Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations

Hasan Metin Aktulga, Aydın Buluç, Samuel Williams, Chao Yang
Computational Research Division
Lawrence Berkeley National Lab
{hmaktulga, abuluc, swwilliams, cyang}@lbl.gov

***Abstract*—Obtaining highly accurate predictions on the properties of light atomic nuclei using the configuration interaction (CI) approach requires computing a few extremal eigenpairs of the many-body nuclear Hamiltonian matrix. In the Many-body Fermion Dynamics for nuclei (MFDn) code, a block eigensolver is used for this purpose. Due to the large size of the sparse matrices involved, a significant fraction of the time spent on the eigenvalue computations is associated with the multiplication of a sparse matrix (and the transpose of that matrix) with multiple vectors (SpMM and SpMM_T). Existing implementations of SpMM and SpMM_T significantly underperform expectations. Thus, in this paper, we present and analyze optimized implementations of SpMM and SpMM_T. We base our implementation on the compressed sparse blocks (CSB) matrix format and target systems with multi-core architectures. We develop a performance model that allows us to understand and estimate the performance characteristics of our SpMM kernel implementations, and demonstrate the efficiency of our implementation on a series of real-world matrices extracted from MFDn. In particular, we obtain 3-4× speedup on the requisite operations over good implementations based on the commonly used compressed sparse row (CSR) matrix format. The improvements in the SpMM kernel suggest we may attain roughly a 40% speed up in the overall execution time of the block eigensolver used in MFDn.**

***Keywords*—Sparse Matrix Multiplication; Block Eigensolver; Nuclear Configuration Interaction; Extended Roofline Model**

I. Introduction

The solution of the quantum many-body problem transcends several areas of physics and chemistry. Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The configuration interaction (CI) method allows computing the many-body wave functions associated with the discrete energy levels of nuclei with high accuracy. Typically, one is only interested in 10-20 low energy states, which can be computed by partially diagonalizing the nuclear many-body Hamiltonian.

Many-body Fermion Dynamics for nuclei (MFDn) is the state-of-the-art nuclear CI code used for studying the structures of light nuclei. MFDn consists of two major parts, (i) the construction of the Hamiltonian matrix $\hat{H}$, and (ii) the computation of its lowest eigenpairs. $\hat{H}$ is constructed in a many-body basis space based on the harmonic oscillator single-particle wave functions. Since $\hat{H}$ is a large symmetric sparse matrix, a parallel iterative eigensolver is preferred [1].

One of the biggest challenges in CI calculations is the massive size of $\hat{H}$ (millions to billions of rows and billions to trillions of nonzeros depending on the nucleus of interest and the desired accuracy level) and the size of its eigenvectors. Although one may be motivated to calculate $\hat{H}$ on-the-fly, the construction of $\hat{H}$ is computationally very expensive. Thus, rather than reconstructing it at each iteration of the eigensolver, $\hat{H}$ is constructed once at the beginning and is preserved throughout the computation in MFDn. Consequently, calculations that are possible using this approach are limited by a supercomputer's memory capacity and compute capability. In order to eliminate redundant computations during the expensive matrix construction phase and to cut the memory requirements, only half of the symmetric $\hat{H}$ matrix is stored in the distributed memory available on a cluster. Therefore each processor owns both a sub-matrix in the lower triangle, as well as its transposed counterpart in the upper triangle. A unique 2D triangular processor grid is used to carry out the computations in parallel in this case [2], [3]. The accuracy that can be obtained through computations in single-precision arithmetic is sufficient to calculate the physical observables of interest in MFDn. Therefore the Hamiltonian matrix is stored in single-precision to further reduce the memory requirements.

While a Lanczos-based eigensolver is commonly used for symmetric sparse matrices, MFDn can use the locally optimal block preconditioned conjugate gradient (LOBPCG) [4], a block eigensolver, for a number of reasons. First, the LOBPCG algorithm allows the use of many-body wave functions from lower model spaces (less accurate but inexpensive calculations) to be used as good initial guesses. Second, a preconditioner can be built based on an understanding of the underlying physics of the problem to improve the convergence rate of the LOBPCG algorithm. In the Lanczos algorithm, one cannot make use of an initial guess nor a preconditioner. Finally and most relevant to our focus in this paper, LOBPCG algorithm requires the multiplication of a sparse matrix with multiple vectors (SpMM), whereas the Lanczos algorithm uses the sparse matrix vector multiplication (SpMV) as a building block. Since the SpMV operation has a low arithmetic intensity, the overall performance of the Lanczos algorithm would ultimately be limited by the processor's DRAM bandwidth. On the other hand, in SpMM, one can make use of the increased data locality in the vector block and attain much higher floating-point operation performance on modern multicore architectures.

As noted above, distributed memory parallel approaches are necessary to solve the actual problems of interest in MFDn.

During matrix construction, a well-balanced distribution of the overall load to the compute nodes is ensured through heuristics that are outside the scope of this paper. In this work, we focus on the performance of local thread-parallel SpMM computations within a single process. Since only half of the Hamiltonian matrix is stored, each process must perform a conventional SpMM as well as the transpose operation SpMM_T ($Y = A^T X$). As we show in Section VI, due to the large size of the sparse matrices involved, a significant fraction of the time spent on the eigenvalue computations is actually associated with the sparse matrix computations.

While we focus on MFDn in this paper, the implications of improved SpMM performance are broader. For example, spectral clustering, one of the most promising clustering techniques, uses the eigenvectors associated with the smallest eigenvalues of the Laplacian of the data similarity matrix to partition the graph into various clusters [5], [6]. For a k-way clustering problem, k eigenvectors are needed, where typically $10 \leq k \leq 100$, an ideal range for block eigensolvers. Additional uses of SpMM are explained in the related work (Section VII).

In this paper, we present a new implementation for the SpMM kernel, CSB_COO, which is based on the compressed sparse block (CSB) framework [7]. We target the high-end multicore processors of the Cray XC30 (Edison) at NERSC. We develop a performance model that allows us to understand and estimate the performance characteristics and bottlenecks of our SpMM kernel implementations. We demonstrate the efficiency of our implementation with respect to existing methods for the regular SpMM operation and the transpose operation on a series of real-world matrices extracted from MFDn. In particular, we obtain 3-4× speedup compared to a row-wise thread-parallel implementation that uses the compressed sparse row (CSR) format. The improvements in the SpMM kernel implies an about 40% speed-up in the overall execution time of the block eigensolver.

II. Eigenvalue Problem in MFDn

The eigenvalue problem arises in nuclear structure calculations because the nuclear wave functions Ψ are solutions of the many-body Schrödinger's equation:

$$H\psi = E\psi \quad (1)$$

$$H = \sum_{i<j} \frac{(p_i - p_j)^2}{2mA} + \sum_{i<j} V_{ij} + \sum_{i<j<k} V_{ijk} + \ldots \quad (2)$$

In the CI approach, both the wave functions ψ and the Hamiltonian H are expanded in a finite basis of Slater determinants (anti-symmetrized product of single-particle states). Each element of this basis is referred to as a many-body basis state. The representation of H under this basis expansion is a sparse symmetric matrix $\hat{H}$. Thus, in CI calculations, Schrödinger's equation becomes a finite-dimensional eigenvalue problem, where one is interested in the lowest eigenvalues (energies) and their associated eigenvectors (wave functions). Many-body basis state i corresponds to the ith row and column of the

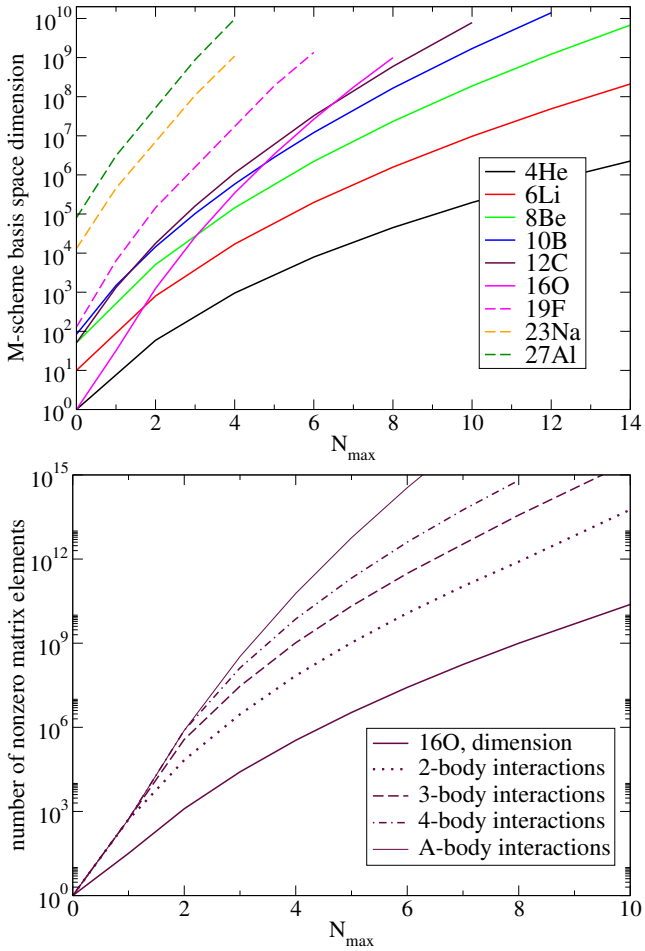

Fig. 1. The dimension and the number of non-zero matrix elements of the various nuclear Hamiltonian matrices as a function of the truncation parameter $N_{\max}$.

Hamiltonian matrix. A nonzero in the Hamiltonian matrix indicates the presence of an interaction between different many-body basis states. Both the total number of many-body states (the dimension of $\hat{H}$) in our adopted harmonic oscillator (HO) basis and the total number of nonzero matrix elements in $\hat{H}$ are controlled by the number of nuclear particles, the truncation parameter $N_{\max}$, and the maximum number of HO quanta above the minimum for a given nucleus (see Figure 1). Higher $N_{\max}$ values yield more accurate results for a given nucleus, but at the expense of an exponential growth in problem size.

As mentioned above, to find the lowest nev number of eigenvalues and eigenvectors of $\hat{H}$, we use the locally optimal block preconditioned conjugate gradient (LOBPCG) algorithm [4]. Algorithm 1 gives the pseudocode for a simplified version of the LOBPCG algorithm without a preconditioner. LOBPCG is a subspace iteration method where we start with an initial guess about the eigenvectors (ψ_0) and refine our guess at each iteration of the solver (ψ_0). So in Alg. 1, ψ_i, R_i and P_i correspond to dense blocks of vectors. In order to ensure good convergence, the dimension of the initial subspace, m, is typically set to 1.5 to 2 times the number of

desired eigenpairs nev. Since we need 10-25 lowest eigenpairs in MFDn, the dense vector block typically has a width of 15 to 48. For numerical stability, the converged eigenpairs are locked, *i.e.* m gets smaller as the algorithm progresses. Therefore in Section V, values of m that are of interest to us ranges from 1 to 48.

Algorithm 1: Pseudocode of the LOBPCG algorithm without a preconditioner, which is used for solving the eigenvalue problem $\hat{H}\Psi = E\Psi$.

Input: $\hat{H}$, matrix of dimensions $n \times n$;
Input: Ψ_0, a block of vectors of dimensions $n \times m$;
Output: Ψ and E such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I$;
Orthonormalize the columns of Ψ_0;
$P_0 \leftarrow 0$;
for *i = 0, 1, …, until convergence* **do**
  $E_i = \Psi_i^T \hat{H} \Psi_i$;
  $R_i \leftarrow \hat{H}\Psi_i - \Psi_i E_i$
  Apply the Rayleigh-Ritz procedure on span$\{\Psi_i, R_i, P_i\}$;
  $\Psi_{i+1} \leftarrow \underset{Y \in span\{\Psi_i, R_i, P_i\}, Y^TY=I}{\text{argmin}} \text{trace}(Y^T \hat{H} Y)$
  $P_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$;
  Check convergence;

III. Experimental Setup

A. MFDn test matrices

In this paper, we use a series of real-world matrices extracted from MFDn. Since our focus is on the performance of single-node sparse matrix computations, we use a small partition of the sparse matrices that arise in actual computations. We have three such test cases, labeled "Nm6", "Nm7", and "Nm8", which are extracted from matrices that correspond to the truncated Hamiltonian of the ^{10}B nucleus at $N_{\max}$ = 6, 7 and 8 cutoff levels, respectively. The actual Hamiltonian matrices are very large and therefore nominally distributed across 15, 45, and 190 processes in the actual MFDn calculations. The sparsity of $\hat{H}$ is determined by the underlying interaction potential. A 2-body interaction potential has been used to construct our matrices. Each processor receives a different sub-matrix of the Hamiltonian, but these sub-matrices have very similar sparsity structures. In this paper, we use processor 1's sub-matrix as our input for performance optimization and evaluation. Figure 2 gives a MATLAB spy sparsity plot of the Nm6 matrix, where each nonzero block is marked by a dot with the intensity of each dot representing the nonzero density of the corresponding block.

Table I enumerates the test matrices used in this paper. Note that the test matrices have millions of rows and hundreds of millions of nonzeros. As we will discuss in Section IV, we use the compressed sparse block (CSB) format [7] in our SpMM implementation. Therefore a sparse matrix is stored in blocks of size $\beta \times \beta$. When blocked with $\beta = 6000$, we observe that both the number of blocked rows and the average number of nonzeros per nonzero block remains high. Moreover, the sparsity pattern ensures that 41-64% of these blocks are nonzero. Unfortunately, there is a high variance on the number of nonzeros per nonzero block making load balancing a potential challenge.

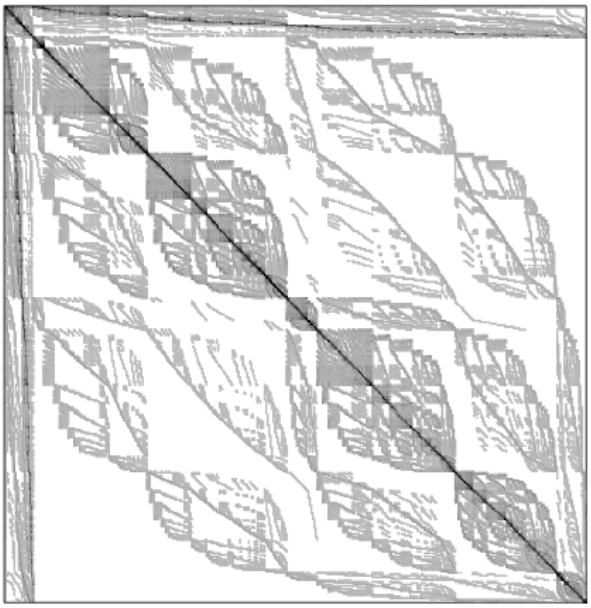

Fig. 2. Sparsity structure of the local Nm6 matrix at process 1 in an MFDn run with 15 processes. A block size of $\beta = 6000$ is used. Each dot corresponds to a block with nonzero matrix elements in it. Darker colors indicate denser nonzero blocks.

Matrix	Nm6	Nm7	Nm8
Rows	2,412,566	4,985,944	7,583,735
Columns	2,412,569	4,985,944	7,583,938
Nonzeros (nnz)	429,895,762	648,890,590	592,325,005
Blocked Rows	403	831	1264
Blocked Columns	403	831	1264
nnz per Block	7991	4191	2311

TABLE I
MFDn matrices (per-process sub-matrix) used as drivers for our optimization efforts. For the purposes of these statistics, all matrices were cache blocked using $\beta = 6000$, and the number of threads $P = 12$.

B. Cray XC30 (Edison)

In this paper, we use the Cray XC30 MPP at NERSC (Edison) which contains more than 10 thousand, 12-core Xeon E5 CPUs [8]. Each of the 12 cores runs at 2.4 GHz and is capable of executing one AVX (8×32-bit SIMD) multiply and one AVX add per cycle and includes both a private 32 KB L1 data cache and a 256 KB L2 cache. Although the per-core L1 bandwidth exceeds 75 GB/s, the per-core L2 bandwidth is less than 40 GB/s. There are two 128-bit DDR3-1866 memory controllers that provide a sustained STREAM bandwidth of 52 GB/s per processor. The cores, the 30MB last-level L3 cache and memory controllers are interconnected with a complex ring network-on-chip which can sustain a bandwidth of about 23 GB/s per core. Given this complex memory hierarchy of varying capacities and bandwidths, the ultimate bottlenecks

	Cray XC30 (Edison)
Core	**Intel Ivy Bridge**
Clock (GHz)	2.4
Data Cache (KB)	32+256
Memory-Parallelism	HW-prefetch
Processor	**Xeon E5-2695 v2**
Cores/CPU	12
Threads/CPU	24[1]
Last-level L3 Cache/CPU	30 MB
Single-Precision GFlop/s	460.8
Aggregate L2 Bandwidth	480 GB/s
Aggregate L3 Bandwidth	276 GB/s
STREAM Bandwidth[2]	52 GB/s
Memory per NUMA node	32 GB

TABLE II
OVERVIEW OF EVALUATED PLATFORMS. [1]WITH HYPER THREADING. BUT ONLY 12 THREADS WERE USED IN OUR COMPUTATIONS. [2]MEMORY BANDWIDTH IS MEASURED USING THE STREAM COPY BENCHMARK.

to performance can be extremely non-intuitive and require performance modeling. The characteristics of this machine are summarized in Table II.

We use the Intel Fortran compiler (version 13.1.3) with the options `-fast -openmp` and constrain our experiments to a single processor as a proxy for a MPI process. For comparisons with original CSB, which uses Intel Cilk Plus for parallelism, we use the Intel C++ compiler (version 13.1.3) with flags `-O2 -no-ipo -parallel -restrict -xAVX`. Since Intel Cilk Plus uses dynamically loaded libraries that are not natively supported by the Cray operating system, we use the cluster compatibility mode [9], a solution that enables Cray machines to run every application that runs on a standard Linux cluster with only a small performance degradation.

IV. MULTIPLICATION OF THE SPARSE MATRIX WITH MULTIPLE VECTORS

We denote the local partition of $\hat{H}$ by A and Ψ_i by X. As can be seen in Table I, the number of rows and columns of A are typically very close to each other. Therefore, for simplicity, we take A to be a square matrix and denote its dimension by N. Y is the output vector block. Both X and Y are dense vector blocks of dimensions N by m. Naively, one can realize SpMM by applying one SpMV to each column of the block of vectors X. In this paper, we use a row-major layout for X as it matches well with the other routines in LOBPCG. Thus, for spatial locality, the simplest SpMM implementation should be implemented as an extension of SpMV in which the operation on scalar elements $y_i = \sum A_{i,j}x_j$ becomes an operation on m-element vectors $Y_i = \sum A_{i,j}X_j$. This operation can be implemented as a for-loop for each nonzero.

A. CSR Format (Baseline)

The most common sparse matrix storage format is compressed sparse rows (CSR) in which the nonzeros of each matrix row are stored consecutively as a list in memory. One maintains an array of pointers (which are simply integer offsets) into the list of nonzeros in order to mark the beginning of each row. An additional index array is used to keep the column indices of the nonzeros. Nonzero values and column indices are stored in separate arrays of length nnz, and the row pointers array is of length $N + 1$. For single-precision sparse matrices whose rows and columns can be addressed with 32-bit integers (i.e., $n \leq 2^{32} - 1$), the storage cost for the CSR format is $8nnz + 4N + 1$. One may reuse matrices stored in the CSR format for the SpMM_T operation by reinterpreting row pointers and column indices as column pointers and row indices, respectively. Such an interpretation would correspond to a compressed-sparse column (CSC) representation in which one operates on columns rather than row sums to implement the SpMM_T operation.

B. Cache-blocked CSB Format

MFDn's very large block of vectors ($10m$ MB to $30m$ MB each) coupled with its matrices' sparsity pattern make attaining locality on the block of vectors very difficult with the CSR representation. After a few rows, it is likely that vector data will have been evicted from the L2 cache, while after a few hundred rows, it is very likely that data will have been evicted from even the last level L3 cache. Moreover, during the SpMM_T operation, CSC's scatter operation coupled with the reduction required for threading can significantly impede performance. Thus, it is imperative that we adopt a data structure that can attain good locality for the vector blocks and does not suffer from the performance penalties associated with the CSR and CSC implementations.

Our data structure for storing sparse matrices is a variant of the compressed sparse blocks (CSB) format [7]. For a given block size parameter β, CSB nominally partitions an $N \times N$ matrix into $\beta \times \beta$ blocks. When β is on the order of $\sqrt{N}$, we can address nonzeros within each block by using half the bits needed to index into the rows and columns of the full matrix (16 bits instead of 32 bits). Therefore, for $\beta = \sqrt{N}$, the storage cost of CSB matches the storage cost of traditional formats such as CSR. In addition, CSB automatically enables cache blocking [10], [11]. Each $\beta \times \beta$ block is independently addressable through a 2D array of pointers. SpMM operation can be performed by processing this 2D array by rows, while SpMM_T can be realized by processing it via columns. Although one can easily thread across either the rows or the columns without any need for temporary storages, threading across both presents a data hazard which must be resolved.

The formal CSB definition does not specify how the nonzeros are stored within a block. An existing implementation of CSB for sparse matrix-vector (SpMV) and sparse matrix-transpose-vector (SpMV_T) multiplication stores nonzeros within each block using a space filling curve to exploit data locality and enable efficient parallelization of the blocks themselves [7].

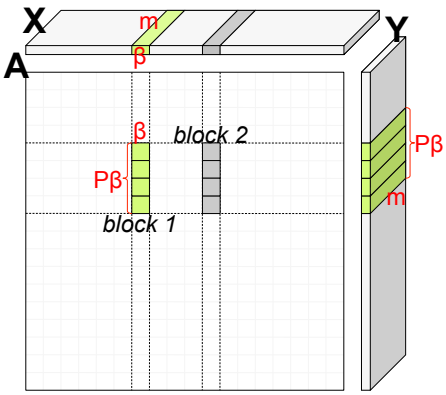

Fig. 3. Computational structure of the SpMM operation Y=AX with $P = 4$ threads. After initializing Y to zero, the operation proceeds by performing all $P\beta \times \beta$ local SpMM operations Y=AX+Y one blocked row at a time. The operation A^TX is realized by permuting the blocking ($\beta \times P\beta$ blocks) and implementing a local SpMM_T on a cache block.

C. Implementation and Optimization

In this paper, our baseline SpMM implementation was written in Fortran using the CSR format for the matrices. The operation $Y = AX$ was threaded using an `!$omp parallel do` with dynamic scheduling over the rows of the matrix, while the transpose operation $Y = A^TX$ was threaded over columns where each thread uses a private copy of the block of destination vectors Y. These private copies are then merged to complete the operation.

On a multi-core CPU like the Xeon E5 with 12 cores, the CSR implementation described above is certainly not suitable for performing SpMM_T on large sparse matrices. Keeping a private copy of the output vector for each thread requires an additional $O(NmP)$ storage, where P denotes the number of threads. The fact that more storage space than the sparse matrix itself would be needed for values of m as small as 8 in the Nm8 testcase shows how significant the memory overheads can actually be. Keeping several private output vector copies may also adversely effect data reuse in cache. Therefore we implemented the *rowpart* algorithm. It uses the CSR implementation for SpMM, but to perform the SpMM_T operation, the sparse matrix is preprocessed and divided into row partitions with equal number of nonzeros for load-balancing purposes. To ensure good performance, each thread maintains a starting and ending index of its row partition boundaries for each column. The *rowpart* implementation requires an extra storage space of only $O(NP)$ and the preprocessing overheads are insignificant when used with an iterative solver.

Our new parametrized implementation for SpMM and SpMM_T, CSB/OpenMP, is based on the CSB format. Like the other implementations described above, CSB/OpenMP is written in Fortran using OpenMP for thread parallelism. As visualized in Figure 3, the matrix is partitioned into $\beta \times \beta$ blocks which are stored in coordinate format (COO) with 16-bit indices and 32-bit single-precision values. When performing AX, threading creates block rows of size $P\beta \times N$; while performing A^TX, threads sweep through block columns of size $N \times P\beta$ and one uses the COO's row indices as column indices and vice versa. We tune for the optimal value of β for each value of m for each matrix.

For the comparisons with the original Cilk-based CSB, we extended the fully parallel SpMV and SpMV_T algorithms [7] in CSB to operate on multiple vectors. We refer to the resulting implementation as CSB/Cilk in this paper. We used a vector of `std::array`'s, a compile-time fixed-sized variant of the built-in arrays for storing X and Y. This effectively creates tall-skinny matrices in row major order. We aligned those input and output vectors to 32-byte boundaries for efficient vectorization. CSB heuristically determines its block parameter β itself, considering the parallel slackness, size of the L2 cache, and the addressability by 16-bit indices. The β chosen for the single vector case was 16,384 or 8,192 (depending on the matrix), and it got progressively smaller all to way to $\beta = 1024$ as m increases due to increased L2 working set limitations.

The SpMM and SpMM_T implemented using CSB/Cilk employ three levels of parallelism. For the SpMV case (the transpose case is symmetric), it first parallelizes across block rows, then within dense block rows using temporary vectors, and finally within each sufficiently dense block if needed. The analysis relies on the fact that the additional parallelization costs of second and third levels are amortized since they are only performed on sufficiently dense blocks and block rows that threaten load balance. Such blocks and block rows can be shown to have enough work to amortize the parallelization overheads. Our SpMM and SpMM_T CSB/OpenMP implementations differ from the CSB/Cilk implementation in the sense that the OpenMP versions do not parallelize within individual block rows (in the case of SpMM) or block columns (in the case of SpMM_T).

In all four implementations (CSR, rowpart, CSB/OpenMP, CSB/Cilk), the innermost loops ($Y_i = \sum A_{i,j}X_j$ for SpMM and $Y_j = \sum A_{i,j}X_i$ for SpMM_T) were manually unrolled for each value of m. In Fortran `!$dir simd` directives and in C `#pragma simd always` pragmas were used. We inspected the assembly code to verify that the compiler generated packed SIMD/AVX instructions for best performance. To minimize TLB misses, we use large pages by loading `craype-hugepages2M` module in the Cray programming environment during compilation and runtime.

D. Performance Expectations

Conventional wisdom suggests that SpMV performance is a function of DRAM STREAM bandwidth and data movement from compulsory misses [12] on matrix elements. Programmers may thus use a simplified Roofline model [13] to bound SpMV time by $8 \cdot nnz/BW_{stream}$ for single-precision CSR matrices [14]. This success has lead to some programmers assuming that performing SpMV's on multiple right-hand sides (SpMM) is essentially no more expensive than performing one SpMV. Unfortunately, this is premised on three major

assumptions — (i) compulsory misses for the vectors are small compared to the matrix, (ii) there are few capacity misses associated with the vectors, and (iii) cache bandwidth does not limit performance. The former is certainly invalidated once the number of right-hand sides reaches two-thirds the average number of nonzeros per row (assuming an 8-byte total space to store single precision nonzeros, 4-byte single-precision vector elements, and a write-allocate cache). The second is only true for low-bandwidth matrices that demand working sets smaller than the last-level cache. The final assumption is highly dependent on microarchitecture, matrix sparsity, and the value of m. We easily demonstrate that for MFDn's matrices and moderate values of m, this conventional wisdom fails to provide a reasonably tight performance bound.

In this paper, we construct a new Roofline performance model that captures how cache locality and bandwidth interact to constrain performance for CSB-like sparse kernels. Let us consider three progressively more restrictive cases: vector locality in the L2, vector locality in the L3, and vector locality in DRAM. As it is highly unlikely a $\beta \times \beta$ block attains good vector locality in the tiny L1 caches, we will ignore this case. Although potentially an optimistic assumption, we assume we may always hit peak L2, L3, or DRAM bandwidth with the caveat that, on average, we overfetch 16 bytes.

First, if we see poor L1 locality for the block of vectors but good L2 locality, then for each nonzero, CSB must read 8 bytes of nonzero data, $4m$ bytes of the source vector, and $4m$ bytes of the destination vector. It may then perform $2m$ flops and write back $4m$ bytes of destination data. Thus we perform $2m$ flops and must move $8+12m$ bytes at an idealized 40 GB/s per core on Edison. Ultimately, this limits CSB performance to 6.6 GFlop/s per core, or about 80 GFlop/s per chip. One should observe that we have assumed high locality in the L2. As this is unlikely, this bound is rather loose.

Unfortunately, static analysis of sparse matrix operations has its limits. In order to understand how locality in the L3 and L3 bandwidth constrain performance we implemented a simplified L2 cache simulator to calculate the number of capacity misses associated with accessing X and Y. For each $\beta \times \beta$ block the simulator tries to estimate the size of the L2 working set based on the average number of nonzeros per column. When the average number of nonzeros per column is less than one, then the working set is bound to $(8m+32) \cdot nnz$ bytes — each nonzero requires a block of the source vector and a block of the destination vector plus overfetch. When the average number of nonzeros per column reaches one, we saturate the working set at $8m\beta$ bytes — full blocks of source and destination vectors. If the working set is less than the L2 cache capacity we must move $8 \cdot nnz + 4m\beta$ bytes when the number of nonzeros per column is equal to or greater than 1 and $(8+4m+16) \cdot nnz$ bytes (but never more than $8 \cdot nnz + 4m\beta$ bytes) when the number of nonzeros per column is less than 1 (miss on the nonzero and the source vector). If the working set exceeds the cache capacity, then we forgo any assumptions on reuse of X or Y in the L2 and incur $(8+4m+16) \cdot nnz + 8m\beta$ bytes of data movement. Thus, the resultant bound on data movement is dependent on both m and the matrix in question.

Finally, let us consider the bound arising from a lack of locality in the L3 and finite DRAM bandwidth. As shown in Figure 3, CSB matrices are partitioned into blocks of size $\beta \times \beta$, and P threads stream through block rows (or block columns for $A^T X$) performing local SpMM operations on blocks of size $P\beta \times \beta$. If one thread (a $\beta \times \beta$ block) gets ahead of the others, then it will likely run slower as it is reading X from DRAM while the others are reading X from the last-level cache. Thus, we created a second simplified cache simulator to track DRAM data movement. However, rather than tracking how individual cores process each $\beta \times \beta$ block, it tracks how a chip processes each $P\beta \times \beta$ block. Thus, the model streams through the block rows of a matrix like Figure 2 and for each nonzero $P\beta \times \beta$ block examines its cache to determine whether the corresponding block of X is present. If it misses, then it fetches the entire block (proxy for prefetch and TLB effects) and increments the data movement tally. If the requisite cache working set exceeds the cache capacity, then we evict a block (LRU policy). Additionally, we add the nonzero data movement and the read-modify-write data movement associated with the destination block of vectors Y ($8Nm$ bytes).

Ultimately, we may combine the estimates for DRAM, L2, and L3 data movement to provide a narrow range of expected SpMM performance as a function of m. Thus, for low arithmetic intensity (small m), the Roofline suggests we will be DRAM-bound, when the Roofline plateaus, it does so not because of peak flop/s, but because of either L2 or L3 bandwidth. In the future, we could use this lightweight simulator as a model-based replacement for the current expensive empirical tuning of beta.

V. SpMM Results

In this section, we present SpMM and SpMM_T performance results for our various optimized implementations across the set of test matrices. We report the average performance obtained over two iterations where the number of requisite floating-point operations is $2 \cdot nnz \cdot m$.

A. CSB Benefit

Figure 4 presents both SpMM ($Y = AX$) and SpMM_T ($Y = A^T X$) performance for the Nm6 matrix as a function of m (the number of vectors). We observe that for $m = 1$, a conventional CSR SpMV implementation does about as well as can be expected. However, as m increases, the benefit of CSB variants' blocking on cache locality is manifested. The CSB/OpenMP version delivers noticeably better better performance than the CSB/Cilk implementation. This may be due in part to additional parallelization overheads of the Cilk version (that uses temporary vectors to introduce parallelism at the block row and block computation levels) or performance issues associated with Cray's cluster compatibility mode. Ultimately, we see performance saturate at better than 65 GFlop/s by $m = 16$. This represents a roughly 45% increase in performance over CSR, and 20% increase over CSB/Cilk.

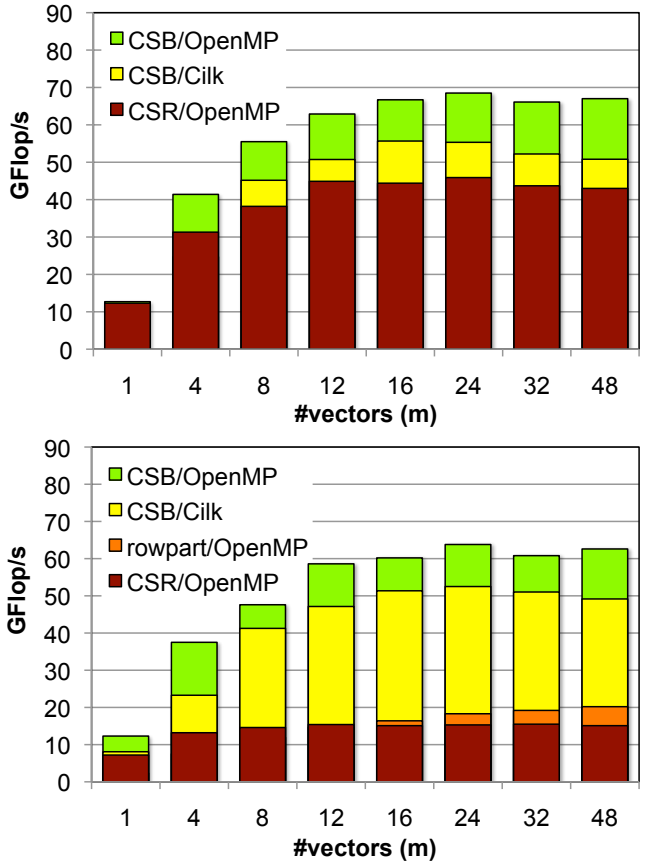

Fig. 4. Optimization benefits on Edison using the Nm6 matrix for SpMM (top) and SpMM_T (bottom) as a function of m (the number of vectors).

CSB truly shines when performing SpMM_T. The ability to efficiently thread the computation coupled with improvements in locality allow CSB/OpenMP to realize a 35% speedup for SpMV over CSR and nearly a 4× improvement in SpMM for $m \geq 16$. The row partitioning scheme had only a minor benefit and only at very large m. Moreover, CSB ensures SpMM and SpMM_T performance are now comparable (67 GFlop/s vs 62 GFlop/s with OpenMP) — a clear requirement as both computations are required for MFDn with LOBPCG.

As a very important note, we would like to point out that the increase in arithmetic intensity afforded by SpMM allows for more than 5× increase in performance over SpMV. This should be an inspiration to explore algorithms that transform numerical methods from being bandwidth-bound (SpMV) to compute-bound (SpMM).

B. Tuning for the Optimal Value of β

As discussed previously, we wish to maintain a working set for the X and Y blocks of vectors as deeply as possible in the cache hierarchy. Each $\beta \times \beta$ block demands a βm working set in the L2 for X as well as Y. Thus, as m increases, we are motivated to decrease β. Figure 5 plots performance of the combined SpMM and SpMM_T operation using CSB/OpenMP on the Nm8 matrix as a function of m

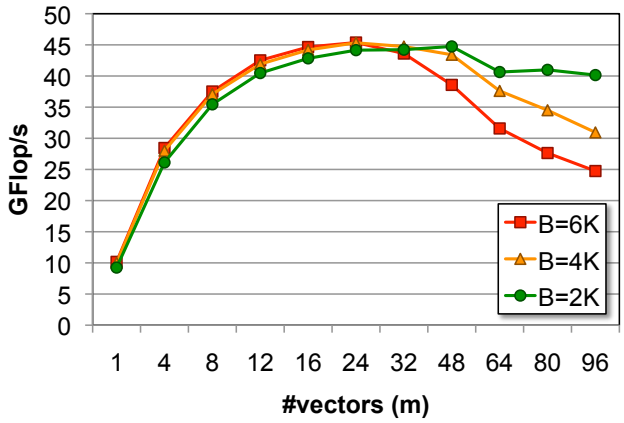

Fig. 5. Performance benefit on the combined SpMM and SpMM_T operation from tuning the value of β for the Nm8 matrix.

for varying β. For small m, there is either sufficient cache capacity to maintain locality on the block of vectors, or the other performance bottlenecks are so pronounced that they mask any capacity misses. However, for large m (we show up to $m = 96$ for illustrative purposes), we clearly see that progressively smaller β are the superior choice as they ensure a constrained resource (e.g. L3 bandwidth) is not flooded with cache capacity miss traffic. Still, note in Figure 5 that no matter what β value is used, the maximum performance obtained for $m > 48$ is lower than the peak of 45 Gflops/s achieved for lower values of m. This suggests that for large values of m, it may be better to perform the SpMM and SpMM_T computations as batches of tasks with narrow vector blocks. In the following sections, we always use the best value of β for a given value of m.

C. Speedup for Combined $[AX, A^TX]$ Operation

LOBPCG on the symmetric matrices of MFDn requires the computation of both AX and A^TX on each iteration. Moreover, we wish to understand the performance benefit for the larger (and presumably more challenging) MFDn matrices. The net benefit is the reduction in the total time required for the two operations, and not a simple geometric mean of their GFlop rates.

Figure 6 presents the combined performance of AX and A^TX as a function of m for our three MFDn test matrices. Clearly, the CSB-like implementations deliver extremely good performance for the combined operation with the Fortran CSB/OpenMP delivering the best performance. We observe that as expected, as one increase the number of vectors m, performance increases to a point at which it saturates. A naive understanding of locality would suggest that regardless of matrix size, the ultimate SpMM performance should be the same. However, as one moves to the larger and sparser matrices, performance saturates at lower values. Understanding these effects and providing possible remedies requires introspection of our performance model.

D. Performance Analysis

In Figure 6, we also provide three Roofline performance bounds based on DRAM, L3, and L2 data movements and bandwidth limits as described in Section IV-D. In all cases, we use the empirically determined optimal value of β for each m as a parameter in our performance model. The L2 and L3 bounds take the place of the traditional in-core (peak flop/s) performance bounds. Bounding data movement for small m (where compulsory data movement dominates) is trivial and thus accurate. However, as m increases, capacity and conflict misses start dominating. In this space, quantifying the volume of data movement in a deep cache hierarchy with an unknown replacement policy and unknown reuse pattern is difficult at best. As Figure 2 clearly demonstrates, the matrices in question are not random (worst case), but exhibit some structure. Thus, one should remember that these Roofline curves for large m are not strict performance bounds but rather guidelines.

Clearly, for small m performance is highly-correlated with DRAM bandwidth. As we proceed to larger m, we see an inversion for the sparser matrices where L3 bandwidth can surpass DRAM bandwidth as the preeminent bottleneck. We observe that for the denser Nm6 matrix, performance is close to our optimistic L2 bound. Nevertheless, the model suggests that the L3 bandwidth is the true bottleneck while DRAM bandwidth does not constrain performance for $m \geq 8$. Conversely, the sparser Nm8's performance is well correlated with the DRAM bandwidth bound for $m \leq 16$ at which point the L3 and DRAM bottlenecks reach parity.

Ultimately, our Roofline model tracks the performance trends well and highlights potential bottlenecks — DRAM, L3, and L2 bandwidths and capacities — as one transitions to larger m or larger and sparser matrices.

VI. Performance Implications

In this section, we investigate the performance implications of using CSB/OpenMP in MFDn. For this purpose, we have computed the 10 lowest eigenpairs of the Hamiltonian matrix arising in ${}^{10}B$, $N_{\max}$=7 calculations, the problem from where the Nm7 matrix was extracted. This computation has been run on 540 cores of Edison using 45 MPI processes with 12 OpenMP threads each. The initial subspace size is set to 16, and it gradually shrinks as eigenpairs converge. It took 70 iterations for the LOBPCG solver to converge to the 10 lowest eigenpairs in this case. In Table III, we report the running times of major computational parts in MFDn for this test.

The existing LOBPCG eigensolver uses the CSR-based rowpart method without manually unrolled innermost loops and explicit SIMDization. Therefore the SpMM and SpMM_T computations in MFDn attain lower flop rates than what has been reported for rowpart in Section V. We estimate that our CSB/OpenMP implementation is slightly more than 3× faster compared to MFDn's rowpart implementation for values of m=1, 4, 8, 12 and 16. As a result, we expect that using the CSB/OpenMP implementation for this case would give about 38% improvement in the eigensolver phase, and 32% improvement overall.

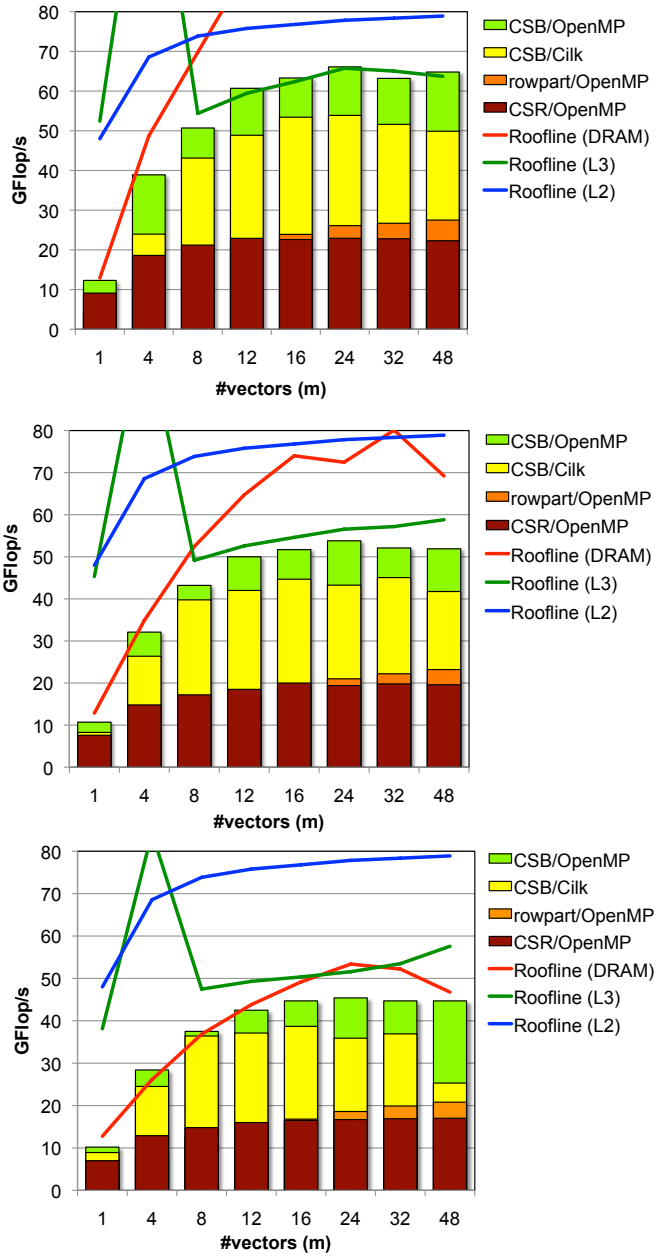

Fig. 6. SpMM and SpMM_T combined performance results on Edison using the Nm6, Nm7 and Nm8 matrices (from top to bottom) as a function of m (the number of vectors).

We note that on a machine like Edison, which has 32 GBs of memory per NUMA node, actual MFDn computations would use much larger local matrices that occupy 20-25 GBs of memory as opposed to the testcases we have used in this study. This means significantly more nonzeros with a relatively small increase in matrix dimensions. So in actual runs, SpMM would be more expensive than other vector operations required for LOBPCG. Also the matrices we used have been extracted from calculations involving 2-body interactions only. In fact, more accurate calculations are possible using 3-body interaction matrices, which have 10-50× more nonzeros per row/column.

TABLE III
Running times of major parts in MFDn for ^{10}B, $N_{\max}$=7 calculations. Computing the 10 lowest eigenpairs requires 70 iterations of the LOBPCG solver.

		eigensolve phase		
	matrix construction	**SpMM**	**LOBPCG other**	**communication**
Time (s)	48.1	171.8	93.4	40.5
Percentage in Eigensolver	–	57%	30%	13%
Percentage in Total	14%	49%	26%	11%

So again in this case, SpMM and SpMM_T would constitute a larger fraction of the LOBPCG solver and the total running time. Consequently, we expect the impact of the CSB/OpenMP implementation to be even higher in actual scientific computations with MFDn.

VII. Related Work

Gropp et al. suggested the use of multiple right hand sides for SpMV in a computational fluid dynamics application in order to improve memory bandwidth limitations [15]. SpMM is one of the core operations supported by the autotuned sequential sparse matrix library, OSKI [16]. OSKI's parallel successor, pOSKI, currently does not currently support SpMM although it is a work in progress [17].

Liu et al. [18] recently investigated strategies to improve the performance of SpMM[1] using SIMD instructions such as AVX/SSE that are available in modern multicore machines. Their driving application is the motion simulation of biological macromolecules in solvent using the Stokesian dynamics method. Apart from major differences in matrix structure and sparsity, our work differs substantially by offering a solution for both SpMM and SpMM_T with roughly the same performance. Furthermore, our Roofline model goes beyond the DRAM bandwidth and compute limits by accounting for the data transfers in L2 and last-level caches.

Block methods that rely on SpMM is used to solve large-scale sparse singular value problems [19], with most popular methods being the subspace iteration and block Lanzcos. Singular value decomposition can be used to perform dimensionality reduction tasks such as latent semantic indexing [20]. More generally, iterative Krylov subspace methods such as BiCG, QMR, IDR, and GMRES, can be extended to block Krylov methods [21] that enable larger search space exploration and potentially higher performance due to increased data reuse. Our techniques will have a positive impact on the performance and adoption of these block Krylov methods.

VIII. Conclusions

The performance of Lanczos-based eigensolvers are increasingly constrained by DRAM bandwidth. With chips increasingly approaching pin count limits, future increases in DRAM bandwidth are constrained by the rate at which DRAM and interconnect technology can progress. Block eigensolvers provide an attractive alternative as they are sufficiently compute-intensive to relegate DRAM bandwidth to a secondary bottleneck. In this paper, we examine performance optimization techniques for the dominant compute kernels found in the LOBPCG algorithm – SpMM and SpMM_T. Conceptually SpMM performs a sparse matrix-vector multiplication on a block of m vectors. Using many-body nuclear Hamiltonian test matrices extracted from MFDn, we demonstrate that the use of compressed sparse blocks (CSB) format in conjunction with tuning and one pragma can accelerate SpMM and SpMM_T performance by up to 1.5× and 4×, respectively. Such speedups may allow for roughly a 40% speedup in LOBPCG, when it is used as the eigensolver in MFDn.

Additionally, we construct a new Roofline model that captures the effects of finite L2 and L3 bandwidth and cache capacity misses in addition to the traditional metrics of cache compulsory misses and DRAM bandwidth. When used to analyze performance, we are able to quickly highlight how the performance bottleneck transitions from DRAM bandwidth to either L2 or L3 bandwidth as we increase m or as our matrices get sparser. Such insights may help drive either future computer architectures to focus on locality and on-chip bandwidths or possibly partitioning software that could be cognizant of various capacity and bandwidth constrains when reordering a matrix.

In the future, we plan on conducting detailed scaling, analysis, and optimization of LOBPCG at scale as there are non-obvious tradeoffs between computer science optimization strategies and reduced algorithmic performance. Such studies will be conducted on not only Edison, but also Blue Gene/Q machines like Mira.

Acknowledgments

Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research and Office of Nuclear Physics. The authors have been funded through the FASTMath Institute, the NUCLEI project and the SUPER project as part of the SciDAC program under the contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

[1]Liu et al. actually uses the name GSpMV for "generalized" SpMV. We refrain from doing so because the same name has been used in conflicting contexts such as SpMV for graph algorithms where the scalar operations can be arbitrarily overloaded.

REFERENCES

[1] P. Maris, H. M. Aktulga, M. A. Caprio, Ü. V. Çatalyürek, E. G. Ng, D. Oryspayev, H. Potter, E. Saule, M. Sosonkina, J. P. Vary *et al.*, "Large-scale ab initio configuration interaction calculations for light nuclei," *Journal of Physics: Conference Series*, vol. 403, no. 1, p. 012019, 2012.

[2] H. M. Aktulga, C. Yang, E. Ng, P. Maris, and J. Vary, "Topology-aware mappings for large-scale eigenvalue problems," *Euro-Par 2012 Parallel Processing*, no. Lecture Notes in Computer Science (LNCS), pp. 830–842, 2012.

[3] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, 2013.

[4] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM journal on scientific computing*, vol. 23, no. 2, pp. 517–541, 2001.

[5] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 3, pp. 568–586, 2011.

[6] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[7] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*, 2009, pp. 233–244.

[8] "Edison website," http://www.nersc.gov/users/computational-systems/edison.

[9] Z. Zhao, Y. H. He, and K. Antypas, "Cray cluster compatibility mode on hopper," in *Cray Users Group Conference (CUG'12)*, Stuttgart, Germany, 2012.

[10] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. SC2007: High performance computing, networking, and storage conference*, 2007.

[12] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, 1989.

[13] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, April 2009.

[14] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, University of California, Berkeley, December 2008.

[15] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, "Toward realistic performance bounds for implicit cfd codes," in *Proceedings of parallel CFD*, vol. 99, 1999, pp. 233–240.

[16] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[17] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," Technical report, EECS Department, University of California, Berkeley, Tech. Rep., 2012.

[18] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, "Improving the performance of dynamical simulations via multiple right-hand sides," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 36–47.

[19] M. W. Berry, "Large-scale sparse singular value computations," *International Journal of Supercomputer Applications*, vol. 6, no. 1, pp. 13–49, 1992.

[20] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIS*, vol. 41, no. 6, pp. 391–407, 1990.

[21] M. H. Gutknecht, "Block krylov space methods for linear systems with multiple right-hand sides: an introduction," in *Modern Mathematical Models, Methods and Algorithms for Real World Systems*, A. Siddiqi, I. Duff, and O. Christensen, Eds. Anamaya Publishers, 2007, pp. 420–447.