A High Performance Block Eigensolver for Nuclear Configuration Interaction Calculations

Hasan Metin Aktulga, Md. Afibuzzaman, Samuel Williams, Aydın Buluç, Meiyue Shao, Chao Yang, Esmond G. Ng, Pieter Maris, James P. Vary

Abstract—As on-node parallelism increases and the performance gap between the processor and the memory system widens, achieving high performance in large-scale scientific applications requires an architecture-aware design of algorithms and solvers. We focus on the eigenvalue problem arising in nuclear Configuration Interaction (CI) calculations, where a few extreme eigenpairs of a sparse symmetric matrix are needed. We consider a block iterative eigensolver whose main computational kernels are the multiplication of a sparse matrix with multiple vectors (SpMM), and tall-skinny matrix operations. We present techniques to significantly improve the SpMM and the transpose operation SpMM^T by using the compressed sparse blocks (CSB) format. We achieve 3–4$\times$ speedup on the requisite operations over good implementations with the commonly used compressed sparse row (CSR) format. We develop a performance model that allows us to correctly estimate the performance of our SpMM kernel implementations, and we identify cache bandwidth as a potential performance bottleneck beyond DRAM. We also analyze and optimize the performance of LOBPCG kernels (inner product and linear combinations on multiple vectors) and show up to 15$\times$ speedup over using high performance BLAS libraries for these operations. The resulting high performance LOBPCG solver achieves 1.4$\times$ to 1.8$\times$ speedup over the existing Lanczos solver on a series of CI computations on high-end multicore architectures (Intel Xeons). We also analyze the performance of our techniques on an Intel Xeon Phi Knights Corner (KNC) processor.

Keywords—Sparse Matrix Multiplication; Block Eigensolver; Configuration Interaction; Extended Roofline Model; Tall-Skinny Matrices

1 Introduction

The choice of numerical algorithms and how efficiently they can be implemented on high performance computer (HPC) systems critically affect the time-to-solution for large-scale scientific applications. Several new numerical techniques or adaptations of existing ones that can better leverage the massive parallelism available on modern systems have been developed over the years. Although these algorithms may have slower convergence rates, their high degree of parallelism may lead to better time-to-solution on modern hardware [1]. In this paper, we consider the solution of the quantum many-body problem using the configuration interaction (CI) formulation. We present algorithms and techniques to significantly speed up eigenvalue computations in CI by using a block eigensolver and optimizing the key computational kernels involved.

The quantum many-body problem transcends several areas of physics and chemistry. The CI method enables computing the wave functions associated with discrete energy levels of these many-body systems with high accuracy. Since only a small number of low energy states are typically needed to compute the physical observables of interest, a partial diagonalization of the large CI many-body Hamiltonian $\hat{H} \in \mathbb{R}^{N\times N}$ is sufficient. More formally, we are interested in finding a small number of extreme eigenpairs of a large, sparse, symmetric matrix:

$$\hat{H}x_i = \lambda x_i, \quad i = 1, \ldots, m, \quad m \ll N. \tag{1}$$

Iterative methods such as the Lanczos and Jacobi–Davidson [2] algorithms, as well as their variants [3], [4], [5], can be used for this purpose. The key kernels for these methods can be crudely summarized as (repeated) sparse matrix–vector multiplications (SpMV) and orthonormalization of vectors (level-1 BLAS). As alternatives, block versions of these algorithms have been developed [6], [7], [8] which improve the arithmetic intensity of computations at the cost of a reduced convergence rate and increased total number of matrix–vector operations [9]. In block methods, SpMV becomes a sparse matrix multiple vector multiplication (SpMM) and vector operations become level-3 BLAS operations.

Related Work: Due to its importance in scientific computing and machine learning, several optimization techniques have been proposed for SpMV [10], [11], [12], [13], [14], [15]. Performance of SpMV is ultimately bounded by memory bandwidth [16]. The widening gap between processor performance and memory bandwidth significantly limits the achievable performance in several important applications. On the other hand, in SpMM, one can make use of the increased data locality in the vector block and attain much higher FLOP rates on modern architectures. Gropp et al. was the first to exploit this idea by using multiple right hand sides for SpMV in a

• *H. M. Aktulga and M. Afibuzzaman are with Michigan State University, 428 S. Shaw Lane, Room 3115, East Lansing, MI 48824. H. M. Aktulga also has an affiliate appointment with the Lawrence Berkeley National Laboratory. Corresponding author e-mail: hma@msu.edu.*
• *S. Williams, A. Buluç, M. Shao, C. Yang, and E. G. Ng are with the Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, MS 50F-1650 Berkeley, CA 94720.*
• *P. Maris and J. Vary are with Iowa State University, Dept. of Physics and Astronomy, Ames, IA 50011.*

computational fluid dynamics application [17]. SpMM is one of the core operations supported by the auto-tuned sequential sparse matrix library OSKI [12]. OSKI's shared memory parallel successor, pOSKI, currently does not support SpMM [18]. More recently, Liu et al. [1] investigated strategies to improve the performance of SpMM [1] using SIMD (AVX/SSE) instructions for Stokesian dynamics simulation of biological macromolecules on modern multicore CPUs. Röhrig-Zöllner et al. [19] discuss performance optimization techniques for the block Jacobi–Davidson method to compute a few eigenpairs of large-scale sparse matrices, and report reduced time-to-solution using block methods over single vector counterparts for quantum mechanics problems and PDEs. Finally, Anzt et al. [20] describe an SpMM implementation based on the SELLC matrix format, and show that performance improvements in the SpMM kernel can translate into performance improvements in a block eigensolver running on GPUs.

Our Contributions: Our work differs from previous efforts substantially, in part due to the immense size of the sparse matrices (with dimensions on the order of several billions and total number of nonzero matrix elements on the order of tens of trillions) involved. We therefore exploit symmetry to reduce the overall memory footprint, and offer an efficient solution to perform SpMM on a sparse matrix and its transpose (SpMM^T) with roughly the same performance [21]. This is achieved through a novel thread parallel SpMM implementation, CSB/OpenMP, which is based on the Compressed Sparse Blocks (CSB) storage format [22] (Sect. 3). We demonstrate the efficiency of CSB/OpenMP on a series of CI matrices where we obtain 3–4$\times$ speedup over the commonly used compressed sparse row (CSR) format. To estimate the performance characteristics and better understand the bottlenecks of the SpMM kernel, we propose an extended Roofline model to account for cache bandwidth limitations (Sect. 3).

In this paper, we extend our previous work (presented in [21]) by considering an end-to-end optimization of a block eigensolver. As discussed in Sect. 4, performance of the tall-skinny matrix operations in block eigensolvers is critical for an excellent overall performance. We observe that the implementations of these level-3 BLAS operations in optimized math libraries perform significantly below expectations for typical matrix sizes encountered in block eigensolvers. We propose a highly efficient thread parallel implementation for inner product and linear combination operations that involve tall-skinny matrices and analyze the resulting performance.

To demonstrate the merits of the proposed techniques, we incorporate the CSB/OpenMP implementation of SpMM and optimized tall-skinny matrix kernels into a LOBPCG [8] based solver in MFDn, an advanced nuclear CI code [23], [24], [25]. We demonstrate through numerical experiments that the resulting block eigensolver can outperform the widely used Lanczos algorithm (based on single vector iterations) with modern multicore architectures (Sect. 5.4). We also analyze the performance of our techniques on an Intel Xeon Phi Knights Corner (KNC) processor to assess the feasibility of our implementations for future architectures.

While we focus on nuclear CI computations, the impact of optimizing the performance of key kernels in block iterative solvers is broader. For example, spectral clustering, one of the most promising clustering techniques, uses eigenvectors associated with the smallest eigenvalues of the Laplacian of the data similarity matrix to cluster vertices in large symmetric graphs [26], [27]. Due to the size of the graphs, it is desirable to exploit the symmetry, and for a k-way clustering problem, k eigenvectors are needed, where typically $10 \leq k \leq 100$, an ideal range for block eigensolvers. Block methods are also used in solving large-scale sparse singular value decomposition (SVD) problems [28], with most popular methods being the subspace iteration and block Lanzcos. SVDs are critical for dimensionality reduction in applications like latent semantic indexing [29]. In SVD, singular values are obtained by solving the associated symmetric eigenproblem that requires subsequent SpMM and SpMM^T computations in each iteration [30]. Thus, our techniques can have a positive impact on the adoption of block solvers in closely related applications.

2 Background

2.1 Eigenvalue Problem in CI Calculations

Computational simulations of nuclear systems face multiple hurdles, as the underlying physics involves a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The eigenvalue problem arises in nuclear structure calculations because the nuclear wave functions Ψ are solutions of the many-body Schrödinger's equation expressed as a Hamiltonian matrix eigenvalue problem, $H\Psi = E\Psi$.

In the CI approach, both the wave functions Ψ and the Hamiltonian H are expressed in a finite basis of Slater determinants (anti-symmetrized product of single-particle states, typically based on harmonic oscillator wave functions). Each element of this basis is referred to as a many-body basis state. The representation of H within an A-body basis space, using up to k-body interactions with $k < A$, results in a sparse symmetric matrix $\hat{H}$. Thus, the Schrödinger's equation becomes an eigenvalue problem, where one is interested in the lowest eigenvalues (energies) and their associated eigenvectors (wave functions). A specific many-body basis state corresponds to a specific row and column of the Hamiltonian matrix. A nonzero in the Hamiltonian matrix indicates the presence of an interaction between either the same or different many-body basis states. Both the total number of many-body states N (the dimension of $\hat{H}$) and the total number of nonzero matrix elements

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.

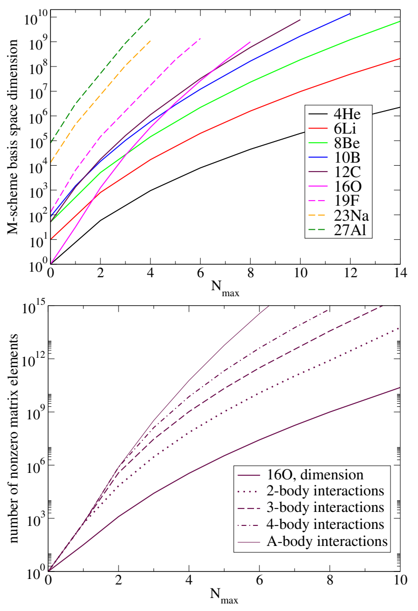


Fig. 1: The dimension and the number of non-zero matrix elements of the various nuclear Hamiltonian matrices as a function of the truncation parameter $N_{\max}$. While the bottom panel is specific to ^{16}O, it is also representative of a wider set of nuclei [23], [25].

in $\hat{H}$ are controlled by the number of nuclear particles, the truncation parameter $N_{\max}$, which is the maximum number of HO quanta above the minimum for a given nucleus, and by the maximum number of particles allowed to interact simultaneously. Higher $N_{\max}$ values yield more accurate results, but at the expense of an exponential growth in problem size (see Fig. 1). Many nuclear physics applications seek to reach at least an $N_{\max}$ of 10 to obtain a sequence of values of observables as a function of $N_{\max}$ using which exact answers can be estimated through extrapolations to infinite $N_{\max}$.

2.2 CI Implementation in MFDn

The CI method is implemented in MFDn [23], [25]. A major challenge in CI is the massive size of the matrix $\hat{H} \in \mathbb{R}^{N\times N}$, where N can be in the range of several billions and the total number of nonzeros can easily exceed trillions. Since only the low-lying eigenpairs are of interest, iterative eigensolvers are used to tame the computational cost [31], [32]. As the identification of nonzeros in $\hat{H}$ and calculation of their values are very expensive, MFDn constructs the sparse matrix only once and preserves it throughout the computation. To accelerate matrix construction and reduce the memory footprint, only half of the symmetric $\hat{H}$ matrix is stored in the distributed memory available. A unique 2D triangular processor grid is then used to carry out the computations in parallel [31], [32]. In this scheme, a "diagonal" processor stores only the lower triangular part of a sub-matrix along the diagonal of $\hat{H}$. Each "non-diagonal" processor, a processor that owns a sub-matrix from either the lower or the upper half of $\hat{H}$, is assigned the operations related to the transpose of that sub-matrix. A well-balanced distribution of the nonzeros among processors is ensured through efficient heuristics [24]. Exploiting symmetry in MFDn demands SpMV^T (SpMM^T) in addition to the SpMV (SpMM) operations, and thus data structures that efficiently implement both operations. The accuracy from single-precision arithmetic is in general sufficient to calculate the physical observables. Hence, in MFDn, the Hamiltonian matrix is stored in single-precision to further reduce the memory footprint.

2.3 Motivation for a Block Eigensolver

As the load balancing issue and communication overheads on distributed memory systems have been addressed in our previous work [24], [31], [32], here we mainly focus on the performance of the thread-parallel computations within a single MPI rank. Conventionally, in MFDnm as well as in other CI codes, the Lanczos algorithm is used due to its excellent convergence properties. However, locally optimal block preconditioned conjugate gradient (LOBPCG) [33], a block eigensolver, is an attractive alternative for a number of reasons. First, the LOBPCG algorithm allows effective use of many-body wave functions from closely related model spaces (e.g. smaller basis, or different single-particle wave functions) to be used as good initial guesses. Second, the LOBPCG algorithm can easily incorporate an effective preconditioner which can often be constructed based on physics insights to significantly improve convergence. Third and most relevant to our focus in this paper, the LOBPCG algorithm naturally leads to an implementation with a high arithmetic density, as the main computational kernels involved are the multiplication of a sparse matrix with multiple vectors, and level-3 BLAS on dense vector blocks, as opposed to the SpMVs and level-1 BLAS operations that are the building blocks in Lanczos. Finally, although not studied here, we note that the potential benefits of a block eigensolver can be even more significant for CI implementations based on on-the-fly computation of the Hamiltonian.

Algorithm 1: LOBPCG algorithm (for simplicity, without a preconditioner) used to solve $\hat{H}\Psi = E\Psi$.

```
Input: $\hat{H}$, matrix of dimensions $N \times N$;
Input: $\Psi_0$, a block of vectors of dimensions $N \times m$;
Output: $\Psi$ and $E$ such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I_m$;
Orthonormalize the columns of $\Psi_0$;
$P_0 \leftarrow 0$;
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$\{\Psi_i, R_i, P_i\}$;
    $\Psi_{i+1} \leftarrow \underset{S\in\text{span}\{\Psi_i,R_i,P_i\},\ S^TS=I_m}{\text{argmin}} \text{trace}(S^T\hat{H}S)$
    $P_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$;
    Check convergence;
```

Alg. 1 gives the pseudocode for a simplified version of the LOBPCG algorithm without preconditioning. LOBPCG is a subspace iteration method that starts with an initial guess of the eigenvectors (Ψ_0) and refines its approximation at each iteration (Ψ_i). R_i denotes the block residual at iteration i and P_i contains the direction information from the previous step. Hence, in Alg. 1, Ψ_i, R_i and P_i correspond to dense blocks of vectors.

We observe that achieving numerical stability with LOBPCG requires using double-precision arithmetic for most MFDn calculations. Therefore, after SpMM computations are completed, the resulting vector blocks $\hat{H}\Psi_i$ are typecast into double precision before the start of LOBPCG computations. Another technique that we use for numerical stability is the locking (deflation) of the converged eigenpairs. Hence, m gets smaller as the algorithm progresses. Finally, 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.

3 Multiplication of the Sparse Matrix with Multiple Vectors (SpMM)

To exploit symmetry in a block eigensolver, each process must perform a conventional SpMM ($Y = AX$), as well as a transpose operation SpMM^T ($Y = A^T X$), where A corresponds to the local partition of $\hat{H}$ and X to a row partition of Ψ_i. The matrix Y is the output vector block in each case. The number of rows and the number of columns of A are typically very close to each other, therefore, for simplicity, we take A to be a square matrix of size $n \times n$. Both X and Y are dense vector blocks of dimensions $n \times m$. As SpMM and SpMM^T are performed in separate phases of the MPI parallel algorithm [32], we use the same input/output vectors to simplify the presentation.

Naively, one can realize SpMM by storing the vector blocks in column-major order and applying one SpMV to each column of X. However, to exploit spatial locality, a row-major layout should be preferred for vector blocks X and Y. This format also ensures good data locality for the tall-skinny matrix operations of LOBPCG. Thus, the simplest SpMM kernel can be implemented as an extension of SpMV where the operation on scalar elements $y_i = \sum A_{i,j} x_j$ becomes an operation on m-element vectors $Y_i = \sum A_{i,j} X_j$. The input and output vectors can be aligned to 32-byte boundaries (by padding with zeros if necessary) for efficient vectorization of the m-element loops. This operation can be implemented by looping over each nonzero $A_{i,j}$.

3.1 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 each nonzero. 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 *local* row and column indices can be addressed with 32-bit integers (i.e., $n \leq 2^{32} - 1$), the storage cost for the CSR format is $8nnz + 4(n+1)$ bytes. 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 rows to implement the SpMM^T operation.

3.2 Cache-blocked CSB Format

Large vector blocks (with $4 \leq m \leq 50$, and $n > 10^6$) can potentially prevent a CSR based SpMM implementation from taking full advantage of the locality in vector blocks depending on the matrix sparsity structure. 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, in a thread-parallel SpMM^T, CSC's scatter operation on thread-private output vectors (necessary to prevent race conditions) coupled with the reduction required for partial thread results can significantly impede performance [23]. 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 [22]. For a given block size parameter β, CSB nominally partitions the $n \times n$ local matrices 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 [13]. In CSB format, each $\beta \times \beta$ block is independently addressable through a 2D array of pointers. The SpMM operation can then be performed by processing this 2D array by rows, while SpMM^T can simply be realized by processing it via columns.

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 transpose sparse matrix–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 [22].

3.3 Implementation and Optimization

CSR/OpenMP: Our baseline SpMM implementation uses the CSR format. The SpMM operation was threaded using an OpenMP parallel for loop with dynamic scheduling over the matrix rows. SpMM^T operation was threaded over columns (which are simply reinterpretations of CSR rows for the transpose) where each thread uses a private copy of the output vector block to prevent race conditions. Private copies are then reduced (using thread parallelism) to complete the SpMM^T operation.

Rowpart/OpenMP: On multi-core CPUs with several cores, the CSR implementation above is certainly not suitable for performing SpMM^T on large sparse matrices. Thread private copies of the output vector require

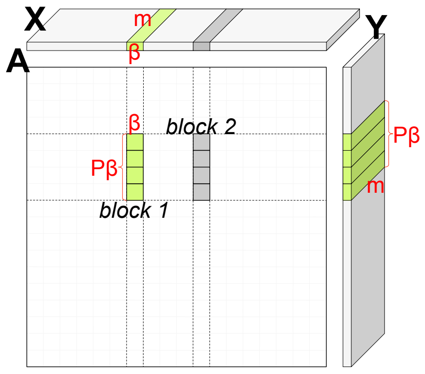


Fig. 2: Overview of the SpMM operation with $P = 4$ threads. The operation proceeds by performing all $P\beta \times \beta$ local SpMM operations Y=AX+Y one blocked row at a time. The operation $A^T X$ is realized by permuting the blocking ($\beta \times P\beta$ blocks).

an additional $\mathcal{O}(nmP)$ storage, where P denotes the number of threads. In fact, more storage space than the sparse matrix itself could be needed for even small values of m for matrices with only tens of nonzeros per row. In terms of performance, thread private output vectors may adversely affect data reuse in the last level of cache, and requires an expensive post-processing step. Therefore we implemented the *Rowpart* algorithm. It is identical to our baseline CSR implementation for SpMM, but for a memory efficient and load balanced SpMM^T, it preprocesses the columns of the transpose matrix and determines row indices for each thread such that row partitions assigned to threads contain (roughly) equal number of nonzeros. Each thread then maintains a starting and ending index of its row partition boundaries per column. Extra storage space cost of *Rowpart* is only $\mathcal{O}(nP)$ and the preprocessing overheads are insignificant when used in an iterative solver.

CSB/OpenMP: Our new parametrized implementation for SpMM and SpMM^T, CSB/OpenMP, is based on the CSB format. As the other implementations, CSB/OpenMP is written in Fortran using OpenMP. As shown in Fig. 2, the matrix is partitioned into $\beta \times \beta$ blocks that are stored in coordinate format (COO) with 16-bit indices and 32-bit single-precision values. The SpMM operation is threaded over individual rows of blocks (corresponding to $\beta \times n$ slices of the matrix), which creates block rows of size $P\beta \times n$. In SpMM^T, threads sweep through block columns of size $n \times P\beta$ and use 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 a given matrix.

CSB/Cilk: For comparisons with the original Cilk-based CSB implementation, we extended the fully parallel SpMV and SpMV^T algorithms [22] in CSB to operate on multiple vectors. 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. The original CSB implementation heuristically determines the block parameter β, considering the parallel slackness, size of the L2 cache, and the addressability by 16-bit indices. The parameter β chosen for the single vector cases presented in Sect. 5 was 16,384 or 8,192 (depending on the matrix), and it got progressively smaller all the way to $\beta = 1024$ as m increases (due to increased L2 working set limitations).

SpMM and SpMM^T implemented using CSB/Cilk employ three levels of parallelism. For SpMM (the transpose case is symmetric), it first parallelizes across rows of blocks, then within dense rows of blocks using temporary vectors, and finally within sufficiently dense blocks if needed. Additional parallelization costs of second and third levels are amortized by performing them on sufficiently dense rows of blocks and individual blocks that threaten load balance. Such blocks and rows of blocks can be shown to have enough work to amortize the parallelization overheads. Our CSB/OpenMP implementation differs from the CSB/Cilk implementation in that CSB/OpenMP does not parallelize within individual rows/columns of blocks or within dense blocks. Rather, CSB/OpenMP partitions the sparse matrix into a sufficiently large number of rows/columns of blocks by choosing an appropriate β. Dynamic scheduling is leveraged to ensure load balance among threads.

In all implementations (CSR, Rowpart, CSB/OpenMP, CSB/Cilk), 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 m value. In Fortran `!$dir simd` directives and in C `#pragma simd always` pragmas were used for vectorization. We inspected the assembly code to ensure that packed SIMD/AVX instructions were generated for best performance. To minimize TLB misses, we used large pages during compilation and runtime.

3.4 An Extended Roofline Model for CSB

Conventional wisdom suggests that SpMV performance is a function of STREAM bandwidth and data movement from compulsory misses on matrix elements. Then the simplified Roofline model [16] provides a lower bound to SpMV time by $8 \cdot nnz/BW_{stream}$ for single-precision CSR matrices [14]. This simple analysis may lead one to conclude 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 assumptions — (i) compulsory misses for 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 first premise is certainly invalidated once the number of right-hand sides reaches half the average number of nonzeros per row (assuming an 8-byte total space for single-precision nonzeros, 4-byte single-precision vector elements, and a write-allocate cache). The second would be true for low-bandwidth matrices with working sets smaller than the last level cache. The final assumption is highly dependent on microarchitecture, matrix sparsity structure, and the value of m. We observe that for MFDn matrices and moderate values of m, this conventional wisdom fails to provide a good performance bound.

In this paper, we construct an extended Roofline performance model that captures how cache locality and bandwidth interact to tighten the performance bound 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 acting on multiple vectors 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 ideally at the peak L2 bandwidth. Ultimately, this would limit SpMM performance to 6.6 GFlop/s per core, or about 80 GFlop/s per chip on Edison which has an L2 cache bandwidth of 40 GB/s per core (see Sect. 5.1). One should observe that we have assumed high locality in 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 L2 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, the working set size is bounded by $(8m+32) \cdot nnz$ 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. So, this bound on data movement depends on both m and the input matrix.

Finally, let us consider the bound due to a lack of locality in L3 and finite DRAM bandwidth. As shown in Fig. 2, CSB matrices are partitioned into blocks of size $\beta \times \beta$, and P threads stream through block rows (or block columns for SpMM^T) 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 which tracks how a chip processes each $P\beta \times \beta$ block, rather than tracking how individual cores process their $\beta \times \beta$ blocks. Our model streams through the block rows of a matrix (like in Fig. 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 and increments the data movement tally. If the requisite cache working set exceeds the cache capacity, then we evict a block (LRU policy). Finally, we add the nonzero data movement and the read-modify-write data movement associated with the output vector block Y ($8nm$ bytes).

Ultimately, the combined estimates for DRAM, L2, and L3 data movement provide us a narrow range of expected SpMM performance as a function of m. For low arithmetic intensity (small m), the Roofline suggests we would be DRAM-bound, but the Roofline plateaus. It likely does so because of either L2 or L3 bandwidth limitations rather than the peak FLOP rate. As we demonstrate in Sect. 5 through numerical experiments, the extended Roofline model closely tracks the observed SpMM performance and allows us to analyze the impact of various potential sources of bottlenecks. In the future, we plan to use this lightweight simulator as a model-based replacement for the expensive empirical tuning of β.

4 Tall-Skinny Matrix Operations

Besides sparse matrix operations, all block methods require operations on the dense blocks of vectors themselves, which we denote as tall-skinny matrix computations owing to the shape of the multiple vector structures involved. The LOBPCG algorithm mainly involves inner product and linear combination operations. Performance in these kernels are critical for the overall eigensolver performance for three reasons. First, an optimized SpMM algorithm incurs a significantly reduced cost on a per SpMV basis. Second, while the per iteration cost of vector operations is $\mathcal{O}(N)$ for Lanczos-like solvers, in block methods these operations cost $\mathcal{O}(Nm^2)$ which grows quickly with m. Finally, and most importantly, the LOBPCG algorithm involves several of these operations in each iteration. For example, computing E_i and updating the residual R_i before the Rayleigh–Ritz procedure in Alg. 1 requires an inner product and a linear combination, respectively. The Rayleigh–Ritz procedure itself requires computing the overlap matrix between each pair of the current Ψ_i, R_i, P_i vectors themselves, as well as their overlap with the vector blocks from the previous iteration, leading to a total of 18 inner product operations. Following the Rayleigh–Ritz procedure is the updates to the Ψ_i, R_i, P_i blocks of vectors for the next iteration, which require computing linear combinations. There are a total of 10 such linear combination operations per Rayleigh–Ritz procedure.

Note that the Hamiltonian matrix in MFDn is partitioned into a 2D triangular grid, and during parallel SpMM the X and Y vector blocks are shared/aggregated among the processes in the row/column groups of this triangular grid [31], [32]. Efficient parallelization of

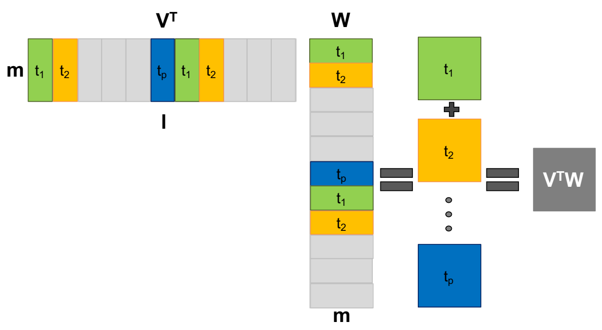


Fig. 3: Overview of the custom implementation for thread-parallel vector block inner product operation V^TW.

LOBPCG operations requires further partitioning X and Y among processes in the same row groups, resulting in smaller local blocks of vectors of size $l \times m$ ($l = n/p_{row}$, where p_{row} is the number of process in a row of the triangular grid). In fact, as shown in Alg. 1, each process has to keep several matrices of size $l \times m$ due to the need for storing the residual R and previous direction P information locally. Since we are mainly interested in the performance of the kernels, we will generically denote the local $l \times m$ blocks of vectors as V and W.

We denote the inner product of two blocks of vectors V and W as V^TW, and the linear combination of the vectors in a block by a small square coefficient matrix C as VC. Both V^TW and VC have high arithmetic intensities. Specifically, for both kernels the number of flops is $\mathcal{O}(lm^2)$ and the total data movement is $\mathcal{O}(lm)$, yielding an arithmetic intensity of $\mathcal{O}(m)$. These kernels can be implemented as level-3 BLAS operations using optimized math libraries such as Intel's MKL or Cray's LibSci. While one would expect to achieve a high percentage of the peak performance (especially for large m), as demonstrated in Sect. 5.3, both MKL and LibSci perform poorly for these kernels. This is most likely due to the unusual shape of the matrices involved (typically $l \gg m$ for large-scale computations).

To eliminate the performance bottlenecks with the V^TW and VC computations, we developed custom thread-parallel implementations for them. Fig. 3 gives an overview of our V^TW implementation. We store V and W in row-major order, consistent with the storage of the vector blocks in sparse matrix computations. We partition V and W into small row blocks of size $s \times m$, and compute the inner product V^TW by aggregating the results of (vendor tuned) `dgemm` operations between a row block in V and the corresponding one in W (as mentioned in Sect. 2.3, for numerical stability LOBPCG computations are performed in double precision, hence the use of `dgemm`). The loop over $s \times m$ blocks is thread parallelized using OpenMP. To achieve load balance with minimal scheduling overheads, we use the `guided` scheduling option. Race conditions in the output matrix are resolved by keeping a thread-private buffer matrix of size $m \times m$. We perform a reduction, which is also thread-parallel, over the buffer matrices to compute the final overlap matrix.

Our custom VC kernel is implemented similarly by partitioning V into row blocks. In this case, C is a square matrix of size $m \times m$ which is read-shared by all threads. Again, the loop over the $s \times m$ blocks of V is thread parallelized with `guided` scheduling. To prevent race conditions, we let each thread perform the computation using the full C matrix, i.e., a `dgemm` between matrices of size $s \times m$ and $m \times m$. Each thread then uniquely produces the corresponding set of s output rows.

5 Performance Evaluation

5.1 Experimental Setup

We use a series of computations with MFDn for a comprehensive evaluation of our methods for sparse and tall-skinny matrix computations. As the overall execution time is dominated by on-node computations, we begin with single-socket performance evaluations of SpMM (Sect. 5.2) and LOBPCG computations (Sect. 5.3). In Sect. 5.4, we inspect the resulting solver's performance in a distributed memory setting.

MFDn Matrices: We identified three test cases, "Nm6", "Nm7" and "Nm8", which are matrices corresponding to the ^{10}B Hamiltonian at $N_{\max}$ = 6, 7, and 8 truncation levels, respectively. The actual Hamiltonian matrices are very large and therefore are nominally distributed across several processes in the actual calculations. For a given nucleus, the sparsity of $\hat{H}$ is determined by (i) the underlying interaction potential, and (ii) the $N_{\max}$ parameter. We used a 2-body interaction potential; a 3-body or a higher order interaction potential would result in denser matrices presenting more favorable conditions for achieving computational efficiency. For a given nucleus and interaction potential, increasing the $N_{\max}$ value reduces the density of nonzeros in each row, thereby allowing us to evaluate the effectiveness of our techniques on a range of matrix sparsities.

Each process on a distributed memory execution receives a different sub-matrix of the Hamiltonian, but these sub-matrices have similar sparsity structures. For simplicity and consistency, we use the first off-diagonal processor's sub-matrix as our input for single-socket evaluations. Table 1 enumerates the test matrices used in this paper. Note that the test matrices have millions of rows and hundreds of millions of nonzeros. As discussed in Sect. 3, we use the compressed sparse block (CSB) format [22] in our optimized 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 block rows and the average number of nonzeros per nonzero block remain high. Fig. 4 gives a sparsity plot of the Nm6 matrix at the block level, where each nonzero block is marked by a dot whose intensity represents the density of nonzeros in the corresponding block. For our test matrices, 41–64% of these blocks are nonzero. We observe a high variance on the number of nonzeros per nonzero block.

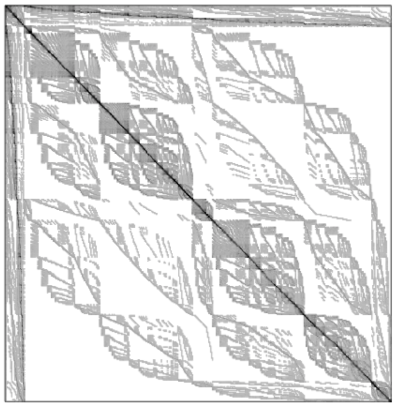


Fig. 4: *Sparsity structure of the local Nm6 matrix at process 1 in an MFDn run with 15 processes. A block size of $\beta = 6000$ is used. Each dot corresponds to a block with nonzero matrix elements in it. Darker colors indicate denser nonzero blocks.*

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
Average nnz per Block	7991	4191	2311

TABLE 1: MFDn matrices (per-process sub-matrix) used in our evaluations. For the statistics in this table, all matrices were cache blocked using $\beta = 6000$.

Vector block sizes: In nuclear physics applications, up to 20–25 eigenvalues are needed, and 5–15 eigenpairs will be sufficient. In LOBPCG, to ensure a rich representation of the subspace and ensure good convergence, the number of basis vectors m needs to be set to 1.5 to 2 times the number of desired eigenvectors nev. As the algorithm proceeds and eigenvectors converge, converged eigenpairs are deflated (or locked) and the subspace shrinks. Therefore, we examine the performance of our optimized kernels for values of m in the range 1 to 48.

Computing Platforms: We primarily use high-end multicore processors (Intel Xeon) for performance studies. However, the energy efficiency requirements of HPC systems point to an outlook where many-core processors will play a prominent role. To guide our future efforts in this area, we conduct performance evaluations on an Intel Xeon Phi processor as well.

Our multi-core results come from the Cray XC30 MPP at NERSC (Edison) which contains more than 10 thousand, 12-core Xeon E5 CPUs. 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.

Our many-core results have been obtained at the Babbage testbed at NERSC. Intel Xeon Phi (MIC) cards are connected to the host processor through the PCIe bus and contain 60 cores running at 1 GHz, with 4 hardware threads per core. Each MIC card has an on-device 8 GB of high bandwidth memory (320 GB/s). Cores are interconnected by a high-speed bidirectional ring. Each core has a 32 KB L1 data cache and 512 KB L2 cache locally with high speed access to all other L2 caches to implement a fully coherent cache. Note that there is not a shared last level L3 cache on the MIC cards. Each core supports 512-bit wide AVX-512 vector ISA that can execute 8 double-precision (or 16 single-precision or integer) operations per cycle. With Fused Multiply-Add (FMA), this amounts to 16 DP or 32 SP FLOPS/cycle. Peak performance for each MIC card is 1 TFlop/s in DP, and 2 TFlop/s in SP. The characteristics of both processors are summarized in Table 2.

Processor	**Xeon E5-2695 v2**	**Xeon Phi 5110P**
Core	**Ivy Bridge**	**Pentium P54c**
Clock (GHz)	2.4	1.05
Data Cache (KB)	32+256	32 + 512
Memory-Parallelism	HW-prefetch	SW-prefetch + MT
Cores/Processor	12	60
Threads/Processor	24[1]	4
Last-level L3 Cache	30 MB	–
SP GFlop/s	460.8	2,022
DP GFlop/s	230.4	1,011
Aggregate L2 BW	480 GB/s	960 GB/s[2]
Aggregate L3 BW	276 GB/s	–
STREAM BW[3]	52 GB/s	320 GB/s
Available Memory	32 GB	8 GB

TABLE 2: Overview of Evaluated Platforms. [1] With hyper threading, but only 12 threads were used in our computations. [2] Based on the `saxpy1` benchmark in [34]. [3] Memory bandwidth is measured using the STREAM copy benchmark.

We use the Intel Fortran compiler with flags `-fast -openmp`. For comparison with the original CSB using Intel Cilk Plus, we use the Intel C++ compiler with flags `-O2 -no-ipo -parallel -restrict -xAVX`. As Intel Cilk Plus uses dynamically loaded libraries not natively supported by the Cray operating system, we use Cray's cluster compatibility mode that causes only a small performance degradation. The Xeon Phi's performance was evaluated in the *native* mode, enabled through the `-mmic` flag in Intel compilers.

5.2 Performance of Sparse Matrix Computations

We first present the SpMM and SpMM^T performance results for our optimized implementations. We report the average performance over five iterations where the number of requisite floating-point operations is $2{\cdot}nnz{\cdot}m$.

5.2.1 *CSB Benefit*

Fig. 5 presents SpMM ($Y = AX$) and SpMM^T ($Y = A^T X$) performance for the Nm6 matrix as a function of m (the number of vectors). For $m = 1$, a conventional CSR SpMV implementation does about as well as can be expected. However, as m increases, the benefit of CSB

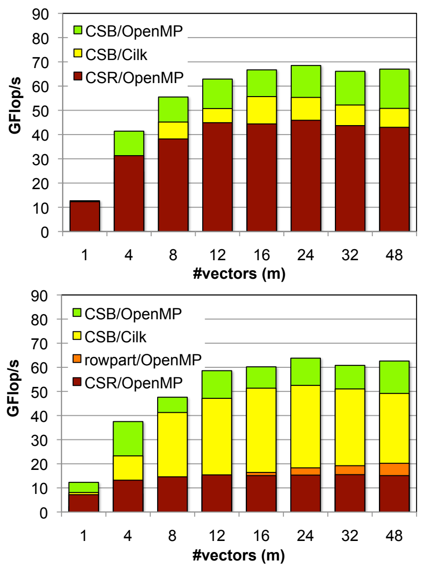


Fig. 5: Optimization benefits on Edison using the Nm6 matrix for SpMM (top) and SpMM^T (bottom) as a function of m (the number of vectors).

variants' blocking on cache locality is manifested. The CSB/OpenMP version delivers noticeably better performance than the CSB/Cilk implementation. This may be due in part to performance issues associated with Cray's cluster compatibility mode, but most likely due to additional parallelization overheads of the Cilk version that uses temporary vectors to introduce parallelism at the levels of block rows and blocks. This additional level of parallelism is eliminated in CSB/OpenMP by noting that the work associated with each nonzero is significantly increased as m increases, and we leverage the large dimensionality of input vectors for load balancing among threads. Ultimately, we observe that CSB/OpenMP's performance saturates at around 65 GFlop/s for $m > 16$. This represents a roughly 45% increase in performance over CSR, and 20% increase over CSB/Cilk.

The CSB based implementations truly shine when performing SpMM^T. The ability to efficiently thread the computation coupled with improvements in locality allows 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 has only a minor benefit and only at very large m. Moreover, the CSB format 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.

As an important note, we point out that the increase in arithmetic intensity introduced 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 memory bandwidth-bound (SpMV) to compute-bound (SpMM).

5.2.2 *Tuning for the Optimal Value of β*

As discussed previously, we wish to maintain a working set for the X and Y vector blocks as close to the processor as possible in the memory hierarchy. Each $\beta \times \beta$ block demands a working set of size βm in the L2 for

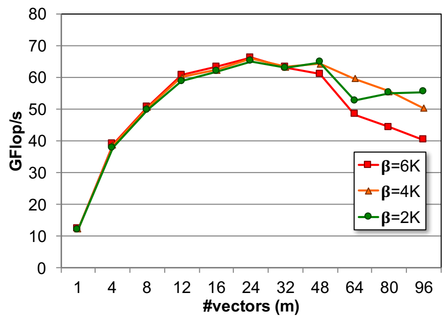


Fig. 6: Performance benefit on the combined SpMM and SpMM^T operation from tuning the value of β for the Nm8 matrix.

X and Y. Thus, as m increases, we are motivated to decrease β. Fig. 6 plots performance of the combined SpMM and SpMM^T operation using CSB/OpenMP on the Nm8 matrix as a function of m for varying β. For small m, there is either sufficient cache capacity to maintain locality on the block of vectors, or other performance bottlenecks are pronounced enough to mask any capacity misses. However, for large m (shown 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 Fig. 6 that no matter what β value is used, the maximum performance obtained for $m > 48$ is lower than the peak of 65 GFlop/s achieved for lower values of m. This suggests that for large values of m, it may be better to perform SpMM and SpMM^T as batches of tasks with narrow vector blocks. In the following sections, we always use the best value of β for a given m.

5.2.3 *Speedup for Combined SpMM/SpMM^T Operation*

Our ultimate goal is to include the LOBPCG algorithm as an alternative eigensolver in MFDn. As discussed earlier, the computation of both SpMM and SpMM^T is needed for this purpose. We are therefore interested in the performance benefit for the larger (and presumably more challenging) MFDn matrices. Fig. 7 presents the combined performance of SpMM and SpMM^T as a function of m for our three test matrices. Clearly, the CSB variants deliver extremely good performance for the combined operation with the CSB/OpenMP delivering the best performance. We observe that as one increases 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 using our performance model.

5.2.4 *Performance Analysis*

Given the complex memory hierarchies of varying capacities and bandwidths in highly parallel processors, the ultimate bottlenecks to performance can be extremely non-intuitive and require performance modeling. In Fig. 7, we provide three Roofline performance bounds based on DRAM, L3, and L2 data movements and

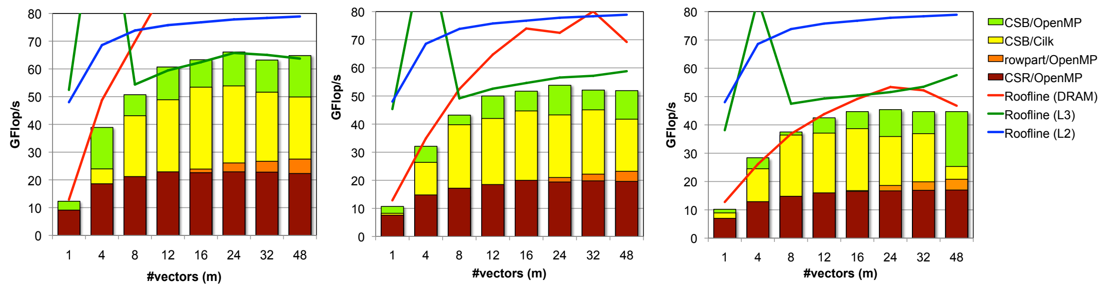


Fig. 7: SpMM/SpMM^T combined performance with Nm6, Nm7 & Nm8 matrices (from left to right) as a function of m (#vectors). We identify 3 Rooflines (one per level of memory) with the extended roofline model.

bandwidth limits as described in Sect. 3.4. 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 non-trivial. As Fig. 4 clearly demonstrates, the matrices in question are not random (worst case), but exhibit a structure. We note 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 pre-eminent 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.

5.3 Performance of Tall-Skinny Matrix Operations

In Fig. 8, we analyze the performance of our custom inner product (V^TW) and linear combination (VC) operations proposed as alternatives to the BLAS library calls for tall-skinny matrices. As mentioned in Sect. 4, our custom implementations still rely on the library `dgemm` calls to perform multiplications between small matrix blocks. Hence, we report the performance of two different versions, *Custom/MKL* based on MKL `dgemm`, and *Custom/LibSci* based on the LibSci `dgemm`.

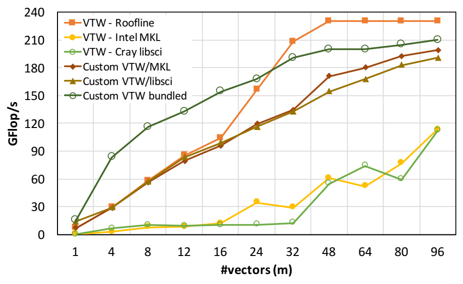
As shown in Fig. 8, we obtain significant speedups over MKL and LibSci in computing V^TW. Both of our custom V^TW kernels exhibit a similar performance profile and outperform their library counterparts significantly. The speedups we obtain range from about 1.7× (for larger values of m) up to 10× (for $m \approx 16$). As small m values are common in an application like MFDn, this represents a significant performance improvement for LOBPCG over using the library `dgemm`. However, for the VC kernel, we do not observe speedups from our custom implementations (Fig. 8) – they closely track the performance of their library counterparts. In fact, for larger values of m, the *Custom/MKL* implementation is outperformed slightly by MKL. While l was fixed at 1 M for these tests, we observed similar results in other cases (l=10 K, 100 K, and 500 K).

Applying the (regular) Roofline model to the tall-skinny matrix kernels reveals that while the performance of these kernels are memory-bound for small values of m, they do not attain a performance close to the expectations when $m > 16$. A key observation here is that the overall performance of both kernels are significantly higher for larger m values. Since LOBPCG contains operations with multiple blocks of vectors, i.e., Ψ_i, R_i, P_i, one potential optimization is to combine these 3 blocks of vectors into a single $l \times 3m$ matrix. In this case, the 18 inner product operations on $l \times m$ matrices would be turned into 2 separate inner products with $l \times 3m$ matrices. As the *Custom bundled* curve shows in Fig. 8, the additional performance gains are significant for V^TW, achieving up to 15× speedup compared to the library counterparts. However, the linear combination operation VC does not benefit from bundling as much as the inner product does. For VC, the improvements we observe are limited to a factor of 1.5 for values of m ranging from 12 to 48. The main reason behind the limited performance gains in this case is that the tall-skinny matrix products can be converted into `dgemm`'s of dimensions $l \times 3m$ and $3m \times m$, as opposed to being extended to $3m$ in all shorter dimensions as in the case of V^TW.

5.4 Putting it Altogether

We now demonstrate the benefits of an architecture-aware eigensolver implementation in actual CI problems. MFDn's existing solver uses the Lanczos algorithm with full orthogonalization (Lanczos/FO) for numerical stability and good convergence. We implemented the LOBPCG algorithm in MFDn using the optimized SpMM and tall-skinny matrix kernels described above. The distributed memory implementations for both solvers are similar and use the 2D partitioning scheme described in Sect. 2.2 and in more detail in [32].

Beyond optimizing the kernels, there are a number of key issues that need to be considered to leverage the full



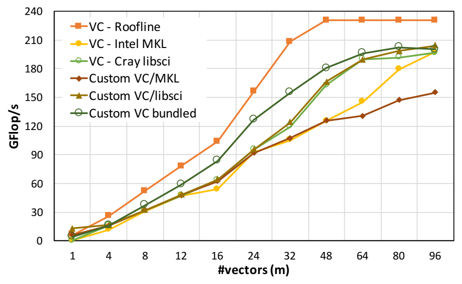


Fig. 8: Performance of the inner product $V^T W$ (top), and linear combination VC (bottom) operations, using Intel MKL, Cray LibSci, and our custom implementations on Edison. Tall-skinny matrix sizes are $l \times m$, where $l = 1M$. Regular roofline is used to model the performance with vector blocks of width m, therefore it sometimes runs below the bundled kernel performance which uses vector blocks of width $3m$.

benefits of LOBPCG. These include the choice of initial eigenvector guesses, the design of a preconditioner to accelerate convergence, and suitable data structures for combining the optimized SpMM and tall-skinny kernels. Full details and analyses of initial guesses and preconditioning techniques used are discussed by Shao *et al.* [35]. We describe them briefly here:

- *Initial Guesses:* CI models with truncations smaller than the target $N_{\max}$ result in significantly smaller problem sizes. Roughly speaking, reducing $N_{\max}$ by 2 (subsequent truncations must be evenly separated) gives an order of magnitude reduction in matrix dimensions. We observe that eigenvectors computed with smaller $N_{\max}$ values provide good approximations to the eigenvectors in the target model space, so we solve the eigenvalue problem of the smaller $N_{\max}$ first and use these results as initial guesses to our target problem. This idea can be applied recursively for additional performance benefits.
- *Preconditioning:* Preconditioners transform a given problem into a form that is more favorable for iterative solvers, often by reducing the condition number of the input matrix. To be effective in large-scale computations, a preconditioner must be easily parallelizable and computationally inexpensive to compute and apply, while still providing a favorable transformation. We build such a preconditioner in MFDn by computing crude approximations to the inverses of the diagonal blocks of the Hamiltonian (easy to parallelize). The diagonal blocks in CI typically contain very large nonzeros (important for a quality transformation).
- *Bundling Blocks of Vectors:* While bundling all three blocks of vectors into a single, but thicker tall-skinny matrix is favorable for LOBPCG (see Sect. 4), this would harm the performance of the SpMM and SpMM^T operations as the locality between consecutive rows of input and output vectors would be lost. A work around to this problem is to copy the input vectors Ψ_i from the vector bundle into a separate block of vectors at the end of LOBPCG (in preparation for SpMM), and then copy $H\Psi_i$ back into the vector bundle after SpMM^T (in preparation for LOBPCG). Our experiments show that overheads associated with such copies are small compared to the gains obtained from bundled $V^T W$ and VC operations, hence we opt to bundle the blocks of vectors in LOBPCG into a single one in our implementation.

Problem	^{10}**B, Nm6**	^{10}**B, Nm7**	^{10}**B, Nm8**
Dimension (in millions)	12.06	44.87	144.06
Nonzeros (in billions)	5.48	26.77	108.53
nev (m for LOBPCG)	8 (12)	8 (12)	8 (12)
residual tolerance	1e-6	1e-6	1e-6
MPI ranks (w/6 omp threads)	15	66	231
Lanczos Iterations	240	320	240
SpMVs	240	320	240
Inner Products	28,800	51,200	28,800
LOBPCG Iterations	28	48	38
SpMVs	324	548	428
Inner Products	72,656	121,296	93,936

TABLE 3: Statistics for the full MFDn matrices used in distributed memory parallel Lanczos/FO and LOBPCG executions.

- *Alignment/Padding:* If vector rows are not aligned with the 32 byte boundaries, the overall performance of the solver is significantly reduced due to the presence of unpacked vector instructions. Hence we set the initial basis dimension m for LOBPCG such that m is a multiple of 4 to ensure vectorization at least with SSE instructions – when m is a multiple of 8, AVX instructions are automatically used, but forcing m to always be a multiple of 8 introduces overheads not compensated by AVX vectorization. Despite setting m to be a multiple of 4 initially, as LOBPCG iterations are performed, eigenvectors start converging one by one and they need to be locked (deflated) when they do so. To prevent the basis size from shrinking to a non-multiple of 4 and maintain good performance throughout, we shrink the basis only when the number of active vectors decreases to a smaller multiple of 4, while replacing the converged eigenvectors with 0-vectors in the meantime.

Our testcases to compare the eigensolvers are the full ^{10}B problems with $N_{\max}$ truncations of 6, 7 and 8, seeking 8 eigenpairs in all cases. Table 3 gives more details for the testcases and the distributed memory runs. We executed the MPI/OpenMP parallel solvers using 6 threads/rank (despite having 12 cores/socket), because the MPI library on Edison currently supports only serialized MPI communications by multiple threads (i.e., `MPI_THREAD_SERIALIZED` mode). Using 12 threads per rank resulted in not being able to fully saturate the network injection bandwidth, and therefore increased communication overheads in both cases.

In Fig. 9, we break down the overall timing

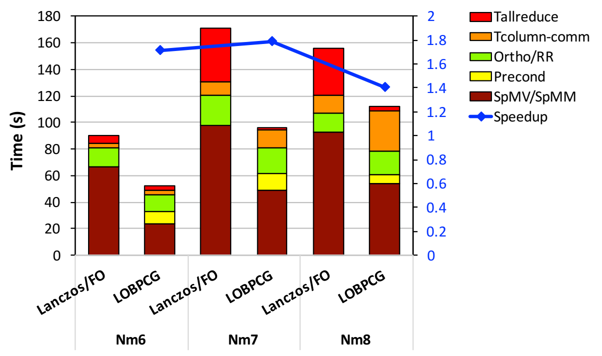


Fig. 9: Comparison and detailed breakdown of the time-to-solution using the new LOBPCG implementation vs. the existing Lanczos/FO solver. Nm6, Nm7 and Nm8 testcases executed on 15, 66, and 231 MPI ranks (6 OpenMP threads per rank), respectively, on Edison.

into the following parts: sparse matrix computations (*SpMV/SpMM*), application of the preconditioner (*Precond*), orthonormalization for Lanczos and Rayleigh–Ritz procedure for LOBPCG (*Ortho/RR*), communications among column groups of the 2D processor grid ($T_{col-comm}$)—row communications are fully overlapped with (*SpMV/SpMM*) [32], and finally `MPI_Allreduce` calls needed for reductions during *Ortho/RR* step ($T_{allreduce}$). The solve times for smaller $N_{\max}$ truncations to obtain the initial LOBPCG guesses are included in the execution times in Fig. 9.

We observe that our LOBPCG implementation consistently outperforms the existing Lanczos solver by a factor of 1.7×, 1.8×, and 1.4× for the Nm6, Nm7 and Nm8 cases, respectively. Although LOBPCG requires more SpMVs overall for convergence (see Table 3), the highly optimized SpMM/SpMM^T kernels and tall-skinny matrix computations allows LOBPCG to outperform the Lanczos solver. Since two threads in a socket (one per MPI rank) are used to overlap communications with SpMM computations, the peak SpMM flop rate during the eigensolver iterations is slightly lower than those in Fig. 7. For LOBPCG computations though, we observe that V^TW and VC kernels execute at rates in line with those in Fig. 8. We remark that without the preconditioner and initial guess techniques we adopted, LOBPCG's slow convergence rate leads to similar or worse solution times compared to Lanczos/FO, wiping out the gains from replacing SpMVs with SpMMs (for a detailed analyses, please see Shao *et al.* [35]). In this regard, for nuclear CI calculations, inexpensive solves with smaller $N_{\max}$ truncations and low cost preconditioners are crucial for the performance benefits observed with LOBPCG.

In Table 3, we also show the total number of inner products required by both solvers. The LOBPCG algorithm relies heavily on vector operations as discussed in Sect. 4 and evidenced by the total number of inner products reported here. So despite the use of Lanczos with full orthogonalization, LOBPCG requires more inner products overall. In addition, a smaller but still significant number of linear combination operations, as well as solutions of small eigenvalue problems are required for LOBPCG. Cumulatively, these factors lead to a computationally expensive *Ortho/RR* part for the LOBPCG solver. So despite using highly optimized BLAS 3 based kernels in LOBPCG, we observe that the time spent in *Ortho/RR* part is comparable for Lanczos/FO and LOBPCG. Finally, in the larger runs, i.e., Nm7 and Nm8, we observe that Lanczos/FO solver incurs significant $T_{allreduce}$ times, possibly due to slight load imbalances exacerbated by frequent synchronizations. On the other hand, $T_{allreduce}$ times are much lower for LOBPCG (an expected consequence of the fewer iteration counts), but LOBPCG's $T_{col-comm}$ are significantly higher due to the larger volume of communication required in this part.

5.5 Evaluation on Xeon Phi Knights Corner (KNC)

Our performance evaluation on Xeon Phi is limited to the isolated SpMM and tall-skinny matrix kernels due to MFDn's large memory requirements and the limited device memory available on the KNC (which was used in the native mode as a proxy for the upcoming Knights Landing architecture). We used the same testcases as before, and experimented with 30, 60, 120, 180, and 240 threads to determine the ideal thread count. We obtained the best performance with 120 threads for both SpMM and tall-skinny matrix kernels and report those results.

Comparing the average SpMM GFlop rates on KNC (given in Fig. 10) with the Ivy Bridge (IB) Xeon, we see that the peak performance on KNC is much lower (as much as 3× for Nm6, $m = 12$, for instance) than that on IB for all cases. This is likely due to the significantly smaller cache space available per KNC thread. In any case, we can clearly see that the CSB/OpenMP implementation delivers significantly better performance than traditional CSR and Rowpart implementations, as it did on IB. On KNC, we also observe the pattern of increased performance with increasing values of m values. But unlike IB where the use of SSE vs AVX vectorization did not make a significant difference (see the similar GFlop rates for $m = 12$ and $m = 16$ in Fig. 7), on KNC utilizing packed AVX-512 vectorization is crucial as indicated by the sharp performance drop in going from $m = 16$ to $m = 24$. We utilize the same extended Roofline model as before, but KNC does not offer a shared L3 cache. Since the data movement into the L2 cache is from the device memory, the device memory bandwidth is used in this case, and the corresponding curve in Fig. 10 is marked as Roofline (L2-DRAM) to distinguish it from the regular Roofline model. With KNC's high bandwidth memory and limited cache space, original DRAM Roofline gives a very loose bound, and so does the plain L2 Roofline. But L2-DRAM Roofline provides a relatively tight envelope.

For the tall-skinny matrix operations, we only present results using the MKL library as LibSci was not available on KNC. In Fig. 11, we observe that our custom V^TW kernel significantly outperforms MKL's `dgemm` (up to 25×). $Custom/Bundled$ implementation gives important performance gains for $8 < m < 48$. For the VC

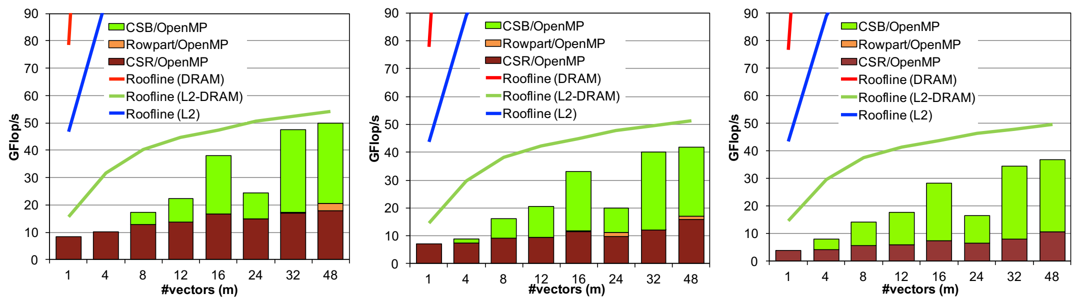


Fig. 10: SpMM and SpMM^T combined performance results on Babbage using the Nm6, Nm7 and Nm8 matrices (from left to right) as a function of m (the number of vectors).

kernel however, our custom implementations provide only slight improvements over MKL. Overall, the performance achieved for both kernels is very low compared to the peak performance predicted by the Roofline model.

The poor performance observed for SpMM and tall-skinny matrix operations on KNC suggests that further optimizations are necessary to achieve good eigensolver performance on future systems.

6 Conclusions

Block eigensolvers are favorable alternatives to SpMV-based iterative eigensolvers for modern architectures with a widening gap between processor and memory system performance. In this study, we focused on the sparse matrix multiple vectors product (SpMM, SpMM^T) and tall-skinny matrix operations (V^TW, VC) that constitute the key kernels of a block eigensolver. Using many-body nuclear Hamiltonian test matrices extracted from MFDn, we demonstrated that the use of the Compressed Sparse Blocks (CSB) format in conjunction with manual unrolling for vectorization and tuning can improve SpMM and SpMM^T performance by up to 1.5$\times$ and 4$\times$, respectively, on modern multi-core processors. As block eigensolvers are sufficiently compute-intensive, the DRAM bandwidth may be relegated to a secondary bottleneck. We presented an extended Roofline model that captures the effects of L2 and L3 bandwidth limits in addition to the original DRAM bandwidth limit. This extended model highlighted how the performance bottleneck transitions from DRAM to the L3 bandwidth for large m values or sparser matrices.

Contrary to the common wisdom, we observe that simply calling `dgemm` in optimized math libraries does not suffice to attain high flop rates for tall-skinny matrix operations in block eigensolvers. Through custom thread-parallel implementations for inner product and linear combination operations and bundling separate vector blocks into a single large block, we have obtained 1.2$\times$ to 15$\times$ speedup (depending on m and the type of operation) over MKL and LibSci libraries.

By integrating our optimized kernels with effective initial guesses and preconditioners into an actual CI code, MFDn, we demonstrated significant speedups using the block eigensolver LOBPCG over an existing Lanczos solver. This case study, albeit being limited to a few test cases, provides a good benchmark for typical CI calculations and demonstrates that block solvers can better serve the needs of large scale scientific computations on modern architectures due to the widening gap between chip vs. DRAM performance.

Finally, we explored the performance of proposed kernels on the KNC many-core processor. Despite its higher peak rate, the limited cache space and increased core counts led to poor performance compared to Xeon CPUs, indicating a need for further performance optimization work. In fact, in the future, we plan on conducting detailed scaling, analysis, and optimization of block eigensolvers on many-core and GPU architectures, as there are major challenges to effectively utilize increased parallelism, while working with limited cache space and a multi-level memory hierarchy.

Acknowledgments

This research was supported by the U.S. DOE's ASCR and NP Offices as part of the Applied Mathematics Program and Scientific Discovery through Advanced Computing (SciDAC) Program, which involves the NUCLEI Project (grant DESC0008485), the FASTMath Institute, and the SUPER Institute. The computational work was performed at NERSC. Funding for Lawrence Berkeley National Laboratory was provided under contract DE-AC02-05CH11231 and funding for Iowa State University was provided through grant DE-FG02-87ER40371.

References

[1] 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.

[2] G. H. Golub and C. F. Van Loan, *Matrix computations*, 4th ed. Johns Hopkins University Press, 2013.

[3] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[4] K. Wu and H. Simon, "Thick-restart Lanczos method for large symmetric eigenvalue problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 2, pp. 602–616, 2000.

[5] A. Stathopoulos and Y. Saad, "Restarting techniques for the (Jacobi–)Davidson symmetric eigenvalue methods," *Electronic Transactions on Numerical Analysis*, vol. 7, pp. 163–181, 1998.

[6] G. H. Golub and R. Underwood, "The block Lanczos method for computing eigenvalues," *Mathematical Software*, vol. 3, pp. 361–377, 1977.

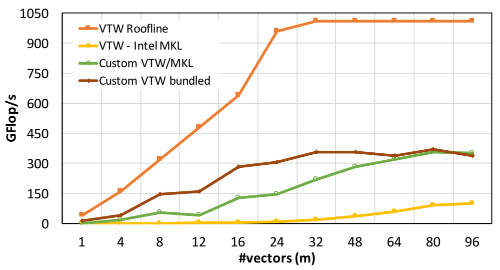
[7] A. Stathopoulos and J. R. McCombs, "A parallel, block, Jacobi–Davidson implementation for solving large eigenproblems on coarse grain environment," in *PDPTA*, 1999, pp. 2920–2926.

[8] 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.

[9] Y. Saad, "On the rates of convergence of the Lanczos and the block-Lanczos methods," *SIAM Journal on Numerical Analysis*, vol. 17, no. 5, pp. 687–706, 1980.

[10] A. Pinar and M. T. Heath, "Improving performance of sparse matrix–vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 30.

[11] E.-J. Im, "Optimizing the performance of sparse matrix–vector multiplication," Ph.D. dissertation, UC Berkeley, 2000.



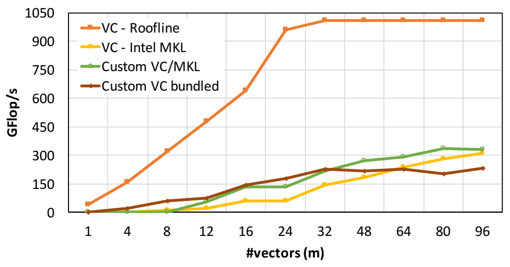


Fig. 11: Performance of V^TW (left) and VC (right) kernels, using the MKL library, as well as our custom implementations on an Intel Xeon Phi processor. Local vector blocks are $l \times m$, where $l = 1M$.

[12] 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.

[13] 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.

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

[15] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

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

[17] 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.

[18] 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.

[19] M. Röhrig-Zöllner et al., "Increasing the performance of the Jacobi–Davidson method by blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C697–C722, 2015.

[20] H. Anzt, S. Tomov, and J. Dongarra, "Energy efficiency and performance frontiers for sparse computations on gpu supercomputers," in *Proceedings of the sixth international workshop on programming models and applications for multicores and manycores*. ACM, 2015, pp. 1–10.

[21] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1213–1222.

[22] 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.

[23] P. Maris, M. Sosonkina, J. P. Vary, E. Ng, and C. Yang, "Scaling of ab-initio nuclear physics calculations on multicore computer architectures," *Procedia Computer Science*, vol. 1, no. 1, pp. 97 – 106, 2010.

[24] P. Maris, H. M. Aktulga, S. Binder, A. Calci, Ü. V. Çatalyürek, J. Langhammer, E. Ng, E. Saule, R. Roth, J. P. Vary *et al.*, "No-Core CI calculations for light nuclei with chiral 2-and 3-body forces," *Journal of Physics: Conference Series*, vol. 454, no. 1, p. 012063, 2013.

[25] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le, "Accelerating configuration interaction calculations for nuclear structure," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–15.

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

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

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

[29] 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.

[30] H. Zha, O. Marques, and H. D. Simon, "Large-scale SVD and subspace methods for information retrieval," in *Solving Irregularly Structured Problems in Parallel*. Springer, 1998, pp. 29–42.

[31] H. M. Aktulga, C. Yang, E. Ng, P. Maris, and J. Vary, "Topology-aware mappings for large-scale eigenvalue problems," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science (LNCS), vol. 7484, 2012, pp. 830–842.

[32] 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*, vol. 26, pp. 2631–2651, 2013.

[33] 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.

[34] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon Phi," in *5th ACM/SPEC International Conference on Performance Engineering*. ACM, 2014, pp. 137–148.

[35] M. Shao, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver," *arXiv preprint arXiv:1609.01689*, 2016.

Hasan Metin Aktulga (PhD Purdue University, 2010) is an Assistant Professor of Computer Science and Engineering at Michigan State University. His research interests are in the areas of high performance computing, scientific computing, and numerical linear algebra.

Md. Afibuzzaman is a graduate student in the Department of Computer Science and Engineering at Michigan State University. His work on high performance computing and scientific computing.

Samuel Williams (PhD UC Berkeley, 2008) is a staff scientist in the Performance and Algorithms Research Group at the Lawrence Berkeley National Laboratory (LBNL). His research interests include high-performance computing, auto-tuning, performance modeling, computer architecture, and hardware/software co-design.

Aydın Buluç (PhD UCSB, 2010) is a Staff Scientist at LBNL. He works on high-performance graph analysis, parallel sparse matrix computations, communication-avoiding algorithms, with applications in computational genomics and biology. Previously, he was an Alvarez Fellow.

Meiyue Shao (PhD EPFL, 2014) is a postdoctoral fellow in the Scalable Solvers Group at LBNL. His interests are numerical linear algebra and its applications, high performance computing and mathematical software, and electronic structure calculations.

Chao Yang (PhD Rice University, 1998) joined LBNL in 2000, and is currently a Senior Scientist. His expertise is in numerical linear algebra, optimization, large-scale data analysis and high performance computing.

Esmong Ng (PhD University of Waterloo, 1983) is currently the Head of the Applied Mathematics Department at LBNL. He has more than 35 years of experience in high-performance numerical algorithms and scientific computing, including 30 years of working with and at Department of Energy national laboratories.

Pieter Maris (PhD University of Groningen, 1993) is a Research Associate Professor at the Dept. of Physics and Astronomy at Iowa State University. His research interests are in the area of computational nuclear physics.

James Vary (PhD Yale University, 1970) is a Professor of Physics at Iowa State University. His research activities span strong interaction physics from ab-initio nuclear structure theory to include fundamental tests of nature's symmetries and to nuclear applications of Quantum Chromodynamics (QCD).