	**None**
3dtube	8	3	3.04	1.37
laminar_duct3D	6	3	-3.85	2.71
dense2	8	6	5.09	1.16
cant	6	3	-5.66	4.36
pkustk04	6	3	4.46	0.61
pdb1HYS	6	3	0.89	3.27
Si34H36	6	2	-8.98	4.63
consph	3	3	1.81	**None**
shipsec1	6	6	4.02	**None**
pwtk	6	6	0.30	**None**

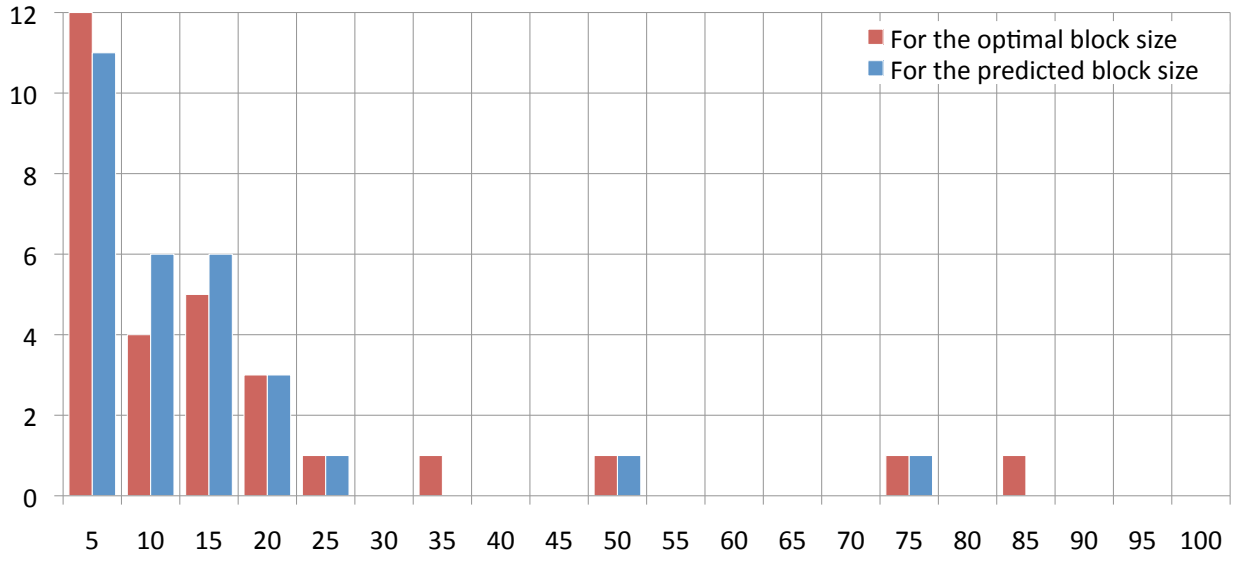


Figure 4.17: Weighted sum of absolute relative errors in the PBR predictor model. The number of matrices that fall in each of the model error η_{sum} bins (from 0% to 100% with 5% increments), for approximations of the performance for both optimal and the predicted block sizes.

lead to significant performance loss. For the 17 matrices with mispredicted optimal block sizes, the average performance loss did not exceed 3%, with a maximum of 12.3% for the *finan512* matrix.

Because our prediction includes two components, the performance of PBR and the performance of the remainder CSR part, prediction errors with opposite signs may cancel each other out, which could misleadingly yield a good fit. To validate that the high accuracy of our model is not due to this effect, we computed the weighted sum of the absolute values of the relative error η for both the PBR and the CSR remainder parts:

$$\eta_{sum} = (cov) \times |\eta_{PBR}| + (1 - cov) \times |\eta_{Rem.CSR}|,$$

where *cov* denotes the nonzero coverage by PBR. Figure 4.17 shows a histogram of how many matrices fall in each of the model error η_{sum} bins. Our model accurately approximates the performance of not only the optimal block size, but also of the predicted block size, which are both depicted in the figure. The majority of matrices has a total weighted error of less than 25%, validating that our model has a good fit and the cancelation of model errors is a rare occurrence.

Overall, these results show that the block sizes selected by our predictor model lead to optimal or near-optimal speedup. We note that block size prediction is an optimization; if the exact optimal block size is desired and a matrix structure is sufficiently often reused, it is possible to perform a trial structure conversion using each block size and choose the optimal block size via benchmarking.

4.4 Applicability of PBR

In this section, we evaluate the applicability of PBR by testing it against a much wider set of problems than in Section 4.3, characterize the types of problems for which PBR is most and least suitable, and demonstrate how a code cache implementation can significantly mitigate code compilation costs. Since PBR's performance is dependent on matrix nonzero structure, and since matrix structure in turn is dependent on the underlying problem, it is critical to evaluate PBR on problems drawn from as wide a range of application areas as possible. Furthermore, a better understanding of the link between PBR-SMVM performance and underlying matrix structure will guide practitioners in deciding whether or not to use PBR, and may suggest alternatives for algorithm designers who can influence the structure

Table 4.11: Number of matrices for each problem kind.

Problems with 2D/3D Geometry		**Problems without 2D/3D Geometry**	
Problem Kind	**# Matrices**	**Problem Kind**	**# Matrices**
2D/3D	35	chemical process simulation	26
acoustics	5	circuit simulation	49
computational fluid dynamics	40	combinatorial	5
computer graphics/vision	2	counter-example	1
electromagnetics	8	directed graph	11
materials	25	directed weighted graph	16
model reduction	45	directed weighted random graph	1
random 2D/3D	2	economic	44
semiconductor device	28	frequency-domain circuit sim.	4
structural	118	optimization	89
thermal	15	power network	54
		statistical/mathematical	2
		theoretical/quantum chemistry	16
		undirected graph	1
		undirected weighted graph	68

of matrices, e.g., via discretization techniques or matrix reorderings. Since the primary advantage of PBR is in reducing memory bandwidth usage (which is directly related to matrix nonzero coverage), and since the most expensive overhead of PBR is code generation and compilation, we explore these two factors in our experiments.

We use all non-complex square matrices of dimension 10,000 or larger from the University of Florida sparse matrix collection [Dav]—a total of 710 matrices drawn from 599 applications with widely varying origins. Each matrix in the collection is associated with a problem *kind* indicating the application area from which it is taken. Table 4.11 lists the 26 problem kinds included in our study, split into two groups (following the characterization in [Dav]): problems with no underlying geometry and problems with 2D or 3D geometry.

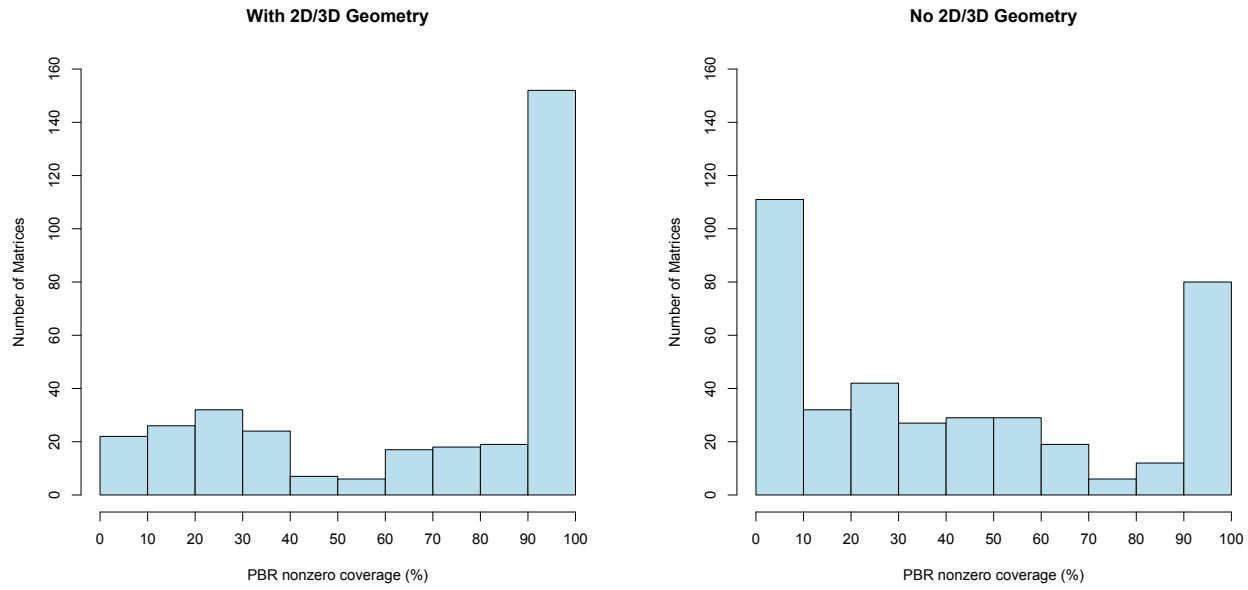


Figure 4.18: PBR nonzero coverage of problems with and without 2D/3D geometry. 323 matrices have 2D/3D geometry (left), and 387 problems do not have 2D/3D geometry (right).

We ran the PBR analyzer on each of the 710 matrices, with block size $k \times k$, for $k = 2, \ldots, 8$. To predict the best block size, we used the model described in Section 4.3.5 with parameters acquired on a 2.8GHz 2-socket quadcore Intel Xeon Harpertown 5400 with 12 MB L2 cache and 4GB RAM per socket.

4.4.1 Nonzero Coverage with PBR

Since a necessary condition for applying PBR is that a high percentage of nonzeros be covered by PBR, we turn first to this metric. At first glance, PBR nonzero coverage seems somewhat uniformly distributed across the 710 matrices. For example, coverage is greater than 50% for 358 of the matrices (50.4%), leaving approximately half of the matrices with coverage less

than 50% with PBR, which suggests that PBR will not achieve noticeable speedups on these matrices. Matrices with high PBR nonzero coverage are very likely to benefit significantly from speedups due to reduced memory bandwidth requirements. For 232 matrices (32.7%) we see coverage of over 90%, which leads to an average reduction of 77% in the indexing information, resulting in an average of 26% savings in overall bandwidth usage.

Figure 4.18 shows the two groups, matrices with underlying 2D/3D geometry (left) and those without (right). The coverage for most matrices with underlying 2D/3D geometry is high, with 65.6% of the matrices showing over 50% coverage and 47% above 90% coverage. Coverage varies greatly for matrices with no 2D/3D geometry. We note that 67 of the 111 matrices with less than 10% coverage in Figure 4.18 (right) come from a single problem, the Reuters problem, whose coverage is 0%. This problem generates extremely irregular and sparse matrices where nonzeros represent occurrences of words in news articles. Figure 4.19 shows the results from four other problem kinds, including one with very high coverage (structural problems) and others with a mix of coverage results (computational fluid dynamics, optimization and simulation problems).

To give more informed guidance to practitioners and algorithm developers, we take a closer look at the relationship between matrix structure and nonzero coverage by considering the four boundary groups in Figure 4.18: matrices with less than 10% coverage, greater than 90% coverage, and with and without underlying 2D/3D geometry. For the 111 matrices with poor coverage and no underlying 2D/3D geometry, an average of over 90% of the nonzeros end up in the remainder matrix because they are found in blocks with less than three nonzeros. The dominance of 1- and 2-nonzero blocks in these extremely sparse and irregularly structured matrices, which lack recurring relationships between rows or columns,

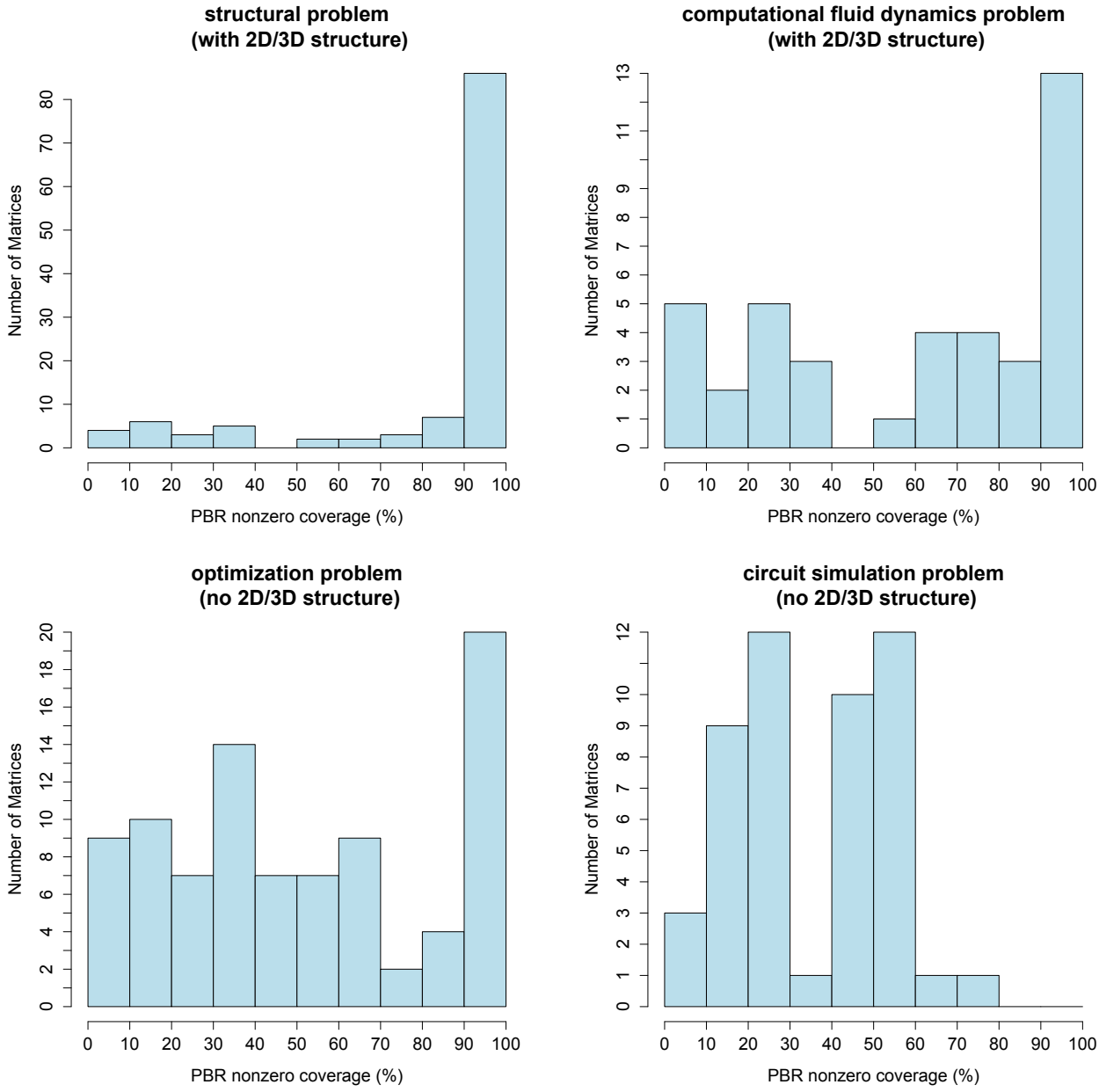


Figure 4.19: PBR nonzero coverage for four example problem kinds: structural (top left) and computational fluid dynamics (top right) have 2D/3D structure; optimization (bottom left) and circuit simulation (bottom right) do not.

implies that PBR is unlikely to yield noticeable speedups on such problems. On the other hand, for the 22 poorly covered matrices from the group with underlying 2D/3D geometry, more than a quarter of nonzeros (26%) were rejected because they did not cover at least 1000 nonzeros, although the pattern they belong to has 3 or more nonzeros. The coverage for these problems with underlying geometric structure would improve if larger versions of these problems were solved. For the 15 CFD problems with coverage less than 40% (Figure 4.19, upper right), an average of 27% of the nonzeros are in blocks that failed to achieve the required 1000 nonzero total.

Turning to the high (>90%) coverage problems, we see that good coverage stems most often from many occurrences of a very small number of patterns. For problems with underlying 2D/3D geometry this is not surprising, since many such problems correspond to discretized PDEs, where finite difference stencils or finite element formulations yield regularly structured sparse matrices. For example, for the 152 high coverage problems with underlying 2D/3D geometry, the most frequently occurring five patterns (for each matrix) account for an average of 82% of the nonzeros; in fact, 94 of these 152 matrices obtain more than 90% coverage from only five patterns or less. The dominance of a few patterns is not as strong for problems with no 2D/3D geometry. Only 26 of the 80 matrices in this category achieve 90% coverage with five patterns or less.

4.4.2 Using a Code Repository

Code generation and compilation of pattern-specific SMVM kernels is costly. If code generation is needed, its overhead adds to the overhead of performing the PBR analysis and

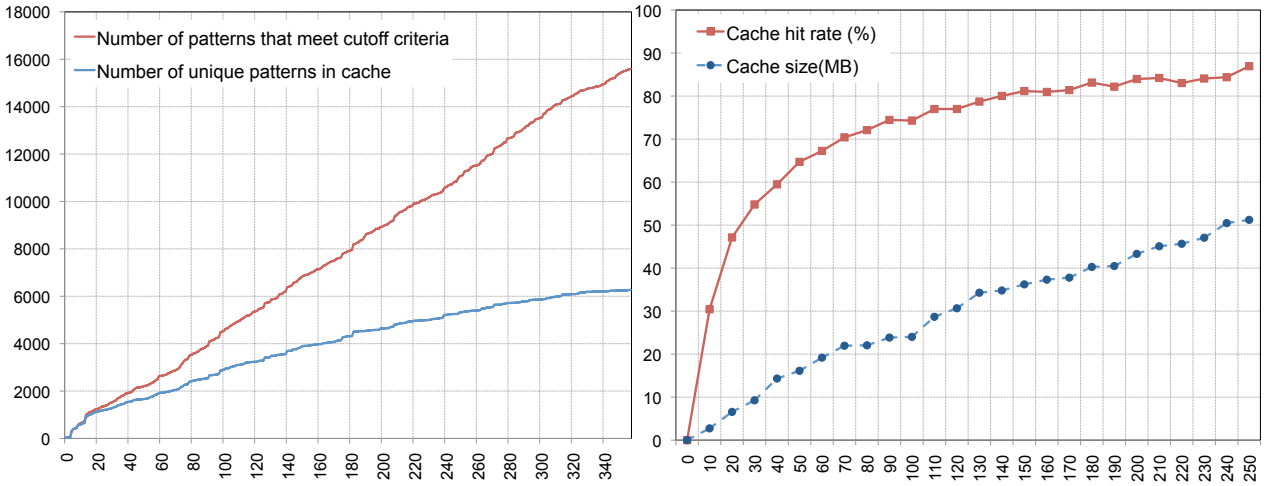


Figure 4.20: Hit rates and size of the PBR code cache. On the left figure, we compare the total number of qualifying patterns to the number of unique patterns in the cache. The number of unique patterns shows a much smaller growth rate. On the right, we show the average cache hit rate after priming the cache with randomly selected subsets of matrices. The x-axis denotes how many matrices are added to the cache before measuring and averaging the cache hit rates for each matrix not used for priming. The growth in the size of the cache (in MB) is depicted on the same figure.

structure conversion, thus increasing the number of SMVM operations needed to amortize it. In Section 4.3.4, we show that the break even point for PBR can range from an average of 459 operations if no code generation is required to an average of 2682 if code for all qualifying patterns must be created (for matrices with more than 50% coverage).

Fortunately, the use of a code repository can reduce or eliminate the cost of code generation in many cases. We have implemented a code repository, which caches compiled kernels as dynamically loadable object modules. In this section, we evaluate the storage requirements and hit rates for this cache. The left half of Figure 4.20 shows how many qualifying patterns occur, on average, in 10 randomly selected subsets with varying sizes. In this experiment, we choose subsets from only the 358 of 710 matrices that achieved at least 50% non-zero

coverage, for which PBR is likely to provide speedups. We also show, on the same figure, how many of these qualifying patterns are unique, i.e., how many distinct codes need to be stored in the cache.

We draw two conclusions from this experiment. First, the experiment shows that only a small fraction of all theoretically possible patterns qualify—only 15595, including those that are reused, for the test set we considered. This relatively small number is explained by the use of the cutoff-criteria which eliminates rarely occurring patterns as well as all 1-bit and 2-bit patterns. Consequently, it becomes feasible to store code for all patterns on disk. Second, the figure shows that the number of unique patterns encountered grows much more slowly than the total number of patterns, indicating a high degree of pattern reuse and suggesting that the cache performs well.

The right half of the figure quantifies the cache hit rates we observed. The cache is primed with a randomly selected subset of N matrices and the average cache hit rate for all $(358-N)$ matrices not used for priming the cache is computed. This experiment is repeated for 10 different random subsets; each data point represents the average of these 10 runs. On the same figure, we also show the disk capacity required by the optimized and stripped GCC-compiled object code for patterns occurring in the 358 matrices with more than 50% coverage. The results show that less than 60MB of disk space is enough to store codes for the block sizes providing the best coverage. Even if all block sizes and all 710 matrices are considered, the required space would still be less than 1 GB.

The results show that 70 matrices are needed to achieve a hit rate of 70%, 140 matrices to achieve a hit rate of 80%, which grows to 86% for 250 matrices. This results suggests that

even with a set of matrices as diverse as this one, a code repository will yield substantial savings in kernel generation and compilation. Finally, note that if all matrices are drawn from one problem kind (e.g., on a computational resource used by only one research group), the code repository will likely yield higher hit rates while still remaining of manageable size.

Chapter 5

Related Work

The fundamentals and implementations of our methods are inspired by and improve on a wide range of available computational techniques in the literature, including sparse storage formats, bandwidth reduction, data and computation reordering, vectorization, parallelization, matrix splitting, manual and compiler optimizations, and software prefetching, among many others. In this chapter, we provide a categorized summary of related work and explore its relationships with our methods.

Most related to our work are techniques to reduce the memory bandwidth usage of SMVM kernels and irregular applications in general. The sustainable memory bandwidth of architectures can be measured or estimated by using several techniques. The *STREAM* benchmark measures the sustainable memory bandwidth by using four common kernels, namely COPY, SCALE, SUM, and TRIAD [McC95]. However, STREAM may report misleading results due to its assumption that memory read and write operations incur similar latencies, which

is no longer valid for modern architectures. It is more realistic to compare the memory bandwidth usage of an irregular application to an upperbound the algorithm theoretically allows, instead of to the machine's peak memory bandwidth. Geus and Röllin discussed that benchmarking highly optimized computational kernels, such as vendor-supplied BLAS daxpy and other routines, is a more realistic approximation of the sustainable bandwidth than the peak memory bandwidth numbers reported by vendors [GR01]. Lee et al. suggested a model to approximate lower bounds for both execution time and cache misses [LVDY04]. Their lower bound for performance only considered memory operations, given that SMVM is memory bound. These authors assume write-back caches with sufficient buffer capacity, therefore they account for only the cost of loads, ignoring the cost of stores. Vuduc et al. developed an analytical performance bound model for the SpTS operation that uses blocking for register re-use [VKH+02]. These authors separate the *dense trailing triangle* from the sparse sections of the lower triangular matrix and count the loads and stores for both sections, and they exclude floating point operations from the model, since they are hidden by memory operations. Williams et al. used the {flop:byte=1:4} ratio as an upperbound, which assumes that SMVM can be modeled as streaming a double array from memory and performing a multiply-add operation on each element [WOV+07].

5.1 Reducing Matrix Storage Overhead

5.1.1 Sparse Matrix Storage Formats

The optimal choice of sparse storage format depends on the structure of a matrix, the architecture on which the code will be run, and the operation(s) to be performed. If the structure of a matrix is not known beforehand, either *Coordinate (COO)* format, or the *Compressed Sparse Row (CSR)* format [Saa96, BBC+94] can be used, with CSR being the most common choice due to its smaller storage requirements. We used CSR as a reference point to evaluate OSF and PBR, since CSR is very common and it makes no assumptions about the matrix structure. In addition, CSR, with its generality, applicability, and performance, has been identified as a proper choice for benchmarks and comparison-based performance studies by Vuduc [Vud03]. The *Compressed Sparse Column (CSC)* format [BBC+94] is a modified version of CSR, which assumes compressed storage of columns, instead of rows. This format constitutes the base for the widely used *Harwell-Boeing* sparse matrix format [ID92]. We provided detailed explanations of COO and CSR in Chapter 2.1.

The indexing overhead can be reduced further if the matrix structure is known to have certain characteristics. For example, if the matrix nonzeros are located consecutively around the diagonal, i.e., if the matrix is banded, then the *Compressed Diagonal Storage (CDS)* format can be used to completely eliminate all indexing overhead. This format keeps each parallel diagonal in an array [BBC+94], since the locations of elements on the band are already known. However, CDS loses its advantage if there are many zero elements in between nonzeros on a row, since this format stores all consecutive elements from the leftmost nonzero to the

rightmost nonzero. For such cases, the *ELLPACK-ITPACK* format can be more space-efficient than CDS since it maintains a separate 2-D index array to keep the column indices of nonzeros, so that zero elements in between nonzeros can be skipped. This format is used in the ELLPACK [RB84], ITPACK [GKY79, KY84], and ESSL [IBM88] numerical libraries.

A modified version of CDS is given by Melhem [Mel87]. This variant targets the case when the matrix's bandwidth varies. Unlike CDS, Melhem's approach includes preconditioning of the matrix and a gather operation. Although computationally more expensive than CDS, this variant uses less indexing overhead and the resulting format is more suitable for vectorization.

Another approach to reducing the indexing overhead is the *feature extraction based algorithm, (FEBA)* by Agarwal et al. This algorithm reduces indexing overhead by extracting dense and near-dense blocks and diagonals from the matrix and storing the remaining nonzeros using a modified ELLPACK-ITPACK format [AGZ92]. This format requires identification of these dense substructures, however.

If the diagonal includes varying numbers of nonzeros, then the *jagged diagonal format (JAD)* can be more effective than the ELLPACK-ITPACK format because it does not use zero filling [Saa88]. Instead, JAD picks the left-most nonzero in each row and stores nonzeros consecutively, forming the so-called *jagged diagonals*. This operation is repeated until all nonzeros are stored. JAD uses a compressed index representation, similar to CSR, to further reduce the indexing overhead.

Since OSF is algorithm-independent (see Section 3.1.3), it can stack any sparse matrix storage format of choice. The primary benefit of OSF comes from sharing a single copy of indexing arrays, therefore formats that include no indexing overhead, such as CDS, are less likely

to benefit from this technique. On the other hand, stacking such formats may still lead to performance improvements due to improved cache utilization and ILP efficiency.

PBR, on the other hand, is a new sparse storage format and requires an analysis and structure conversion if the matrix is not already created and stored in this format. PBR internally uses CSR to store the nonzeros that do not meet the inclusion criteria we describe in Section 4.1.2. Compared to CSR, PBR can eliminate a large fraction of indexing overhead (by up to 95%, see Table 4.1) via code generation. To our knowledge, PBR is the first method to exploit code generation to eliminate indexing overhead.

More detailed information regarding sparse matrix storage formats, along with algorithms, illustrations and examples, are given in surveys by Barrett et al. [BBC+94] and Vuduc [Vud03].

5.1.2 Block-based Representations

If matrices involve dense nonzero substructures, sparse storage formats can be adapted to work on blocks of nonzeros, instead of individual nonzeros, to further reduce the memory bandwidth usage. For example, *block CSR (BCSR)* [PH99, BBC+94] uses the CSR format to store fixed-size $R \times C$ dense blocks, reducing the column index array size from (nnz) to $(\frac{nnz}{R \times C})$ and the row index array size from $(n+1)$ to $(\frac{n}{R}+1)$, when compared to regular CSR. Unlike BCSR, *variable block row (VBR)* supports variable block sizes, but this format fails to achieve comparable performance because of the varying blocksizes in the inner loop [Vud03]. Yet, Vuduc and Moon creatively used VBR as an intermediate tool for identifying dense substructures in the matrix [VM05]. In addition to index overhead savings, computations on

dense blocks facilitate better register re-use, improve cache locality, and allow vectorization and loop-unrolling, as we summarize in the following section. Highly tuned dense libraries, such as Intel's MKL [INT09b], ATLAS [WPD01], and GOTO [GvdG02], can be used for computation on these dense blocks at near-peak speeds.

Block-based techniques are used by SMVM libraries, such as the *SPARSITY* framework [IYV04, IY01] and the *BeBOP Optimized Sparse Kernel Interface (OSKI)*. Unfortunately, blocking includes several drawbacks. First, blocking cannot be applied to sparse matrices that do not contain dense or near-dense substructures. Second, the detection of dense substructures, if they exist, is a non-trivial task [PH99, Vud03, VM05]. Third, near-dense blocks can be turned into dense ones by zero filling, costing additional memory bandwidth and requiring redundant floating point operations on zeros. Researchers often turn to heuristic methods or performance models to decide whether or not to use zero filling [Vud03].

Since OSF can stack any given algorithm, it could be used to improve the performance of block-based methods as well. For example, block-CSR (BCSR) could be stacked using the algorithm-independent OSF developer library, following a process similar to stacking the regular CSR format we describe in Appendix A.

PBR internally utilizes the BCOO format to store block indices, using 2 integers per block. We chose not to use a compressed format, such as BCSR, since BCSR could easily become inefficient if a pattern does not occur at least once per each block-row, which is a likely case in a split matrix. In other words, keeping a compressed row index array (similar to the IA array in CSR), with a fixed-length of $N/R+1$ for submatrices containing less than $N/R+1$ blocks is more costly than simply keeping these blocks using BCOO. It is possible to find

matrices that do not fit in this category and would benefit from BCSR, therefore BCOO and BCSR could be interchangeably used in PBR depending on the matrix structure, which we leave for future work.

5.1.3 Index Compression and Encoding

To reduce the indexing overhead, it is possible to compress and encode the index arrays and also the matrix values in certain cases. Willcock and Lumsdaine described the *delta-coded sparse row (DCSR)* and *row pattern compression (RPCSR)* techniques to compress index arrays [WL06]. DCSR encodes the differences between the column indices of nonzeros of a matrix in a 32-bit wordcode. RPCSR groups intervals between nonzeros with respect to their lengths and row positions, which are referred to as 'patterns,' so that matching patterns can be merged to reduce indexing overhead. Patterns in RPCSR are formed by nonzero intervals, therefore they are conceptually different than PBR patterns, which are formed by matrix substructures. The current implementations of DCSR and RPCSR are hand-tuned machine codes, which are not portable across different architectures. Kourtis et al. suggested *delta unit CSR (CSR-DU)* as an alternative to DCSR, which can take advantage of near-dense, but not necessarily contiguous nonzero concentrations [KGK08]. These authors also argued that, similar to index arrays, compressing recurring nonzero values reduces the memory bandwidth usage, as implemented in the *value indexed CSR (CSR-VI)* storage format. CSR-VI keeps only one entry for each recurring nonzero in a 'unique values' array and replaces nonzeros in the matrix with integers that point to the corresponding elements in this array. Replacing floating point representations with integer indices increases indexing overhead, but reduces overall memory bandwidth usage because integer numbers consume fewer bytes than double

numbers do. However, this representation may lead to less efficient computation, since it requires indirect accesses to the unique values array. Another approach to compressing indices is the run-length encoding by Park et al. [PDZ92], which is particularly effective if rows contain contiguous nonzeros, as in banded matrices or matrices with triangular form.

5.2 Register and Cache Blocking with Matrix Splitting

Block-based approaches and matrix splitting have not only been used for reducing memory bandwidth usage, but also for improving cache utilization and increasing register reuse.

Toledo implemented register blocking by splitting a matrix into three submatrices containing two small blocks (1×2 and 2×2) and the remainder nonzeros, to reduce the number of load instructions and allow reuse of data in the registers [Tol97]. Pinar and Heath developed a heuristic to reorder a matrix to construct larger dense blocks, stored using BCSR, but this method allows only fixed-size aligned blocks [PH99]. Vuduc and Moon eliminated these limitations of BCSR by introducing the *unaligned block compressed sparse row (UBCSR)* format and using matrix splitting [VKH+02].

The use of matrix splitting in PBR closely resembles these methods. PBR separates the optimizable section of matrices, using a set of submatrices, from the non-optimizable section, which is stored in CSR format. Very similarly, Toledo uses two optimizable submatrices for 1×2 and 2×2 blocks and a third non-optimizable submatrix in CSR. Unlike in this work, PBR makes no assumptions regarding the block structures other than the two inclusion criteria to ensure efficiency. PBR does not search for specific block structures (i.e., *dense*

blocks), it simply records whatever recurring block structure it encounters during the analysis step. In addition, to our knowledge, PBR is the only method that generates code for each submatrix.

Im and Yelic introduced *SPARSITY* as a register blocking framework, which automatically determines if register blocking is beneficial, and if it is, uses the optimal block size that is predetermined by architectural models included in their framework [IY01, IYV04]. These authors also explored cache blocking, and concluded that cache blocking could improve performance only if there exist multiple right hand side (RHS) vectors. Nishtala et al. showed that cache blocking techniques can in fact be beneficial for the single RHS case when the x vector is large enough to exceed the cache capacity, the y vector fits in cache, the matrix does not include clustered nonzeros (i.e., dense substructures), and the matrix is sparse enough to prevent efficient register blocking [NVDY04, NVDY07]. Vuduc et al. extended SPARSITY to support the sparse triangular solve (SpTS) operation [VKH+02, Vud03]. Lee et al. explored exploiting matrix symmetry and vector blocking [LVDY04] to improve SMVM performance. Register and cache blocking techniques are combined in the OSKI library [VDY05], which is a commonly used library for fast SMVM kernels tuned for the architecture and the input matrix. Our evaluation shows that both OSF and PBR can match or outperform OSKI performance on recent architectures (Figures B.3, B.4, 4.13, and 4.14).

While primarily targeting reductions in memory bandwidth usage, both PBR and OSF allow for register reuse and improve cache utilization as two additional benefits. Similar to Im and Yelic's SPARSITY [IYV04] framework, OSF promotes register reuse by stacking multiple operations under a single loop, and reusing the array elements kept in registers. In addition, stacking multiple problems, each with a single right hand side (RHS), leads to similar cache

utilization improvements as solving a single problem with multiple RHS vectors. OSF and SPARSITY both utilize data interleaving to exploit this particular case, but OSF does so by solving multiple problems.

PBR allows reuse of data in registers as well, similar to all other block-based approaches, including SPARSITY. Unlike SPARSITY, PBR does not use an autotuning approach to select a block size, mainly because it makes no assumptions about block structures and the corresponding search space would be too large for autotuning to be feasible (2^{64} different structures for a 8×8 PBR block vs. one for a 8×8 dense block). Our model to predict an optimal block size based on statistical data, described in Section 4.2.2, eliminates the need for autotuning.

5.3 Microarchitecture-level Optimizations

Williams et al. evaluated a number of micro-level SMVM optimizations such as thread blocking, cache and register blocking, prefetching, and vectorization of dense substructures on single and multi-core architectures [WOV+07]. Goumas et al. revisited existing SMVM methods, with a focus on explaining the interaction between the architecture and factors that reduce SMVM performance [GKA+08]. This section summarizes these optimizations and their relationship with our methods.

5.3.1 Vectorization and Loop Unrolling

Unrolling and vectorization in sparse computations, and irregular applications in general, is known to be a challenging task due to irregularities in data access and computation patterns. Most techniques rely on reordering data and computation to benefit from these optimizations.

CSR cannot facilitate unrolling or vectorization without modifications. Mellor-Crummey and Garvin described a CSR variant, *length sorted CSR (L-CSR)*, to allow loop unrolling if the majority of rows in a matrix share a small set of row-lengths[1] [MCG04]. This method permutes the matrix to group rows by their row lengths. White and Sadayappan suggested sorting matrices by row lengths, creating block structures, if groups of rows have the same length [WS97]. These authors suggested keeping sorted rows in blocked column-major (BCM) or blocked row-major (BRM) CSR, and showed that either format can be effective, depending on the matrix structure. Similarly, *permuted CSR (CSRP)* format, which is similar to CSR, uses a permutation vector to group rows of the same length to promote efficient vectorization [DFM05]. Unlike BCM-CSR and BRM-CSR, CSRP requires no data rearrangement.

The JAD and ELLPACK-ITPACK formats are inherently vectorizable, but SMVM performance using these formats is very sensitive to varying row-lengths. Paolini and di Brozolo suggested sorting rows in the ELLPACK-ITPACK format in decreasing row lengths to implement more efficient vectorization [PRdB89]. Blelloch et al. used segmented sums to implement vectorization for matrices, which include varying row lengths [BHZ93].

[1] The number of nonzeros in a row

The current implementation of OSF explicitly utilizes loop unrolling via code generation, which emits separate code for each possible stack depth at compile time. This is different from other unrolling approaches, which involve grouping matrix rows by size, then modifying the loop iterator accordingly. OSF achieves unrolling by inlining each stacked operation inside a loop, instead of introducing a secondary inner loop that iterates over stacked operations (see Figure 3.2).

PBR does not explicitly implement loop unrolling, however operations for each nonzero element in a block pattern are inlined by the code generator. In this way, PBR inner loops are populated with multiple independent operations, allowing efficient pipelining and improved ILP, similar to unrolled codes (see Figure 4.1).

PBR utilizes matrix analysis and data reordering to support vectorization, similar to several other format-based vectorization approaches (e.g. JAD, ELLPACK-ITPACK, BCM-CSR, etc.). The matrix nonzeros are placed in zig-zag order (see Figure 4.7) to allow the code generator to emit vectorized codes using SIMD intrinsics.

The structure of OSF is suitable for vectorization, since each loop includes a constant number of independent operations. Interleaving the matrix nonzeros (stacked array AA in Figure 3.2) could allow easier vectorization because consecutive AA elements from different problems could be loaded into the same vector register using a single load operation, similar to the stacked array x, which we already interleave. However, our current evaluation does not include vectorization, which is left for future work.

5.3.2 Prefetching

The primary purpose of prefetching is to hide memory access latencies by fetching the data from memory before it is used by the CPU. For *streaming* applications, such as SMVM, memory access latency cannot be hidden, since these applications are already bounded by memory bandwidth due to their low flop:byte ratio. However, there are several other ways to utilize prefetching.

Toledo noted that although software prefetching (SWP) cannot reduce memory latency in SMVM computations, it can be used to reduce the memory bandwidth usage by avoiding multiple load/store stalls on the same cache line [Tol97]. Goumas et al. compared SWP to hardware prefetching (HWP) and concluded that SWP is not necessary if the processor has support for HWP [GKA+08]. Williams et al. confirmed the conclusions of Goumas et al. on Intel architectures, but added that AMD architectures can still benefit from SWP, despite their HWP support [WOV+07]. These authors showed that SWP can significantly improve performance if dedicated prefetch instructions are used to tag data with locality information, so that temporal data obtains priority over non-temporal data during replacement [WOV+07, WOV+09].

Our use of prefetching in PBR is almost identical to Williams et al.; we tag the streaming array of nonzeros as non-temporal and find the optimal prefetch distance with an exhaustive search. Our performance evaluation confirms these authors' conclusion that SWP is effective on AMD, and it is not on Intel machines. Our current OSF implementation does not include prefetching.

5.3.3 Sparse Matrix Compilation

Matrix compilation methods shield the optimizing compiler from irregularities of sparse computations via code preprocessing and generation before compilation takes place. Gustavson et al. described a code generator that generates FORTRAN code that processes only nonzeros in a matrix and implements the Crout elimination method [GLW70]. Bik and Wijshoff developed a source to source code translation technique that translates a dense code, which can be optimized easily by compilers, into an equivalent sparse code [BW93, Bik96]. Kotlyar et al. approached sparse matrix formats as if they were *database relations* and developed the *Bernoulli compiler* that can work on any user-defined format [KPS97a, KPS97b, KPS97c], thus eliminating dependence on formats when compared to Bik and Wijshoff's code translator. Kotlyar and Pingali extended the Bernoulli compiler to also handle imperfectly nested loops with dependencies [KP97].

Similar to these techniques, OSF utilizes a code transformation and generation process, which converts codes written in the OSF macro API into pure C code (Figure 3.11). This conversion assists the compiler in implementing more efficient optimizations, since the generated code includes manually unrolled inner loops, allowing the compiler to optimize the surrounding outer loop. Code generation in PBR also allows for efficient compiler optimizations, by inlining operations in the inner loop, explicitly keeping data in registers, and including hints for prefetching. Both OSF and PBR differ from other matrix compilation techniques in that they generate code primarily to reduce indexing overhead.

5.3.4 Reordering and Matrix Bandwidth Reduction

SMVM algorithms operating on matrices with large bandwidths access large x array segments, which potentially leads to more frequent irregular accesses to memory, significantly increasing the number of cache misses. Matrix bandwidth reduction techniques aim to mitigate this situation by altering data and computation patterns to narrow matrix bandwidth and regularize access patterns of SMVM algorithms.

To reduce the matrix bandwidth, Cuthill and McKee developed an automatic nodal numbering technique using breadth first search (BFS) [CM69]. This method, and its reverse-ordered variant, were widely studied and several improvements were suggested [LS76, Tol97, DMS+92, TJ92, GR01]. Other work include a hybrid scheme that combines BFS with graph partitioning [AFR98], a first touch reordering technique [DK99], the use of space filling curves [ORF96, AFR98, MCWK01], and a recursive partitioning algorithm [BB87]. Several performance models to analyze the behavior of reordering techniques were also suggested [TJ92, BG97]. Han and Tseng suggested two new algorithms, namely *GPART* and *Z-SORT* for efficient data and computation reordering [HT06]. These authors target a difficult but frequent case, where multiple irregular accesses are made inside a single loop iteration. Temam and Jalby reduced the bandwidth by splitting a matrix into a sum of submatrices, each with a bandwidth that is smaller than the cache size [TJ92].

PBR utilizes data reordering to achieve matrix splitting. Unlike data reordering techniques for bandwidth reduction, PBR aims to improve locality in the instruction cache, where the generated code is kept, but not in the data cache. PBR also utilizes computation reordering to process consecutive blocks in submatrices. OSF uses data reordering to interleave stacked

array elements for improved cache utilization.

5.4 Parallelization

Distributed parallel libraries, such as PETSc[2] [BB+08] and Trilinos [HBH+03], sequentially compute SMVM for a data partition on each node. OSKI provided an extension for the PETSc library in 2006 [OSK10] as a faster sequential SMVM alternative to be used in this parallel library. Increasingly, nodes of distributed systems include additional layers of parallelism, such as chip multiprocessors (CMP), graphical processing units (GPU), and STI Cell [Gsc06]. Compounding the problem, distributed systems are becoming more and more heterogeneous, i.e., several of these components can co-exist in the same distributed system. The performance problem of SMVM, with its irregular and streaming nature, becomes more acute on heterogeneous systems.

Williams et al. explored SMVM performance on modern multicore architectures and suggested new optimization techniques [WOV+07, WOV+09]. They considered various machines including x86-based multicore machines, two types of STI Cells, and Sun Niagara [JN07]. This study investigated how different architectures respond to different optimization techniques. The parallelization techniques studied include thread-blocking, NUMA-aware optimizations (processor and memory affinity), and row partitioning. Unlike OSKI, the authors have not released a version we could use to compare against OSF and PBR.

[2]PETSc also offers a row-partitioning distributed SMVM routine, but solving SMVM sequentially on each node for partitioned workload leads to more efficient and scalable parallelization when distributed nodes consist of single cores.

Future heterogeneous platforms will increasingly demand algorithms that can dynamically adapt to system components with varying characteristics. Autotuning frameworks are promising tools for achieving this task. Choi et al. described a model-driven autotuning framework for parallel SMVM [CSV10]. The authors evaluated their framework on a NVIDIA T10P GPU and showed that it can outperform the vendor-provided SMVM implementation. They present a new sparse format, namely Blocked ELLPACK (BELLPACK), which expands on the ELLPACK format by explicitly storing dense blocks and avoiding unevenly distributed workloads via row permutation. The vendor-provided SMVM is also based on an ELLPACK variant, but it does not consider blocking [BG09]. A study by Baskaran and Bordawekar describes a compile- and run- time framework for SMVM on GPUs, which achieves a performance comparable to NVIDIA's vendor implementation [MMB08]. Work on GPU SMVM was pioneered by Bolz et al. [BFGS05] and Sengupta et al. [SHZO07], however these implementations could not outperform CPU-based implementations at the time [CSV10].

The mainstream architectures at the time of this dissertation are cache-coherent multiprocessors, which we use for the evaluation of our methods. On these architectures, the CSR method achieves efficient parallelization with straightforward row-partitioning. Unlike CSR, our first implementation of parallel PBR partitioned the contiguous arrays of nonzeros in the PBR submatrices; however, we found that the necessary reduction operations to compute the vector y significantly limited performance. We were able to solve this issue by switching to a row-partitioning implementation (see Section 4.2.7). The literature includes several approaches to optimize parallel reduction in irregular applications, for cases in which reduction is unavoidable. These efforts mostly focus on data and computation reordering in an attempt to keep the data local to processors. Gutiérrez et al. describe a compiler

method to group reduction loop iterations into sets that are further assigned to threads, in an attempt to distribute the reduction loop as conflict-free as possible [GPZ00]. Yu and Rauchwerger describe an adaptive framework that chooses among several reduction algorithms (either existing or novel algorithms by the authors) at runtime, depending on the state of the computation using a decision-tree based selection tree [YR00]. Han and Tseng describe the *LOCALWRITE* algorithm, which partitions irregular reduction operations so that each processor computes values for their local data [HT98, HT00, HT06].

Chapter 6

Future Research Directions

In this chapter we describe several future research directions that build on our development of OSF and PBR. We explain the motivation behind each of these directions, as well as how they will impact the performance and applicability of our methods. We tag each suggested future work as short-term or long-term for realistic time planning.

6.1 Operation Stacking Framework

Using OSF to accelerate existing applications (short-term). This dissertation presents a detailed and comprehensive evaluation of the OSF in Section 3.4. We do not, however, demonstrate the use of OSF for an existing scientific/engineering application, although we expect that doing so would be a straightforward process with the OSF library we provide. One candidate application is the solution of the semi-discretized heat transfer problem

for optimal cooling of steel profiles [Ben04, Pen06, BCO91], using model reduction techniques [ASG01, BG06, GAB08]. The solution of this problem requires repeatedly solving the equation below using different optimal interpolation points (σ):

$$(\sigma E - A)x = b.$$

Here, $A, E \in \Re^{n \times n}$ and $b, x \in \Re^n$, where σ is scalar. With operation stacking, multiple solutions with different σ values can be computed simultaneously, in a single ensemble run. Note that varying σ affects only the numerical values of matrix elements, whereas the matrix structure remains identical, as required by the OSF.

Expanding the OSF library with new iterative solvers (short-term). Although the OSF developer library provides an easy way to convert existing solvers, we believe OSF will find a wider application area if it supports a variety of commonly used solvers out of the box. Solvers that could be stacked include stationary iterative methods such as the Jacobi, Gauss Seidel and Successive Overrelaxation (SOR); and non-stationary methods such as Minimal Residual (MINRES), Symmetric LQ (SYMMLQ), variants of CG on the normal equations (CGNE and CGNR), Biconjugate Gradient (BiCG), Quasi-Minimal Residual (QMR), Conjugate Gradient Squared (CGS), Biconjugate Gradient Stabilized (bi-CGST) and Chebyshev [BBC+94]. In addition, several other common linear algebra operations such as Sparse Triangular Solve (SpTS), sparse matrix-transpose matrix multiply ($A^T A \cdot x$) and sparse matrix powers ($A^p \cdot x$), which are studied by Vuduc [Vud03], are suitable for stacking with OSF.

Evaluation of multiple right hand sides (short-term). Sparse computations on problems with multiple right hand sides (RHS) are known to achieve significantly better performance compared to the single RHS case [IYV04, VDY05].

Our current evaluation explores problems with single RHS case only, however OSF carries the potential to achieve better performance if stacked problems include multiple RHS as well. A study to evaluate RHS case could be completed short term using the OSF library, which should require no modifications for this task.

OSF support for parallel applications (long-term). The current OSF implementation is sequential. We expect that OSF would retain performance benefits on multicore systems as well, because the primary performance limiter of SMVM on today's multicore machines is the shared use of memory bandwidth, which OSF optimizes. In fact, the current OSF implementation could already allow for a multithreaded run, because the leader process can use existing cores to perform a stacked SMVM operation in parallel. The stacked algorithm could be written using either the OpenMP language, or with the thread pool we developed for the parallelization of PBR and CSR (Section 4.3.3).

In a preliminary study, we performed a set of experiments using MPI to investigate the potential benefit of OSF on distributed systems [BR05]. We show that stacked operations can lead to more effective communication by reducing the latency and using the intercommunication network more efficiently, since stacking creates longer strides of data to be exchanged among nodes. Multiple processes of a typical OSF run could be replaced with multiple MPI instances, however OSF library does not provide support to achieve collaboration amongst MPI instances at the moment.

Re-writing the OSF library using C++ (long-term). We have designed OSF with an object oriented approach in mind, but the current implementation uses the C language. The algorithm independent OSF iteration engine operates on abstract classes and objects (Section 3.1.3), which could benefit tremendously from the polymorphism feature of C++. Our current implementation mimics this behavior using function and data pointers, which requires extra book keeping. C++ also allows *template metaprogramming* to generate classes at runtime, which could be creatively used for more flexible code generation than the C macros we use in the current implementation. This reimplementation would benefit the readability and maintainability of the software; we do not expect any performance difference from using C++.

6.2 Pattern-based Representation

Evaluation of non-square block size combinations (short-term). Our current evaluation of PBR assumes square block sizes ranging from 2×2 to 8×8. Our experience with modeling PBR performance (Section 4.2.2) showed that memory accesses patterns significantly vary with block sizes, strongly influencing performance. Therefore, it would be interesting to try non-square combinations, especially to see the difference between row-wide ($R \times C, R < C$) and column-wide ($R \times C, R > C$) blocks. In addition, some matrices could achieve better coverage using non-square blocks, thus yielding better PBR performance.

The main challenge in using non-square block sizes is the number of combinations we would need to analyze, which could potentially increase the matrix analysis cost.This cost can be reduced considerably by analyzing the matrix in one pass using 840-row at a time and deriv-

ing statistics for small patterns from their parent patterns, as we describe in Section 4.2.1. One caveat is that the increased number of hash maps to keep structural information may lead to extensive memory consumption for very large matrices. In addition, each block size combination would need a separate fit of our performance predictor model.

Autotuning for vectorization and prefetching (short-term). In our current evaluation, we run prefetched and vectorized PBR codes separately and report their performance if PBR benefits from these optimizations. A more desirable approach would be a runtime autotuning mechanism to detect whether vectorization and software prefetching provide benefit, and also to determine an optimal prefetch distance if prefetching is beneficial. In our experience, a vectorized kernel is most likely to benefit from prefetching if its non-vectorized version does. Therefore, prefetching should precede vectorization during autotuning. An exhaustive search for the optimal prefetch distance in a range between 0 to 1024 bytes could be completed in 32 steps, using 32-byte increments.

Revisiting the PBR inclusion criteria (short-term). As described in Section 4.1.2, we exclude from PBR those blocks that contain less than three nonzeros, and patterns that cover less than one thousand nonzeros throughout the matrix. The former criterion is deterministic, since blocks with two nonzeros do not provide memory bandwidth reduction savings compared to the CSR method, independent of the matrix size. The latter criterion, however, is an empirical value that we set heuristically.

Exploration of this heuristic criterion would constitute an interesting short-term study. Is this value a good choice for all matrices? If not, is there an analytical or experimental way to identify this value for varying architectures and matrix structures? Considering Amdahl's

law [Amd67], a carefully tuned threshold value could significantly improve performance by increasing PBR nonzero coverage.

Multithreaded matrix analyzer/Code generator (short-term). The current implementation of the PBR library performs the matrix analysis, structure creation, and code generation/compilation steps sequentially. Each of these steps is amenable to parallelization. The matrix analysis step could be parallelized by assigning each thread a 840-row partition in a round robin fashion, and merging the hash maps kept by each thread at the end of operations. The structure could also be created in parallel, because the starting address of blocks for each pattern is predetermined in the matrix analysis step. Threads need to synchronize only once to determine the offsets they need to add to these addresses, then they can fill in the PBR representation in parallel. The code generation and compilation could also be performed in parallel.

A network code repository (short-term). Our overhead analysis in Section 4.3.4 shows that code generation incurs the most significant cost by a large margin. We show in Section 4.4.2 that a local code repository, assuming high hit rates, could largely address this problem. On the other hand, given today's storage capacities, no local repository could store *all* of the $2^{R \times C}$ codes for large block sizes to *completely* eliminate the code compilation step.

However, it takes considerably less time to download precompiled code than to generate and compile its source. Therefore, a distributed online repository could provide a scalable storage for object files. This repository could be at an institution level, e.g., university-wide, or at Internet level with contributions by volunteers. One important caveat is that uncontrolled distribution of binaries could raise security concerns.

Assembly code generators to avoid code compilation (long-term). Another solution for the code compilation overhead would be the generation of custom codes in machine language. Although difficult, it is possible to manually write highly efficient computational kernels in assembly, as done in the GOTO BLAS library [GvdG02]. On the other hand, this task requires considerable expertise and human intuition, therefore it is a not a trivial task to develop a code generator for the same purpose. In addition, such codes would not be portable because machine code is architecture dependent. Despite its challenges, this improvement would completely eliminate the need for code compilation and use of code repositories.

A benchmark for determining the predictor model parameters (long-term). In Section 4.2.2 we describe a performance predictor model to predict an optimal block size without actually running any SMVM operations. Our model uses a multiple linear regression model, whose parameters are predetermined by experimentation for different architectures. The model parameters we report in Section 4.3.5 were obtained for two recent AMD and Intel architectures, using the matrix set we use in our evaluation.

PBR still lacks a benchmark suite, which would provide users with these model parameters for *any* given platform that it is run on, however. The main challenge lies in the selection of experimental matrices. In our evaluation, we included matrices from a wide variety of sizes and structures, thus we achieved high accuracy. However, the same approach may not be practical for a benchmark suite because (1) our matrix set includes large matrices, requiring gigabytes of data to be downloaded, and (2) the benchmarking process for such large matrices would last very long, i.e., longer than what is acceptable as an installation step. An ideal benchmark must complete in reasonable time, while representing a wide variety of matrix structures and sizes. Gahvari et al. [GHDY07] describes a methodology

for benchmarking any given SMVM algorithm in five minutes. These authors argue that (1) structural properties of a large variety of sparse matrices can be mimicked using synthetic matrices that are generated at runtime, requiring no download; (2) their performance can be evaluated in minutes by benchmarking only a small portion of problem sizes to reduce the runtime to a small fraction of the original runtime. The approach in this work could be adopted to benchmark PBR and determine the predictor model parameters for each candidate block size quickly and accurately.

Integration of PBR into widely used mathematical software (long-term). An effective way to have a new technique adopted by large numbers of users is to deliver it as a component of a widely used package. One example is the integration of OSKI [VDY05] into the PETSc [BB+08] package. Since most users see software packages as black boxes, and cannot be expected to provide tuning parameters for using PBR, e.g., about the block size and optimization parameters, PBR must use autotuning. PETSc, MATLAB [GMS92], Mathematica [Mat] and R [R D10] are examples of widely used mathematical software that could benefit from PBR for a faster SMVM.

Chapter 7

Conclusion

The primary intellectual contribution of this dissertation is to show that **recurring nonzero patterns, either in matrices or matrix blocks, can be exploited to significantly improve the performance of sparse computational kernels.** We presented two new techniques, Pattern-based Representation (PBR) and the Operation Stacking Framework (OSF), both of which are motivated by, and rely on this assertion.

Our primary focus is on the Sparse Matrix Vector Multiply (SMVM) operation because this important kernel constitutes a large fraction of the runtime of iterative solvers, e.g., CG and GMRES (see Chapter 1). Achieving high performance with SMVM kernels has historically been a challenging task. Consequently, there has been extensive research in this area, most of which exploits dense block structures to improve performance (see Section 5.2). Unfortunately, it is not a trivial task to detect such blocks, moreover, they may not even exist in a given matrix. Zero-filling techniques can be used to construct dense blocks, but

this approach may easily lead to inefficient computations as the number of filled-in zeros grows.

The primary strength of our approach is that neither of our methods (OSF and PBR) makes any assumptions about nonzero structures, other than that they are *recurring*. If an entire matrix structure recurs among multiple problems, we solve these problems simultaneously using the Operation Stacking Framework (OSF, Chapter 3). If the recurrence is at the block-level, then we represent the matrix using the Pattern-based representation (PBR, Chapter 4), and generate custom codes for each recurring block pattern.

To address the large performance gap created by the memory wall, which is demonstrated in Figure 2.3, both of our methods reduce the memory bandwidth usage of SMVM operations. In OSF, we do so by storing only one copy of the indexing information, which is identical for all participating problems. In PBR, we generate custom codes for each of the sufficiently recurring block patterns, so that indexing information for nonzeros in those blocks does not need to be repeatedly read from memory. The compiled codes are fetched from memory only once, and not evicted from the instruction cache until they process all of the blocks that share the same pattern.

The benefits of our methods are not limited to memory bandwidth. OSF further improves performance by interleaving stacked data for better cache utilization and by unrolling stacked loops, making them larger to improve ILP. PBR improves performance by utilizing software prefetching, SSE vectorization, and loop unrolling. As a result, we show with an extensive evaluation that both of our techniques achieve better performance than the fastest available methods, namely CSR and the OSKI library. We show that, for sequential runs, OSF

can provide a speedup of up to 1.94× and PBR can provide a maximum speedup of 3.4×, compared to the CSR method. For its parallel implementation, PBR provides a speedup of up to 5× over parallel CSR on recent multicore architectures.

We also described in detail the implementations of the OSF and PBR libraries, which constitute a large portion of our effort. OSF provides users with ready-to-use stacked iterative solvers, which require minimal changes to existing codes. Without these stacked solvers, it would be a challenging task to achieve the system-level coordination and synchronization among multiple stacked problems. In addition, we developed the OSF developer library to aid developers in stacking any given solver algorithm. This library requires only simple function and data templates to emit stacked versions of an iterative solver, which are generated at compile time by a set of C macros.

Similarly, we developed the PBR library to facilitate the PBR technique. This library completes all necessary steps to convert a matrix into the PBR format and generates custom codes without involvement by the user. This conversion may take place at runtime if desired (see Section 4.2.1). We aggressively optimized the PBR library to incur low overhead with a time complexity linear in the number of nonzeros in the matrix. The code generator of the PBR library can automatically emit highly optimized codes that exploit several techniques including loop unrolling, prefetching, and vectorization. These custom codes are then stored in a code cache to avoid the high cost of repetitive generation of codes for the same patterns.

The performance problem for memory bound irregular applications becomes more acute on multicore platforms, due to the shared use of available memory bandwidth by multiple cores [WOV+09]. Compounding the problem, many multicore systems use nonuniform

memory access (NUMA) architectures and non-shared high-level caches with cache coherence, which lead to significant access latencies particularly for applications involving frequent thread/process synchronization and collective operations. Although we do not know their design yet, future *extreme-scale* platforms will likely be composed of a massive number of heterogeneous components, making them increasingly memory bound [Dea10, DBA+09, HEC08, ABC+06, HVB+09, KBB+08, SZS08]. For these reasons, we expect that OSF and PBR will retain their benefits in the long term and could be adopted by a wide range of engineering and scientific applications.

Bibliography

[ABB+90] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 2–11, New York Hilton at Rockefeller Center, New York, New York, 1990. IEEE Computer Society Press.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[AFR98] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. *International Parallel Processing Symposium*, 0:298–302, 1998.

[AGK+99] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD applica-

tion. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 1999. IEEE Computer Society Press.

[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: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 32–41, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Amd67] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing requirements. *Computer Design*, 6(12):39–40, 1967.

[AMD09] AMD. *Software Optimization Guide for AMD Family 10h Processors*. Advanced Micro Devices Inc., May 2009.

[Ame94] American National Standards Institute. *IEEE standard for information technology: Portable Operating System Interface (POSIX)*. IEEE Computer Society Press, 1994.

[ASG01] A.C. Antoulas, D.C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. In *Structured Matrices in Operator Theory, Numerical Analysis, Control, Signal and Image Processing, Contemporary Mathematics*, number 280, pages 193–219. AMS Publications, 2001.

[BB87] M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, C-36(5):570 –580, May 1987.

[BB+08] Satish Balay, Jed Brown, , Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[BBC+94] R. Barrett, M. Berry, T. Chan, J. W. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems*. SIAM, 1994.

[BBR09] Mehmet Belgin, Godmar Back, and Calvin J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In Michael Gschwind, Alexandru Nicolau, Valentina Salapura, and José Moreira, editors, *ICS'09: Proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, June 8-12, 2009*, pages 100–109. ACM, 2009.

[BCO91] Laurent Baratchart, Michel Cardelli, and Martine Olivi. Identification and rational l2 approximation: a gradient algorithm. *Automatica*, 27(2):413–418, 1991.

[BCOM+10] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italy, 02 2010.

[BDH+00] S. Browne, Jack Dongarra, G. Ho, N. Garner, and P. Mucci. A portable pro-

gramming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[Ben04] Peter Benner. Solving large-scale control problems. *Control Systems Magazine, IEEE*, 24(1):44–59, Feb. 2004.

[BFGS05] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 171, New York, NY, USA, 2005. ACM.

[BG97] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Advances in Engineering Software*, 28(3):189–201, 1997.

[BG06] C.A. Beattie and S. Gugercin. Inexact solves in krylov-based model reduction. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 3405–3411, December 2006.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[BHZ93] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical report, Carnegie-Mellon University, 1993.

[Bik96] Aart Johannes Casimir Bik. *Compiler support for sparse matrix computations.* PhD thesis, Rijksuniversiteit te Leiden, 1996.

[Boo] *BOOST C++ Libraries.* `http://www.boost.org`.

[BR05] Mehmet Belgin and Calvin J. Ribbens. Improving the performance of HPC applications by using operation stacking. In H. Dag and Y. Deng, editors, *Proceedings of ICCSE2005, The International Conference on Computational Science and Engineering*, pages 183–190. Istanbul Technical University, 2005.

[BRB07] Mehmet Belgin, Calvin J. Ribbens, and Godmar Back. An operation stacking framework for large ensemble computations. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, pages 83–92, New York, NY, USA, 2007. ACM.

[BW93] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 416–424. ACM Press, 1993.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM Press.

[CSV10] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 115–126, New York, NY, USA, 2010. ACM.

[Dav] Tim Davis. The University of Florida sparse matrix collection. Submitted to ACM Transactions on Mathematical Software.

[DBA+09] Jack Dongarra, Pete Beckman, Patrick Aerts, Franck Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo Valero. The International Exascale Software Project: A Call to Cooperative Action by the Global High-Performance Community. *International Journal of High Performance Computing Applications*, 23:309–322, November 2009.

[DCHD90] J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[DDD+06] J. Demmel, J. Dongarra, J. Ding, V. Eijkhout, D. Keyes, S. Li, B. Smith, and R. Vuduc. Performance optimization of sparse kernels in TOPS. URL, 2006. http://www.scidac.gov/ASCR/ASCR_TOPS/reports/TOPS2003Annual1.html (last accessed in 2010).

[DDSvdV98] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998.

[Dea10] Jack Dongarra and Pete Beckman et al. The international exascale software project roadmap. Technical Report UT-CS-10-654, University of Tennessee, May 2010. EECS Technical Report.

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, May 1968.

[Den05] Peter J. Denning. The locality principle. *Communications of the ACM*, 48:19–24, July 2005.

[DFM05] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science*, pages 99–106, Singapore, May 2005.

[DK99] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, New York, NY, USA, 1999. ACM Press.

[DLPR94] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, pages 214–218, 1994.

[DMBS86] Dongarra, Moler, Bunch, and Stewart. *Linpack*. Philadelphia, 1986.

[DMS+92] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*, 1992.

[GAB08] S. Gugercin, A. C. Antoulas, and C. Beattie. $\mathcal{H}_2$ model reduction for large-scale linear dynamical systems. *SIAM Journal on Matrix Analysis and Applications*, 30(2):609–638, 2008.

[GHDY07] Hormozd Gahvari, Mark Hoemmen, James Demmel, and Katherine Yelick. Benchmarking sparse matrix-vector multiply in five minutes. In *SPEC Benchmark Workshop*, Austin TX, January 2007.

[GKA+08] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 283–292, Washington, DC, USA, 2008. IEEE Computer Society.

[GKKS99] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer, editor, *Proceedings of Parallel CFD'99*. Elsevier, 1999.

[GKY79] R. G. Grimes, D. R. Kincaid, and D. M. Young. ITPACK 2.0 User's Guide'. Tech. Report CNA-150, Univ. of Texas, Austin, 1979.

[GLW70] F. G. Gustavson, W. Liniger, and R. Willoughby. Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. *J. ACM*, 17(1):87–109, 1970.

[GMS92] J. R. Gilbert, C. B. Moler, and R. Schreiber. Sparse matrices in MATLAB : Design and implementation. *SIAM J. Matrix Anal. and Appl.*, 13(1):333–356, 1992.

[GPZ00] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In

ICS '00: Proceedings of the 14th International Conference on Supercomputing, pages 78–87, New York, NY, USA, 2000. ACM.

[GR01] Roman Geus and Stefan Röllin. Towards a fast parallel sparse symmetric matrix-vector multiplication. *Parallel Computing*, 27(7):883 – 896, 2001.

[GR05] Tilmann Gneiting and Adrian E. Raftery. Weather forecasting with ensemble methods. *Science*, 310(5746):248, 2005. Washington, D.C.

[GSB+06] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[Gsc06] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006. ACM.

[GV89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, 2nd. Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[GvdG02] Kazushige Goto and Robert van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, The University of Texas at Austin, Department of Computer Sciences, November 1 2002.

[HBH+03] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps,

Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[HCK93] Parviz Moin Haecheon Choi and John Kim. Direct numerical simulation of turbulent flow over riblets. *Journal of Fluid Mechanics*, 255:503–539, 1993.

[HEC08] *The Potential Impact of High-End Capability Computing on Four Illustrative Fields of Science and Engineering*. The National Academies Press, Washington, DC, USA, 2008.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HT98] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 393, Washington, DC, USA, 1998. IEEE Computer Society.

[HT00] Hwansoo Han and Chau-Wen Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 26(13-14):1861–1887, 2000.

[HT06] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006.

[HVB+09] K. Scott Hemmert, Jeffrey S. Vetter, Keren Bergman, Chita Das, Azita Emami, Curtis Janssen, Dhabaleswar K. Panda, Craig Stunkel, Keith Underwood, and Sudhakar Yalamanchili. IAA Interconnection Networks Workshop 2008. Technical Report FTGTR-2009-03, Future Technologies Group, Oak Ridge National Laboratory, April 2009. http://ft.ornl.gov/pubs-archive/iaa-ic-2008-workshop-report-final.pdf.

[IBM88] IBM. *Engineering and Scientific Subroutine Library (ESSL) "Guide and Reference"*, 1988. (Release 3, Order SC-23-0184).

[ID92] John Lewis Iain Duff, Roger Grimes. User's guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, October 1992.

[INT06] INTEL. *IA-32 Intel Architecture Optimization Reference Manual*. Intel, April 2006.

[INT09a] INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, November 2009.

[INT09b] INTEL. *Intel Math Kernel Library for Linux OS*, March 2009. 314774-009US.

[IY01] Eun-Jin Im and Katherine A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 127–136, London, UK, 2001. Springer-Verlag.

[IYV04] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[JKK+99] Mahesh V. Joshi, George Karypis, Vipin Kumar, Anshul Gupta, and Fred G. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[JN07] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit power efficient SPARC SOC Niagara2). In *ISPD '07: Proceedings of the 2007 international symposium on Physical design*, pages 2–2, New York, NY, USA, 2007. ACM.

[KBB+08] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology challenges in achieving exascale systems, September 2008. http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm (last accessed in 2010).

[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.

[KLvL98] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302, 1998.

[KP97] Vladimir Kotlyar and Keshav Pingali. Sparse code generation for imperfectly nested loops with dependencies. In *11th ACM International Conference on Supercomputing*, pages 23–30, 1997.

[KPS97a] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel code for sparse matrix applications. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1997. ACM.

[KPS97b] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel sparse code for user-defined data structures. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM Press, 1997.

[KPS97c] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of the International European Conference on Parallel and Distributed Computing*, pages 318–327, 1997.

[KY84] David R. Kincaid and David M. Young. The ITPACK project: Past, present, and future. In Garrett Birkhoff and Arthur Schoenstadt, editors, *Elliptic Problem Solvers II*, pages 53–63. Academic Press, New York, NY, USA, 1984.

[LS76] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[LVDY04] Benjamin C. Lee, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. Performance models for evaluation and automatic tuning of symmetric

sparse matrix-vector multiply. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing*, pages 169–176, Washington, DC, USA, 2004. IEEE Computer Society.

[Mat] *Mathematica*. Wolfram Research, Inc., Champaign, IL, USA. `http://www.wolfram.com` (last accessed in 2010).

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[MCG04] John Mellor-Crummey and John Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *International Journal of High Performance Computing Applications*, 18(2):225–236, 2004.

[MCWK01] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[Mel87] Rami Melhem. Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *International Journal of Supercomputer Applications*, 1(1):70–98, Spring 1987.

[MHW04] H. Merlitz, T. Herges, and W. Wenzel. Fluctuation analysis and accuracy of a large-scale in silico screen. *Journal of Computational Chemistry*, 25(13):1568, October 2004.

[MMB08] Rajesh Bordawekar Muthu Manikandan Baskaran. Optimizing sparse matrix-vector multiplication on GPUs. Technical Report RC24704, IBM, 2008.

[MSDS10] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Host Simon. Top500 supercomputing sites, November 2010. http://www.top500.org/ (last accessed in 2010).

[NVDY04] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical report, Berkeley, EECS Dept., 2004.

[NVDY07] Rajesh Nishtala, Richard Vuduc, James Demmel, and Katherine Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18:297–311, 2007. 10.1007/s00200-007-0038-9.

[ORF96] Chao–wei Ou, Sanjay Ranka, and Geoffrey Fox. Fast and parallel mapping algorithms for irregular and adaptive problems. *Journal of Supercomputing*, 10:119–140, 1996.

[OSK10] OSKI website, 2010. http://bebop.cs.berkeley.edu/oski/, last accessed 2010.

[PCM98] T.J. Hanratty P. Cherukat, Y. Na and J.B. McLaughlin. Direct numerical simulation of a fully developed turbulent flow over a wavy wall. *Theoretical and Computational Fluid Dynamics*, 11(2):109–134, May 1998.

[PDZ92] S. C. Park, J. P. Draayer, and S.-Q. Zheng. An efficient algorithm for sparse matrix computations. In *SAC'92: Proceedings of the 1992 ACM/SIGAPP*

Symposium on Applied Computing, pages 919–926, New York, NY, USA, 1992. ACM.

[Pen06] Thilo Penzl. Algorithms for model reduction of large dynamical systems. *Linear Algebra and its Applications*, 415(2-3):322 – 343, 2006. Special Issue on Order Reduction of Large-Scale Systems.

[PH99] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing'99*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

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

[R D10] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[RB84] John R. Rice and Ronald F. Boisvert. *Solving elliptic problems using ELLPACK*. Springer-Verlag, New York, NY, USA, 1984.

[RP96] Karin A. Remington and Roldan Pozo. NIST sparse BLAS user's guide. Technical report, National Institute of Standards and Technology, July 19 1996.

[Saa88] Saad, Y. Krylov subspace methods on supercomputers. Tech. Report 19, Research Institute for Advanced Computer Science, Moffett Field, 1988.

[Saa94] Yosef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Tech-

nical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[Saa96] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, 1996.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[STL] *Standard Template Library Programmer's Guide*. `http://www.sgi.com/tech/stl/` (last accessed in 2010).

[SZS08] Horst Simon, Thomas Zacharia, and Rick Stevens. Modeling and simulation at the exascale for energy and the environment. Technical report, Office of Science, U.S. Dept. of Energy, May 2008. http://www.sc.doe.gov/ascr/ProgramDocuments/Docs/TownHall.pdf (last accessed in 2010).

[THHS99] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Official Aztec user's guide version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.

[TJ92] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 578–587, 1992.

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

[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[VKH+02] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W. Demmel, and Katherine A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, 2002.

[VM05] Richard W. Vuduc and Hyun-Jin 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. Editors: Laurence Tianruo Yang and Omer F. Rana and Beniamino Di Martino and Jack Dongarra.

[Vud03] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, 2003.

[WL06] Jeremiah Willcock and Andrew 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.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[WOV+07] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick,

and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[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.

[WPD01] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[WR71] J. H. Wilkinson and C. Reinsch, editors. *Linear Algebra*, volume II of *Handbook for Automatic Computation*. Springer-Verlag, Berlin, 1971.

[WS97] James B. White, III and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing*, pages 66–71, Washington, DC, USA, 1997. IEEE Computer Society.

[YR00] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *ICS '00: Proceedings of the 14th International Conference on Supercomputing*, pages 66–77, New York, NY, USA, 2000. ACM.

Appendix A

Stacking Iterative Solvers Using OSF

In this appendix, we explain the process of stacking iterative solvers in more detail and demonstrate the stacking process for the initialization part of the CG algorithm as an example. The main goal of this example is to provide developers with a road map, since fundamental steps for conversion remain almost identical for stacking other iterative solvers.

As described in Section 3.1.3, iterative solvers commonly include initialization and iteration sections. As the first step of conversion, we identify these sections and separate them from each other. Figures 3.5 and 3.6 outline these sections for CG and GMRES. Both initialization and iteration steps include a SMVM operation (steps 1 and 10 for CG, and steps 2 and 8 for GMRES), which will be stacked for better performance. As discussed in Section 3.1.3, the iteration is completely managed by the OSF iteration engine (OSFIE). Therefore, developers only provide a function that represents the iteration loop body, not including the iteration itself. For this reason, *for* and *while* directives are excluded from the iteration section as

seen in Figures 3.5 and 3.6.

We first explain some of the fundamental components of the OSF package. Then we show how stacked solvers are used in existing codes. Then, we explain the use of OSF Developer Tools to create these stacked solvers. Finally, we demonstrate each step of the conversion from a regular solver into a stacked solver in detail, using the CG algorithm as an example.

A.1 Components of the OSF Package

We first look at the structure of the OSF package, which is a collection of source files, headers, libraries, Makefiles and scripts to aid operation stacking. OSF includes the following source files:

- `osf_developer_library.c`
- `osf_initialization.c`
- `osf_main.c`
- `osf_user_library.c` (modified by developer)

and header files:

- `osf.h`
- `osf_solvers.h` (modified by developer)
- `osf_stackedaction.h`

An OSF-provided Makefile compiles the OSF package and creates the end-user library (`libosfusr.a`), which includes ready-to-use stacked versions of iterative solvers, and the de-

veloper library (`libosfdev.a`), which includes tools and macro definitions to assist stacked solver developers.

To create and compile a new stacked iterative solver, developers modify `osf_developer_library.c` and `osf_user_library.c`, and also the header file `osf_solvers.h`, which keeps the function prototypes to be included by end users. The remaining files are specific to the OSF implementation; they need neither modification nor recompilation when new solvers are added to the end user library.

A.2 Use of Stacked Solvers

The completed stacked solvers can easily be incorporated into existing codes by end users using these steps:

1. Replace the solver function call with its OSF version, leaving the arguments as they are (see Section 3.2).
2. Include `osf_solvers.h` in code.
3. Compile code by linking to `libosfusr.a`.
4. Invoke the executable by using OSF-provided *osfrun* script.

Figure A.1 shows the *osfrun* script, which is responsible for handling multiple tasks. First, it ensures that the shared memory areas created by the processes this script initiates are

```
[ 1] #!/bin/bash
[ 2] # System specific OSF path
[ 3] # Could also be provided in environmental variables
[ 4]
[ 5] OSFROOT=../OSF
[ 6] OSFLIBDIR=${OSFROOT}/libs
[ 7] OSFBINDIR=${OSFROOT}/bin
[ 8]
[ 9] if [ $# -lt 3 ]; then
[10]     echo Usage: $0 [-o OUTPUTPREFIX] STACKDEPTH STACKEDCODE ARGUMENTS
[11]     exit;
[12] fi
[13]
[14] # Delete all shared memory areas when this shell exits
[15] # $$ is the current pid of the shell, which is the parent pid
[16] # of the stacked programs.
[17] # This statement replaces osf_terminate, assuming that all
[18] # shared memory areas are named *_PID_*.
[19]
[20] trap "/bin/rm -f /dev/shm/*_$$_*; pkill -P $$" EXIT
[21]
[22] if [ $1"x" = "-ox" ]; then
[23]     shift;
[24]     OUTPUTPREFIX=$1
[25]     shift;
[26] fi
[27]
[28] STACKDEPTH=$1
[29] shift
[30] PROG=$1
[31] shift
[32] FILE=$*
[33]
[34] ${OSFBINDIR}/osf_initialization ${STACKDEPTH}
[35]
[36] for I in `seq 1 ${STACKDEPTH}`
[37] do
[38]    if [ ${OUTPUTPREFIX}"x" = "x" ]; then
[39]        ./${PROG} ${FILE} &
[40]    else
[41]        ./${PROG} ${FILE} >& output.$I &
[42]     fi
[43] done
[44]
[45] wait
```

Figure A.1: `osfrun` script.

cleaned, even if the stacked operation prematurely terminates with an error, using a bash trap handler (line [20]). Second, it prepares the system for the stacked operations that will follow by allocating OSF-specific resources, such as shared memory and semaphores, and issuing a unique stack ID (line [34]). The stack ID prevents conflicts when multiple stacked operations are being run simultaneously on the same system. This initialization is a one-time operation, which is independent of the algorithms being stacked, and should not be confused with the initialization section of iterative algorithms. Third, it launches processes by running multiple instances of the same executable simultaneously (lines [36] to [43]), and exits when all processes are finished running. End users run this script as follows:

`osfrun` *`stackdepth executable arguments`*

Where *`stackdepth`* is the desired initial stack depth, *`executable`* is the name of the executable and *`arguments`* are the application-specific argument(s), which are normally passed to the non-stacked version of the application. Optionally, users can direct the output to a file using the '-o' flag.

A.3 OSF Developer Library Macros

Because stacked codes are kept in unrolled form that are hardwired to a particular stack depth, a stacked solver must include functions for each possible stack depth. The OSF developer library provides a set of macros to shield developers from the burden of preparing multiple versions of stacked solvers. Developers annotate the code sections that need replication using these macros and leave the necessary code replication to OSF, which processes

Table A.1: OSF-provided macros for stacking operations.

Name	**Explanation**
`STACKED_OPERATION(OPERATION)`	For operations that can be stacked in their own block scope.
`STACKED_OPERATION_SAME_BLOCK(OPERATION)`	For operations that must be stacked in the surrounding block's scope.
`STACKED_ASSIGNMENT(LHS, RHS)`	For assigning a LHS operation to a RHS operation (LHS = RHS).
`STACKED_ASSIGNMENT_CONST(LHS, c)`	For assigning a LHS operation to a constant c (LSH = c).
`STACKED_OPERATION_WITH_OPERATOR(OPERATION, op)`	For binding stacked operations with an operator (mostly used in convergence check, e.g., criteron1 && criterion2 && ...).

these templates at compile time to generate the final code.

Table A.1 provides a list of macro functions that are used for code replication. Three of these macros take `OPERATION` as input, which is the user-defined operation that needs to be stacked. The `OPERATION` itself also needs to be defined as a macro. For example, the following statement

```
#define EXAMPLEFUNCTION(_si) x(_si) = y(_si);
STACKED_OPERATION_SAME_BLOCK(EXAMPLEFUNCTION)
```

creates code equivalent to:

```
x(0) = y(0);
x(1) = y(1);
```

```
x(2) = y(2);
x(3) = y(3);
```

for a stack depth (*_si*) of four.

OSF also provides a macro, `_IL_(`*array, arrayindex, stackdepth, problemindex*) to conveniently access an interleaved array. This macro returns the element at *arrayindex* of problem # *problemindex* of an interleaved array *array* for stack depth *stackdepth*.

A.4 An Example: Stacking the CG Solver

This section demonstrates the steps to stack the CG solver as an example of using the OSF developer tools. As described in Section 3.1.3, developers stack a solver by providing an algorithm-specific implementation, or *template*, for each of the standard functions listed in Figure 3.7. These implementations are written using OSF-provided functions and macros for code generation and replication.

A.4.1 Initialization Step

We begin with stacking the initialization step of CG. The final stacked template, `void init_CG(int stackid, struct _stackstate *stackstate, void *algorithmdata)`, is given in Figure A.2, which corresponds to the `void(*init[MAX_STACKDEPTH])()` function in Figure 3.7. This function has three arguments: The *stackid* is the unique ID that identifies the current stacked operation. The second argument is a pointer to the struct *_stackstate*,

```
[ 1] void init_CG(int stackid, struct _stackstate *stackstate,  void *algorithmdata) {
[ 2]     struct _problemstate *problemstate = algorithmdata;
[ 3]     int i, k, k1, k2;
[ 4]
[ 5]     #define DEFINE_SUM(i) double sum##i;
[ 6]         STACKED_OPERATION_SAME_BLOCK(DEFINE_SUM)
[ 7]
[ 8]     int     n       = problemstate->n;
[ 9]     int     nnz     = problemstate->nnz;
[10]     int     *ia     = problemstate->ia;
[11]     int     *ja     = problemstate->ja;
[12]     double  *shm_aa = problemstate->shm_aa;
[13]     double  *shm_x  = problemstate->shm_x;
[14]     double  *shm_b  = problemstate->shm_b;
[15]
[16]     #define SUM_I(_si) sum##_si
[17]         STACKED_ASSIGNMENT_CONST(SUM_I, 0.);
[18]
[19]     for (i=0; i < n; ++i) {
[20]         k1 = ia[i] - 1;
[21]         k2 = ia[i + 1] - 2;
[22]         for (k=k1; k < k2 + 1; ++k) {
[23]             #define MULTaa_BY_X(_si) \
[24]             sum##_si += shm_aa[(_si * nnz) + k] * _IL_(shm_x, (ja[k] - 1), STACKDEPTH, _si)
[25]                 STACKED_OPERATION(MULTaa_BY_X);
[26]         }
[27]         #define MOVE_SUM_TO_R(_si) \
[28]         problemstate->shm_r[(_si * n) + i] = shm_b[i] - sum##_si
[29]              STACKED_OPERATION(MOVE_SUM_TO_R);
[30]
[31]         STACKED_ASSIGNMENT_CONST(SUM_I, 0.);
[32]     }
[33] }
```

Figure A.2: Stacked CG initialization function.

which includes OSF-specific information about active stacked operations. The third argument, *algorithmdata* is a void pointer to algorithm specific data, which is cast to a user defined struct, *problemstate*, in line [2]. This struct corresponds to the data structure given in Figure 3.9. However, since we implemented OSF using C, interleaved arrays are declared using the '`osf_register_interleaved()`' function (see Table 3.1) to mimic the functionality of template types in C++.

The first code section that needs to be replicated is the declaration of variables that are used in stacked operations: `double` ***`sum`***. Because we keep the inner loop unrolled, there must be one instance of this variable for each stack depth, i.e., {sum0, sum1, sum2, sum3}, for

a stackdepth of four. This is achieved using the `STACKED_OPERATION_SAME_BLOCK` macro, as explained in Section A.3:

```
#define DEFINE_SUM(_si) double sum##_si;
STACKED_OPERATION_SAME_BLOCK(DEFINE_SUM)
```

is translated into a stacked operation that declares *stackdepth* many double variables sum_{si}, creating code equivalent to:

```
double sum0;
double sum1;
double sum2;
double sum3;
```

These variables are initialized to zero by using `STACKED_ASSIGNMENT_CONST` in line [17].

Next, we define the multiply-add operation in the iteration in lines [23-25]. Line [24] makes use of the `_IL_` macro, which provides convenient access to the stacked and interleaved *shm_x* array by developers. Here, the `STACKDEPTH` is updated automatically to the current stack depth of operations by OSF, which keeps track of converged and ejected problems. Finally, the residual vector is initialized on lines [27-29] and *sum* variables are zeroed for the following iteration on line [31].

A.4.2 Iteration Step

The second function, `int iter_CG (int stackid, struct _stackstate *stackstate, void *algorithmdata)`, handles the iterations in the stacked CG algorithm. This function corresponds to the `void(*iter[MAX_STACKDEPTH])()` function in Figure 3.7, which takes the same three arguments as the initialization function: *stackid*, *_stackstate* and *algorithmdata*. The OSF iteration engine (OSFIE) repeatedly calls this function to perform iterations. The finalized code is given in Figure A.3. While stacking the CG init function (Section A.4.1), we demonstrated replication of operations, and accessing an interleaved array by using OSF macros. In this section, we demonstrate several other important OSF features, such as the custom convergence check, ejection of converged problems and accessing a non-interleaved stacked array using pointers.

Since stacked algorithms handle data of multiple problems, data structures must be adapted to maintain the initial and the remaining stack depths. For example, the scalar variables in the CG algorithm (Figure 3.5), namely ρ, α and β, become vectors of size 'current stackdepth' to store different values for each non-converged stacked problem. The initial stackdepth (`stackdepth`) and current stackdepth (`curstackdepth`) are found by using the `osf_get_stackdepth()` and `osf_get_curstackdepth()` functions (Table 3.1).

Two of the stacked, but non-interleaved arrays are the nonzero array (`aa`) and the residual vector (`r`). As illustrated in Figure 3.4, these arrays are accessed using pointers to data segments for non-converged problems. In the code, lines [22-26] declare these pointers using the `STACKED_OPERATION_SAME_BLOCK` macro. In lines [28,29], a stacked pointer (`si2rbase##_si[]`) is assigned to non-converged segments of the residual vec-

tor. The same operation is repeated for the non-stacked nonzero array in lines [59-60]. The `stackstate->c2i[]` is a lookup table internal to OSF, which translates current problem indices to initial problem indices. Similarly, OSF keeps a second lookup table, `stackstate->i2c[]`, which translates initial problem indices to current problem indices. For example, the 4^{th} problem in Figure 3.4 has the initial index 3, but is later assigned the index 1 (zero-based indexing) after two of the problems converge.

Lines [34-38] perform the dot product of `r` by itself, to find ρ as depicted in line [3] of the CG algorithm (Figure 3.5). It uses the stacked pointer instead of the vector `r`, where `_si` is the stacked index. If the algorithm is in the first iteration, the vector `p` must be initialized to `r` as depicted in line [5] in the CG algorithm. However, since `p` is used in the SMVM operation in line [10] of the CG algorithm, it is kept in interleaved form. Line [43] of the stacked code demonstrates the assignment of a non-interleaved array `r` to an interleaved array `p` using the `_IL_` macro of OSF. This macro is also used in line [54] of the code to perform the addition operation in line [8] of the CG algorithm.

The stacked CSR SMVM kernel, which consumes the largest fraction of the CG solver's runtime (Figure 1.1), is shown in lines [66-77] of the code. In line [71], the non-interleaved `A` array is multiplied by the interleaved `p` array and the non-interleaved `q` array is updated with the result. This line demonstrates the use of the `_NI_` macro, which returns the index of an element in a stacked, but non-interleaved array. The expanded stacked code includes *stacked depth* many copies of this line.

Lines [101-108] demonstrate the algorithm-specific convergence check. When convergence is detected, the corresponding problem is removed from the stack using the

`osf_remove_from_stack[]` function, which marks the converged problem(s) to be ejected. Finally, the function returns the number of converged problems to the OSFIE, which is the caller of this function as depicted in Figure 3.8. The OSFIE performs all necessary data compression and rearrangement (as discussed in Section 3.1.2) using the `eject_converged_problems()` function.

```
[ 1] int iter_CG (int stackid, struct _stackstate *stackstate,  void *algorithmdata) {
[ 2]     struct _problemstate *problemstate = algorithmdata;
[ 3]     int    n       = problemstate->n;
[ 4]     int    nnz     = problemstate->nnz;
[ 5]     double delta   = problemstate->delta;
[ 6]     float   *x     = problemstate->shm_x;
[ 7]     float   *p     = problemstate->shm_p;
[ 8]     int     *ia    = problemstate->ia;
[ 9]     int     *ja    = problemstate->ja;
[10]     float   *aa    = problemstate->shm_aa;
[11]     float   *r     = problemstate->shm_r;
[12]     float   *oldro = problemstate->shm_oldro;
[13]
[14]     int curstackdepth = osf_get_curstackdepth(stackid);
[15]     int stackdepth    = osf_get_stackdepth(stackid);
[16]     float q[n * curstackdepth], ro[curstackdepth], alpha[curstackdepth];
[17]     float beta[curstackdepth], criteria[curstackdepth], product[curstackdepth];
[18]     int i, j;
[19]     int k, k1, k2, rank;
[20]     int iter = (*stackstate->iter);
[21]
[22]     #define DEFINE_AABASE(i) float *si2aabase##i;
[23]         STACKED_OPERATION_SAME_BLOCK(DEFINE_AABASE)
[24]
[25]     #define DEFINE_RBASE(i) float *si2rbase##i;
[26]         STACKED_OPERATION_SAME_BLOCK(DEFINE_RBASE)
[27]
[28]     #define RBASE(_si) si2rbase##_si = r + stackstate->c2i[_si] * n
[29]         STACKED_OPERATION(RBASE);
[30]
[31]     for (i = 0; i < curstackdepth; i++)
[32]         ro[i] = 0.0;
[33]
[34]     for (i = 0; i <  n; ++i) {
[35]         #define RO_IS_R_SQR(_si) \
[36]             ro[_si] += (si2rbase##_si[i] * si2rbase##_si[i]);
[37]         STACKED_OPERATION(RO_IS_R_SQR);
[38]     }
[39]
[40]     if (iter == 1) {
[41]         for (i = 0; i < n; i++) {
[42]             #define P_IS_R(_si)\
[43]                 _IL_(p, i, curstackdepth, _si) = si2rbase##_si[i]
[44]             STACKED_OPERATION(P_IS_R);
[45]         }
[46]     }
[47]     else {
[48]         for (i = 0; i < curstackdepth; ++i)   {
[49]             assert (oldro[i] != 0.0);
[50]             beta[i] = ro[i]/oldro[stackstate->c2i[i]];
[51]         }
[52]         for (i = 0; i < n; i++) {
[53]             #define P_IS_R_P(_si)\
[54]                 _IL_(p, i, curstackdepth, _si) = si2rbase##_si[i] + (beta[_si] * _IL_(p, i, curstackdepth, _si))
[55]             STACKED_OPERATION(P_IS_R_P);
[56]         }
[57]     }
[58]
```

Figure A.3: Stacked CG iteration function.

```
[59]     #define AABASE(_si) si2aabase##_si = aa + stackstate->c2i[_si] * nnz
[60]     STACKED_OPERATION(AABASE);
[61]     for (i = 0; i < curstackdepth * n; i++)
[62]         q[i] = 0.0;
[63]     for (i = 0; i < curstackdepth; i++)
[64]         product[i] = 0.0;
[65]
[66]     for (i=0; i < n; ++i) {
[67]         k1 = ia[i] - 1;
[68]         k2 = ia[i + 1] - 2;
[69]         for (k=k1; k < k2 + 1; ++k) {
[70]             #define Q_IS_AA_P(_si) \
[71]                 _NI_(q, n, i, _si) += si2aabase##_si[k] * _IL_(p, (ja[k] - 1), curstackdepth, _si)
[72]             STACKED_OPERATION(Q_IS_AA_P);
[73]         }
[74]         #define PRODUCT_IS_P_Q(_si)\
[75]                     product[_si] += _IL_(p, i, curstackdepth, _si) * _NI_(q, n, i, _si);
[76]         STACKED_OPERATION(PRODUCT_IS_P_Q);
[77]     }
[78]
[79]     for (i = 0; i < curstackdepth; i++)
[80]         alpha[i] = ro[i] / product[i];
[81]     for (i = 0; i < n; ++i) {
[82]         #define UPDATE_X(_si) \
[83]             _IL_(x, i, curstackdepth, _si) += (alpha[_si] * _IL_(p, i, curstackdepth, _si))
[84]         STACKED_OPERATION(UPDATE_X);
[85]     }
[86]     #define UPDATE_R(_si)\
[87]         for (i = 0; i < n; ++i) { \
[88]             si2rbase##_si[i] -= (alpha[_si] * _NI_(q, n, i, _si));\
[89]         }
[90]     STACKED_OPERATION(UPDATE_R);
[91]
[92]     for (i = 0; i < curstackdepth; ++i) //Zero Criteria
[93]         criteria[i] = 0.0;
[94]     for (k = 0; k < curstackdepth; ++k) {
[95]         for (i = 0; i < n; ++i) {
[96]             criteria[k] += r[(k * n) + i] * r[(k * n) + i];
[97]         }
[98]     }
[99]     j = 0;
[100]    int converged_at_step = 0;
[101]    for (i = 0; i < curstackdepth; i++) {
[102]         if (sqrt(criteria[i]) < delta) {
[103]             osf_remove_from_stack(stackid, stackstate, i);
[104]             ++converged_at_step;
[105]         }
[106]     }
[107]     if (converged_at_step == curstackdepth)
[108]         return converged_at_step;
[109]     for (i = 0; i < curstackdepth; ++i)
[110]         oldro[stackstate->c2i[i]] = ro[i];
[111]     return converged_at_step;
[112] }
```

A.4.3 Replicating the Stacked Code

The stacked CG initialization and iteration functions, as well as other utility functions that are written by developers, are placed in the same source code file, `osf_user_library.c`. As discussed in Section 3.3.2, its source code includes two sections. The first section, which is marked as 'Section A' in Figure 3.12, includes functions that are not being replicated, e.g., the utility functions. Everything included in the second section, 'Section B,' is replicated *stack depth* times. The sections A and B are separated from each other in the code by defining a guard macro `_OSF_GENERATE_ONCE` as follows:

```
/* Beginning of SECTION A */
#ifndef _OSF_GENERATE_ONCE
#define _OSF_GENERATE_ONCE

... functions that are not replicated (such as utility functions) ...

#endif
/* End of SECTION A */
/* Beginning of SECTION B */

... functions that need replication (such as CGinit and CGiter)...

/* End of SECTION B */
```

This concludes the steps to stack any given iterative solver using OSF developer tools. The OSF makefile we provide generates the final code using `osf_user_library.c` as a code template. The name of this file is arbitrary, and multiple source files could be used for stacking multiple solvers as long as they include proper guard macros for the separation of Sections A and B.

Appendix B

Supplementary OSF Results

This appendix presents an evaluation of operation stacking for the Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES) iterative solvers. For these experiments, we used a 1GHz AMD Athlon machine with 512KB RAM, with 256 KB 8-way associate L2 cache. We compiled our code using GCC compiler with '*-O2 -funroll-loops*' optimization flags and utilized the PAPI library to obtain hardware event counter data. This appendix complements the OSF results we present in Section 3.4, mainly by (1) including results obtained by the OSKI library [VDY05], (2) demonstrating that the benefits provided by OSF for CG and GMRES are almost identical and (3) demonstrating how small matrices that completely fit in the cache may cause OSF to incur performance loss.

We first demonstrate the speedup provided by OSF for CG and GMRES algorithms for stackdepths 2, 4 and 8 in Figures B.1 and B.2. The x-axis depicts the experiments using a $n\&s$ tuple, where n denotes a matrix of size $n \times n$, and s stands for the sparsity. If the entire

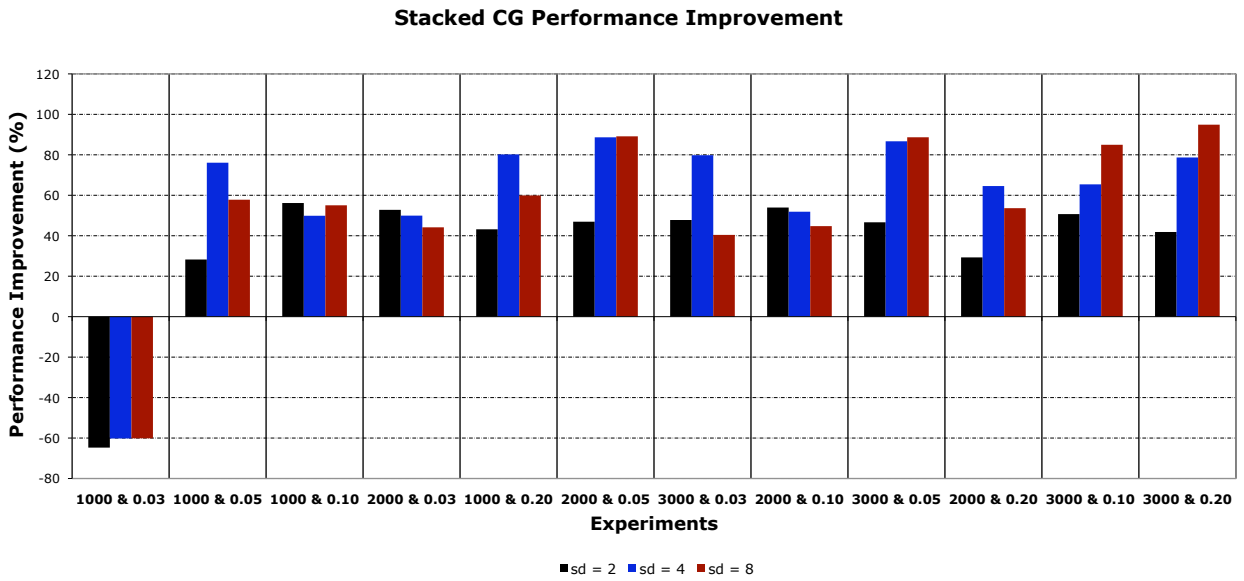


Figure B.1: Speedup for stacked CG algorithm.

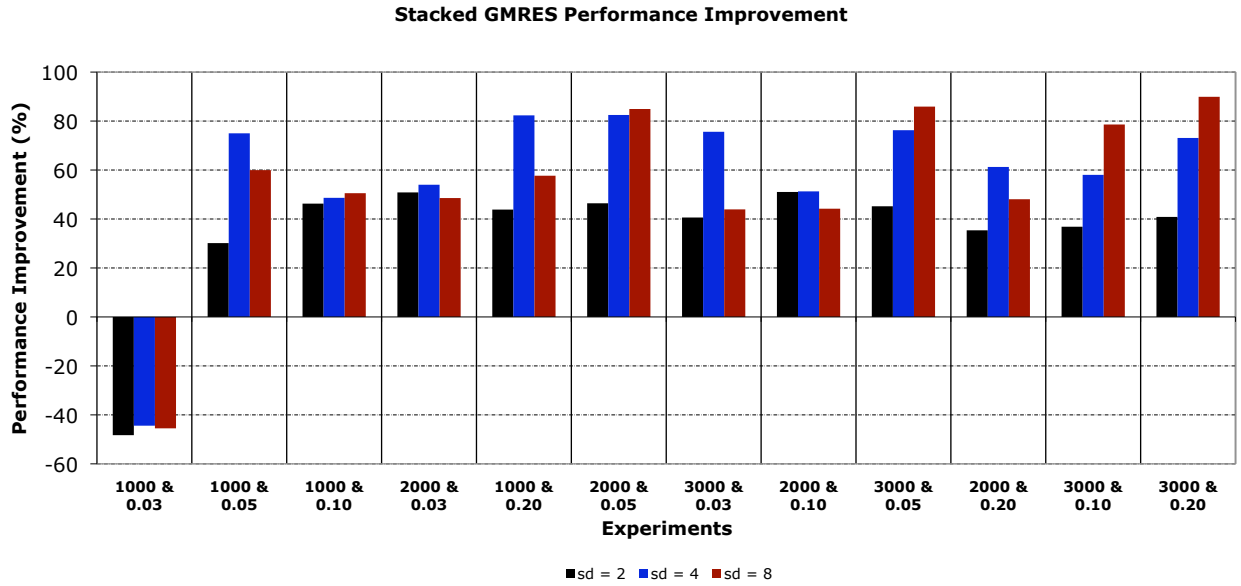


Figure B.2: Speedup for stacked GMRES algorithm.

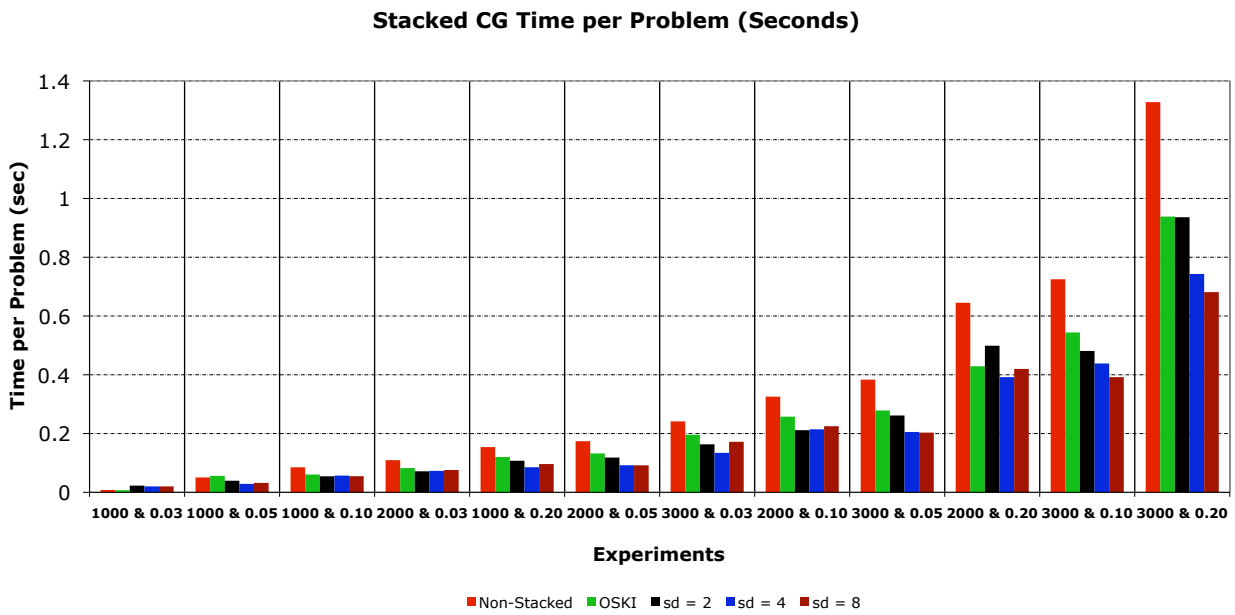


Figure B.3: Reduction in CG overall runtime by OSF.

problem fits in cache, such as in our smallest test case (1000×1000 with 3% sparsity), the non-stacked SMVM algorithm runs well, since no data is ever evicted from the cache after the compulsory fetch. Attempts to stack such small problems cause significant performance loss (as seen in Figures B.1 and B.2), because stacked versions incorporate bigger arrays to hold stacked data, and so may not fit in the cache anymore. Leaving out the smallest problem size, the average performance improvements for stack depth 2, 4, and 8 on the CG are 45%, 70%, and 65% respectively. The corresponding improvements for GMRES are 42%, 67%, and 63%. For both methods, OSF achieves the highest speedup with the largest matrix (3000 & 0.20), with performance improvements of 95% for CG and 90% for GMRES. CG spends slightly more (4%) time in SMVM than GMRES.

To check that our non-stacked SMVM implementation does not provide a misleadingly poor baseline, we compare its performance to the performance of the tuned SMVM implementation

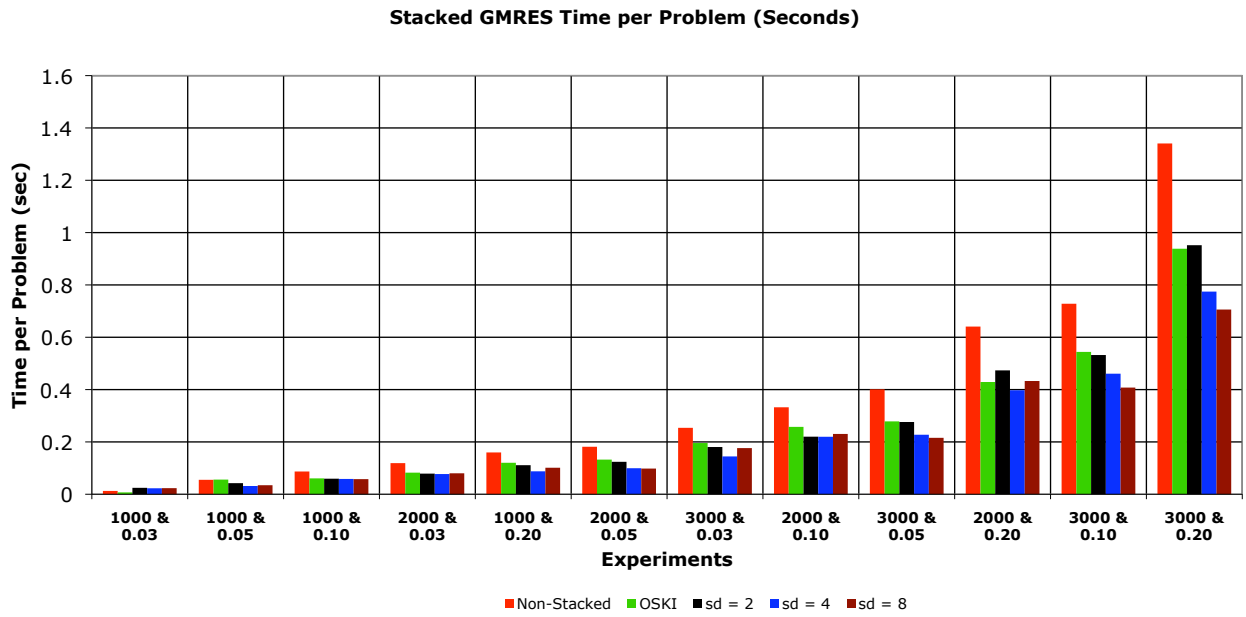


Figure B.4: Reduction in GMRES overall runtime by OSF.

provided by OSKI [VDY05]. Figures B.3 and B.4 depict the time taken to solve a single problem for CG and GMRES algorithms that use non-stacked, OSKI-tuned, and stacked SMVM kernels for stack depths 2, 4, and 8. We provide OSKI hints about the number of SMVM calls (30 iterations) and assumed no symmetry; a more detailed tuning might have yielded better performance. These results do not include the tuning overhead OSKI incurs while analyzing the matrix and rearranging its data. Although OSKI is able to tune a matrix with a given sparsity structure once, then store the results on disk, it has to re-tune if the sparsity structure changes, whereas the benefits of OSF can be reaped immediately. Averaged over the experiments, the OSKI kernel yields a 30% performance improvement, which is slightly less than the improvement obtained by stacking two problems for both methods. We must emphasize, however, that OSKI is not a competing approach to operation stacking and that the two are not mutually exclusive.

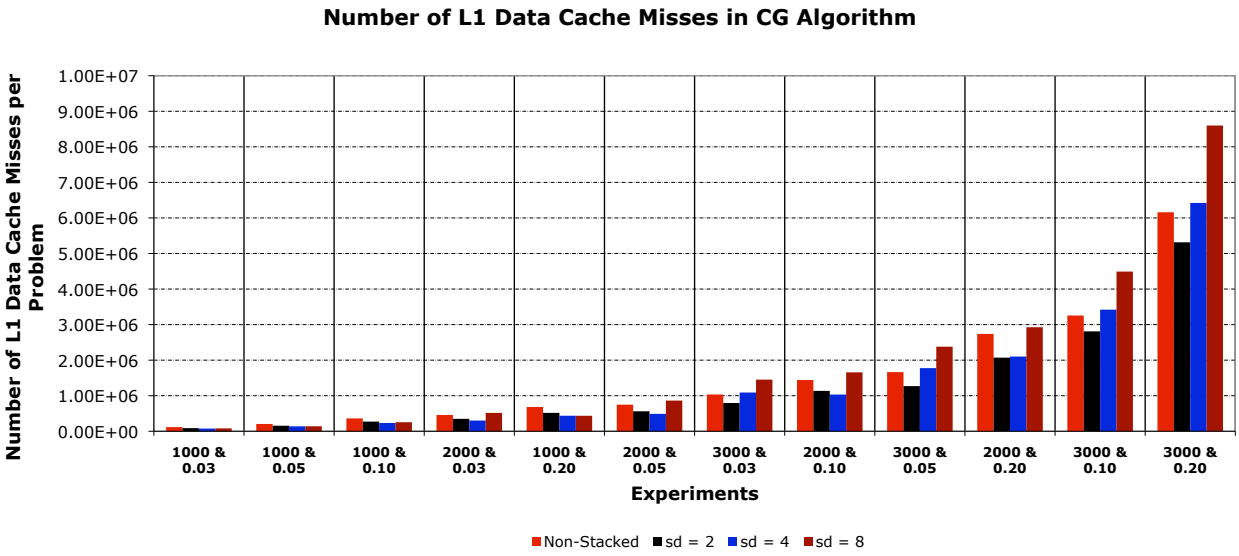


Figure B.5: Number of L1 data cache misses per problem for CG algorithm.

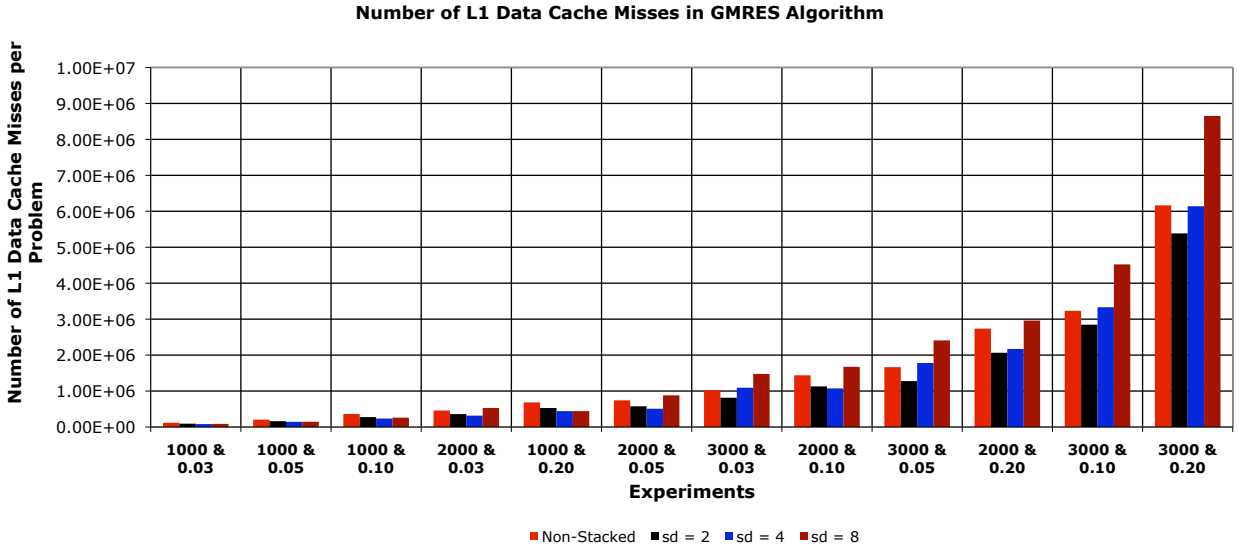


Figure B.6: Number of L1 data cache misses per problem for GMRES algorithm.

To validate the source of the observed performance improvements, we also investigated the changes in cache utilization using operation stacking. Figures B.5 and Figures B.6 depict the number of L1 data cache misses for the CG and GMRES algorithms. Stacking slightly reduces the number of L1 cache misses for stack depths 2 and 4. However, for stack depth 8 the code incurs more misses than in the non-stacked case. As a result, there are more accesses to the L2 cache than in the non-stacked case, as depicted in Figures B.7 and B.8 for CG and GMRES. Despite this situation, stack depth 8 retains its performance advantage for two reasons. First, savings in memory bandwidth reduction by stacking eight problems easily compensate for the increased number of L1 misses, which incur a very small overhead. Second, increased accesses to L2 do not necessarily cause more L2 misses, as depicted in Figures B.9 and B.10 for CG and GMRES, which demonstrate significant reductions in L2 cache misses due to improved spatial locality of stacked and interleaved data. Corresponding improvements in L2 cache hit rates are given in Figures B.11 and B.12. Since the first test case fits completely in the cache, the non-stacked version already exhibits high L2 hit rates, namely 96% for CG and 91% for GMRES. Stacking algorithms for this case degrades this hit rate significantly, which explains the slowdown by OSF in this particular case. For example, the CG algorithm yields hit rates of 19.6%, 3.9%, and 4.6% for stack depths 8, 4, and 2, because stacked data exceeds the cache capacity. As expected, a similar situation also applies to GMRES. The larger matrices, on the other hand, demonstrate significant improvements in L2 data cache hit rates. These cases had almost zero hit rates before operation stacking (the bar indicating the non stacked case is barely noticeable), but they now experience up to 60% hit rates after operation stacking for both CG and GMRES.

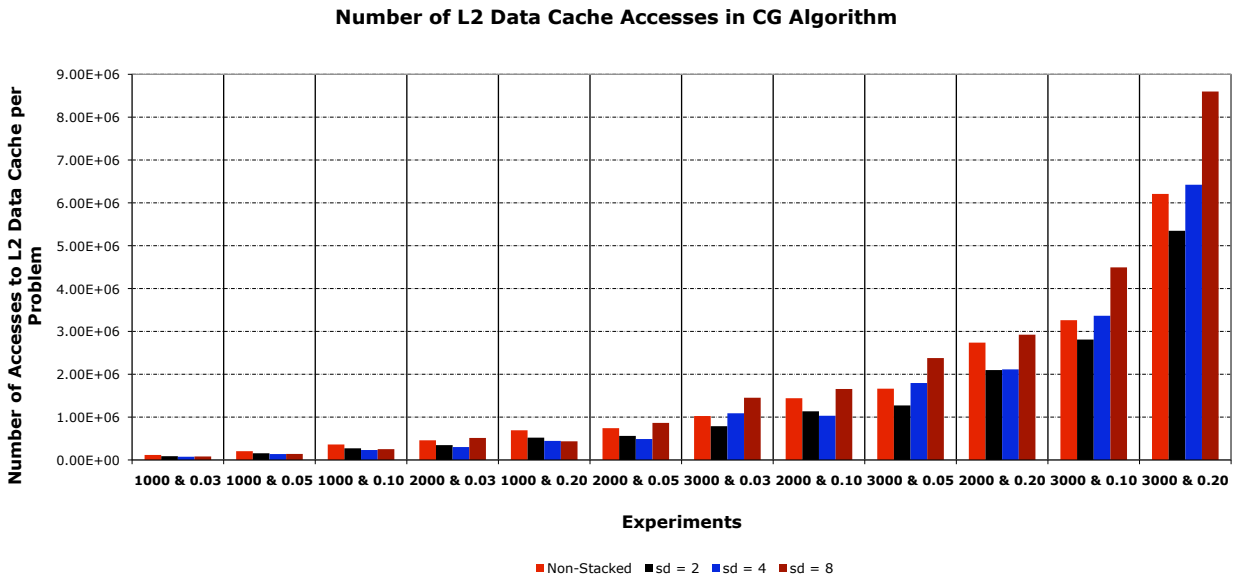


Figure B.7: Number of L2 data cache accesses per problem for CG algorithm.

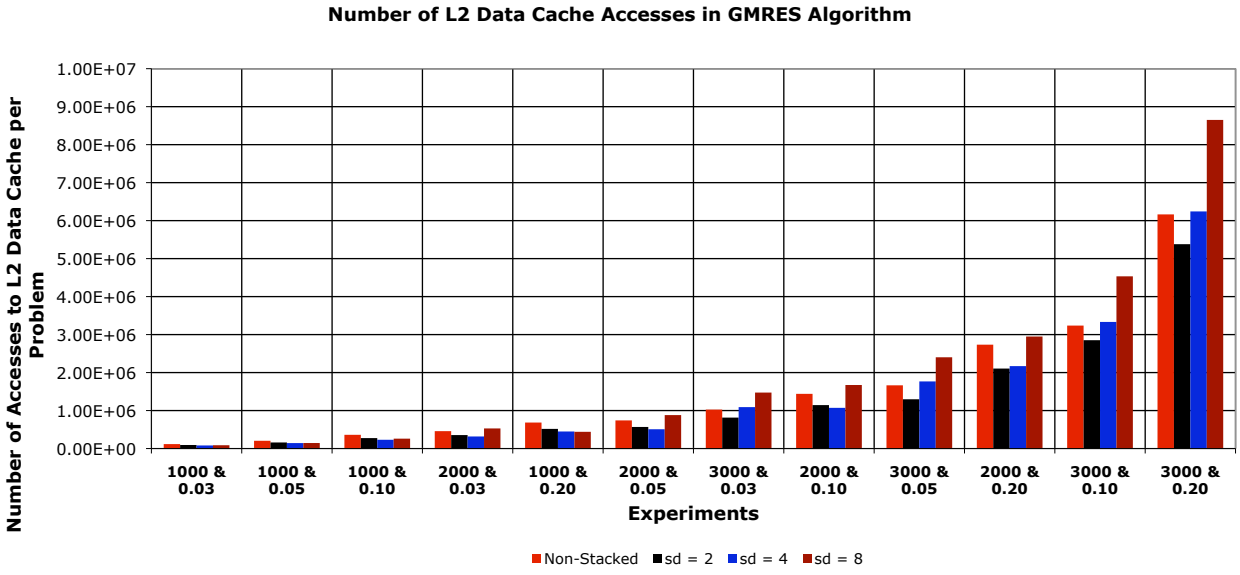


Figure B.8: Number of L2 data cache accesses per problem for GMRES algorithm.

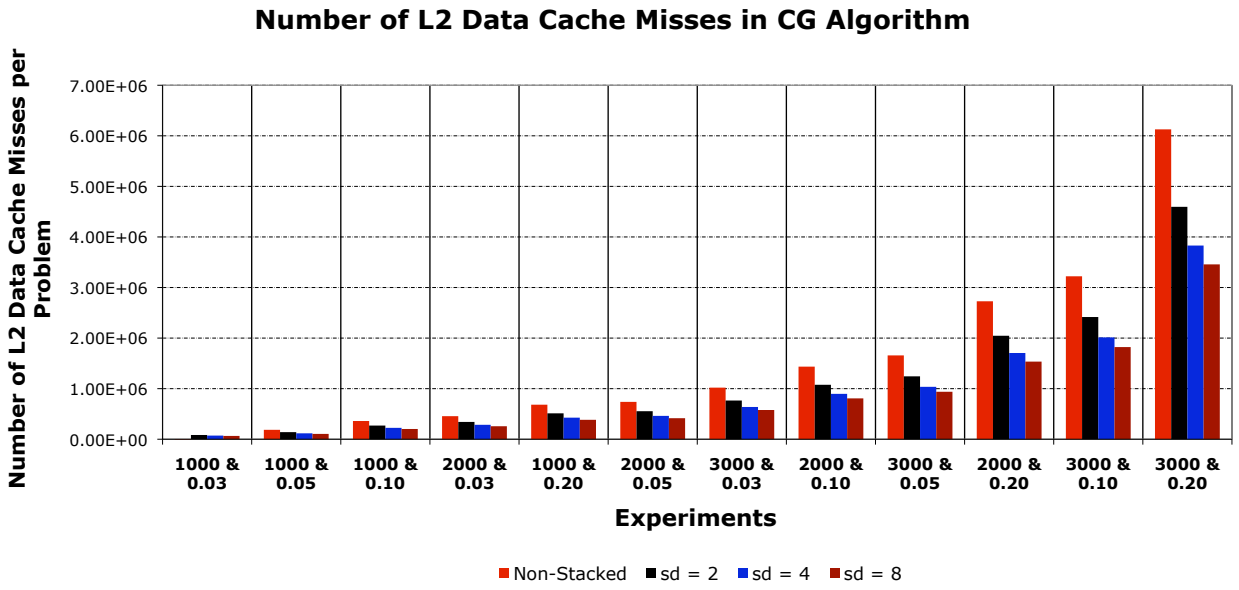


Figure B.9: Number of L2 data cache misses per problem for CG algorithm.

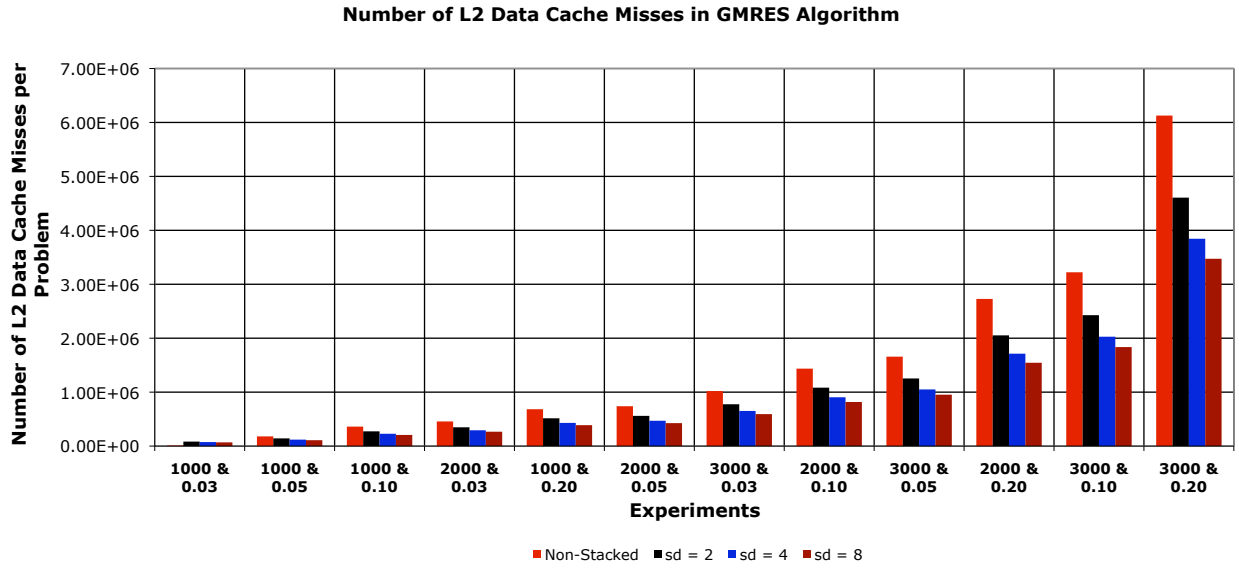


Figure B.10: Number of L2 data cache misses per problem for GMRES algorithm.

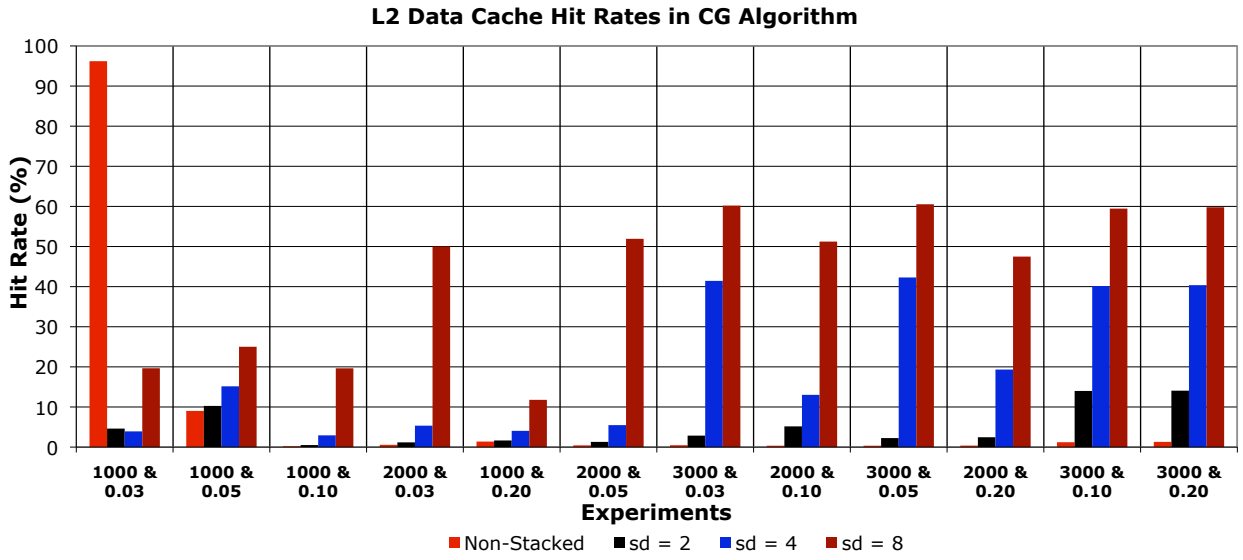


Figure B.11: Number of L2 data cache hits per problem for CG algorithm.

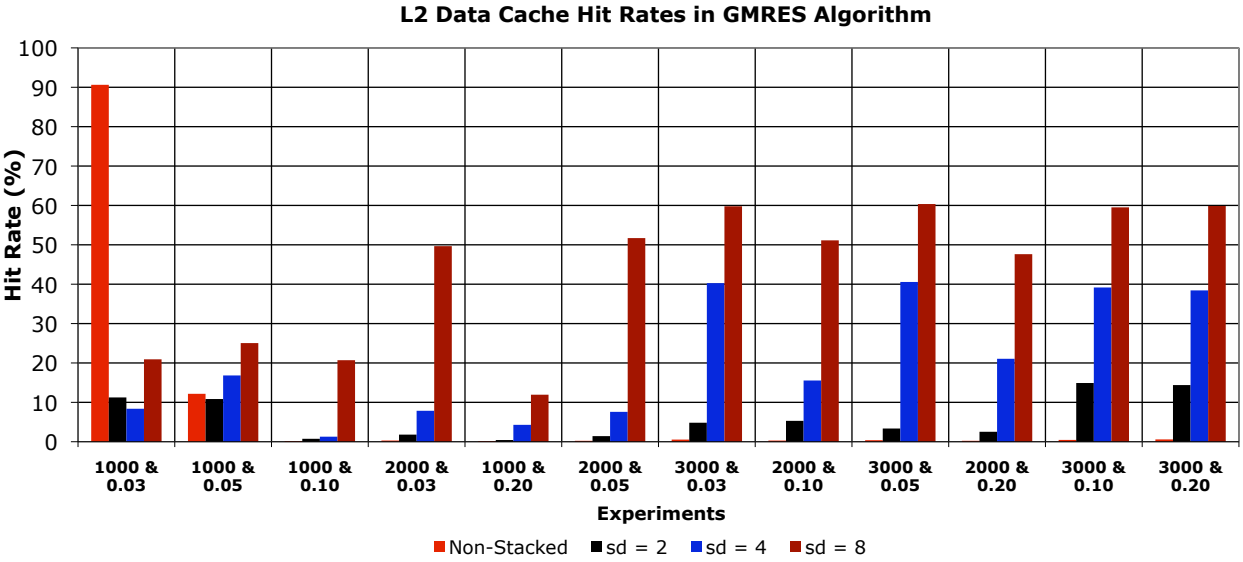


Figure B.12: Number of L2 data cache hits per problem for GMRES algorithm.