DeepSparse: A Task-parallel Framework for Sparse Solvers on Deep Memory Architectures

Md Afibuzzaman[⋆,†], Fazlay Rabbi[⋆,†], M. Yusuf Özkaya[‡], Hasan Metin Aktulga[†], Ümit V. Çatalyürek[‡]
[†]*Computer Science & Engineering, Michigan State University*
[‡]*School of Computational Science & Engineering, Georgia Institute of Technology*
{afibuzza, rabbimd, hma}@msu.edu, {myozka,umit}@gatech.edu
[⋆]Co-first authors

***Abstract*—Data movement is an important bottleneck against efficiency and energy consumption in large-scale sparse matrix computations that are commonly used in linear solvers, eigensolvers and graph analytics. We introduce a novel task-parallel sparse solver framework, named DeepSparse, which adopts a fully integrated task-parallel approach. DeepSparse framework differs from existing work in that it adopts a holistic approach that targets all computational steps in a sparse solver rather than narrowing the problem into small kernels (*e.g.*, SpMM, SpMV). We present the implementation details of DeepSparse and demonstrate its merit in two popular eigensolvers, LOBPCG and Lanczos algorithms. We observe that DeepSparse achieves 2× - 16× fewer cache misses across different cache layers (L1, L2 and L3) over implementations of the same solvers based on optimized library function calls. We also achieve 2× - 3.9× improvement in execution time when using DeepSparse over the same library versions.**

***Index Terms*—Sparse solvers, task parallelism, data dependency graphs, performance optimization.**

I. Introduction

Sparse matrix computations, in the form of solvers for systems of equations, eigenvalue problems or matrix factorizations, constitute the main kernel in fields as diverse as computational fluid dynamics (CFD), quantum many-body problems, machine learning and graph analytics. The scale of problems in these scientific applications typically necessitates execution on massively parallel architectures. Moreover, sparse matrices come in very different forms and properties depending on application area. However, due to the irregular data access patterns and low arithmetic intensities of sparse matrix computations, achieving high performance and scalability is very difficult. These challenges are further exacerbated by the increasingly complex deep memory hierarchies of the modern architectures as they typically integrate several layers of memory storage. While exact specifications and number of layers change as architectures evolve, the underlying principle of memory hierarchy stays the same: Going farther away from the processor, memory capacity increases at the expense of increased latency and reduced bandwidth. As such, minimizing data movement across layers of the memory and overlapping data movement with computations are keys to achieving high performance in sparse matrix computations.

Unlike its dense matrix analogue, the state of the art for sparse matrix computations is lagging far behind. The widening gap between the memory system and processor performance, irregular data access patterns and low arithmetic intensities of sparse matrix computations have effectively made them "memory-bound" computations. Furthermore, the downward trend in memory space and bandwidth per core in high performance computing (HPC) systems [1] has paved the way for a deepening memory hierarchy. Thus, there is a dire need for new approaches both at the algorithmic and runtime system levels for sparse matrix computations.

In this paper, we propose a novel sparse linear algebra framework, named *DeepSparse*, which aims to accelerate sparse solver codes on modern architectures with deep memory hierarchies. Our proposed framework differs from existing work in two ways. First, we propose a holistic approach that targets all computational steps in a sparse solver rather than narrowing the problem into a single kernel, e.g., sparse matrix vector multiplication (SpMV) or sparse matrix multiple vector multiplication (SpMM). Second, we adopt a fully integrated task-parallel approach while utilizing commonly used sparse matrix storage schemes.

In a nutshell, DeepSparse provides a GraphBLAS plus BLAS/LAPACK-like frontend for domain scientists to express their algorithms without having to worry about the architectural details (e.g., memory hierarchy) and parallelization considerations (i.e., determining the individual tasks and their scheduling) [2]–[4]. DeepSparse automatically generates and expresses the entire computation as a task dependency graph (TDG) where each node corresponds to a specific part of a computational kernel and edges denote control and data dependencies between computational tasks. We chose to build DeepSparse on top of OpenMP [5] because OpenMP is the most commonly used shared memory programming model, but more importantly it supports task-based data-flow programming abstraction. As such, DeepSparse relies on OpenMP for parallel execution of the TDG.

We anticipate two main advantages of DeepSparse over

a conventional bulk synchronous parallel (BSP) approach where each kernel relies on loop parallelization and is optimized independently. First, DeepSparse would be able to expose better parallelism as it creates a global task graph for the entire sparse solver code. Second, since the OpenMP runtime system has explicit knowledge about the TDG, it may be possible to leverage a pipelined execution of tasks that have data dependencies, thereby leading to better utilization of the hardware cache.

To summarize, the main contributions of this paper are:

- Introduction of a novel task-parallel sparse solver framework with complete details in regards to the front-end API and description of how a solver code expressed through this API is automatically converted into a task graph,
- an extensive evaluation of the performance of DeepSparse on two applications using a variety of sparse matrices from different domains,
- demonstration that the proposed task-parallel framework can significantly reduce misses across all cache levels (up to 16×) as well as improve the execution time (by up to 3.9×) compared to highly optimized library implementations based on the BSP model.

The rest of this paper is organized as follows: Section II describes the related work and Section III explains different components of the DeepSparse framework. Section IV presents the applications we used to evaluate DeepSparse. Section V describes the experimental environment and evaluates the proposed framework in terms of execution time and effective last level cache utilization performance. Finally, Section VI concludes our presentation with a summary and future work.

II. Related Work

The most fundamental operation in sparse linear algebra is arguably the multiplication of a sparse matrix with a vector (SpMV), as it forms the main computational kernel for several applications, such as, the solution of partial differential equations (PDE) [6] and the Schrödinger Equation [7] in scientific computing, spectral clustering [8] and dimensionality reduction [9] in machine learning, the Page Rank algorithm [10] in graph analytics, and many others. The Roofline model by Williams et al. [11] suggests that the performance of SpMV kernel is ultimately bounded by the memory bandwidth. Consequently, performance optimizations to increase cache utilization and reduce data access latencies for SpMV has drawn significant interest, e.g., [12]–[15].

A closely related kernel is the multiplication of a sparse matrix with multiple vectors (SpMM) which constitutes the main operation in block solvers, e.g., the block Krylov subspace methods [6], [16] and block Jacobi-Davidson method. SpMM has much higher arithmetic intensity than SpMV and can efficiently leverage wide vector execution units. As a result, SpMM-based solvers has recently drawn significant interest in scientific computing [17]–[24]. SpMM also finds applications naturally in machine learning where several features (or eigenvectors) of sparse matrices are needed [8], [9]. Although SpMM has a significantly higher arithmetic intensity than SpMV, the extended Roofline model that we recently proposed suggests that cache bandwidth, rather than the memory bandwidth, can still be an important performance limiting factor for SpMM [17].

LAPACK [3] is a linear algebra library for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. LAPACK routines mostly exploit Basic Linear Algebra Subprograms (BLAS) to solve these problems. PLASMA aims to overcome the shortcomings of the LAPACK library in efficiently solving the problems in dense linear algebra on multicore processors [25], [26]. PLASMA can solve dense general systems of linear equations, symmetric positive definite systems of linear equations and linear least squares problems, using LU, Cholesky, QR and LQ factorizations and supports both single precision and double precision arithmetic. However, PLASMA does not support general sparse matrices and does not solve sparse eigenvalue or singular value problems. PLASMA supports only shared-memory machines.

MAGMA is a dense linear algebra library (like LAPACK) for heterogeneous systems, i.e., systems with GPUs [27]–[29], to fully exploit the computational power that each of the heterogeneous components would offer. MAGMA provides very similar functionality like LAPACK and makes it easier for the user to port their code from LAPACK to MAGMA. MAGMA supports both CPU and GPU interfaces. The users do not have to know details of GPU programming to use MAGMA.

Barrera *et. al.* [30] use computational dependencies and dynamic graph partitioning method to minimize NUMA effect on shared memory architectures. StarPU [31] is a runtime system that facilitates the execution of parallel tasks on heterogeneous computing platforms, and incorporates multiple scheduling policies. However, the application developer has to create the computational tasks by themselves in order to use StarPU.

While the concept of task parallelism based on data flow dependencies is not new, exploration of the benefits of this idea in the context of sparse solvers constitutes a novel aspect of this work. Additionally, to the best of our knowledge, related work on task parallelism has not explored its impact on cache utilization compared to the BSP model as we do in this work.

III. DeepSparse Overview

Figure 1 illustrates the architectural overview of DeepSparse. As shown, DeepSparse consists of two major components: i) *Primitive Conversion Unit* (PCU) which provides a front-end to domain scientists to express their application at a high-level; and ii) *Task Executor* which creates the actual tasks based on the abstract task graph

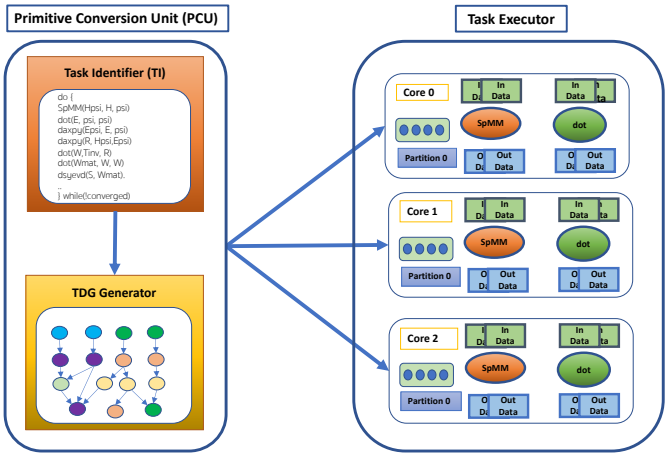

Fig. 1. Schematic overview of DeepSparse.

generated by PCU and hands them over to the OpenMP runtime for execution.

As sparse matrix related computations represent the most expensive calculations in many large-scale scientific computing, we define tasks in our framework based on the decomposition of the input sparse matrices. For most sparse matrix operations, both 1D (block row) and 2D (sparse block) partitioning are suitable options. A 2D partitioning is ideal for exposing high degrees of parallelism and reducing data movement across memory layers [32], as such 2D partitioning is the default scheme in DeepSparse. For a 2D decomposition, DeepSparse defines tasks based on the Compressed Sparse Block (CSB) [13] representation of the sparse matrix, which is analogous to tiles that are commonly used in task parallel implementation of dense linear algebra libraries. However, CSB utilizes much larger block dimensions (on the order of thousands) due to sparsity [13], [17], [33]. Consequently, DeepSparse starts out by decomposing the sparse matrix (or matrices) for a given code into CSB blocks (which eventually corresponds to the tasks during execution with each kernel producing a large number of tasks). Note that the decomposition of a sparse matrix dictates partitioning of the input and output vectors (or vector blocks) in the computation as well, effectively inducing decomposition of all data structures used in the solver code.

DeepSparse creates and maintains fine-grained dependency information across different kernels of a given solver code based on the result of the above decomposition scheme. As such, instead of simply converting each kernel into its own task graph representation and concatenating them, DeepSparse generates a global task graph, allowing for more optimal data access and task scheduling decisions based on global information. Since the global task graph depends on the specific algorithm and input sparse matrix, DeepSparse will explicitly generate the corresponding task dependency graph. While this incurs some computational and memory overheads, such overheads are negligible. The main reason for computational overheads to be negligible is that sparse solvers are typically iterative, and the same task dependency graph is used for several iterations. The reason why memory overheads is negligible is that each vertex in the task graph corresponds to a large set of data in the original problem. After this brief overview, we explain the technical details in DeepSparse.

A. Primitive Conversion Unit (PCU)

PCU is composed of two parts: i) Task Identifier, and ii) Local Task Dependency Graph (TDG) Generator.

1) Task Identifier (TI): The application programming interface (API) for DeepSparse is a combination of the recently proposed GraphBLAS interface [2] (for sparse matrix related operations) and BLAS/LAPACK [3], [4] (for vector and occasional dense matrix related computations). This allows application developer to express their algorithms at a high-level without having to worry about architectural details (e.g., memory hierarchy) or parallelization considerations (e.g., determining the individual tasks and their scheduling). Task identifier parses a code expressed using the DeepSparse API to identify the specific BLAS/LAPACK and GraphBLAS calls, as well as the input/output of each call. It then passes this information to the local task dependency graph generator.

TI builds two major data structures:

(i) ***ParserMap***: `ParserMap` is an unordered map that holds the parsed data information in the form of (`Key`, `Value`) pairs. As TI starts reading and processing the DeepSparse code, it builds a `ParserMap` from the function calls. To uniquely identify each call in the code, *Key* class is made up of three components: `opCode` which is specific to each type of operation used in the code, `id` which keeps track of the order of the same function call in the code (e.g., if there are two matrix addition operations, then the first call will have id = 1 and the second one will have id = 2), and `timestamp` which stores the line number of the call in the code and is used to detect the input dependencies of this call to the ones upstream. For each key, the corresponding `Value` object stores the input and output variable information. It also stores the dimensions of the matrices involved in the function call.

(ii) ***Keyword & idTracker***: `Keyword` is a vector of strings that holds the unique function names (*i.e.*, `cblas_dgemm, dsygv, mkl_dcrmm`, etc.) that have been found in the given code, and the `idTracker` keeps track of the number of times that function (Keyword) has been called so far. Keyword and idTracker vectors are synchronized with each other. When TI finds a function call, it searches for the function name in the *Keyword* vector. If found, the corresponding idTracker index is incremented. Otherwise, the *Keyword* vector is expanded with a corresponding initial *idTracker* value of 1.

Listing 1. TaskInfo Structure

```
struct TaskInfo
{
  int opCode; //type of operation
  int numParamsCount;
  int *numParamsList; //tile id, dimensions etc.
  int strParamsCount;
  char **strParamsList; //i.e. buffer name
  int taskID; //analogous to id of Key Class
}
```

2) Task Dependency Graph Generator (TDGG): The output of Task Identifier (TI) is a dependency graph at a very coarse-level, *i.e.*, at the function call level. For an efficient parallel execution and tight control over data movement, tasks must be generated at a much finer granularity. This is accomplished by the Task Dependency Graph (TDGG), which goes over the input/output data information generated by TI for each function call and starts decomposing these data structures. As noted above, the decomposition into finer granularity tasks starts with the first function call involving the sparse matrix (or matrices) in the solver code which is typically an SpMV, SpMM or SpGEMM operation. After tasks for this function call are identified by examining the non-zero pattern of the sparse matrix, tasks for prior and subsequent function calls are generated accordingly. As part of task dependency graph generation procedure, TDGG also generates the dependencies between individual fine-granularity tasks by examining the function call dependencies determined by TI. Note that the dependencies generated by TDGG may (and often do) span function boundaries and this is an important property of DeepSparse that separates it from a bulk synchronous parallel (BSP) program which effectively imposes barriers at the end of each function call.

The resulting task dependency graph generated by TDGG is essentially a directed acyclic graph (DAG) representing the data flow in the solver code where vertices denote computational tasks, incoming edges represent the input data and outgoing edges represent the output data for each task. TDGG also labels the vertices in the task dependency graph with the estimated computational cost of each task, and the directed edges with the name and size of the corresponding data, respectively. During execution, such information can be used for load balancing among threads and/or ensuring that active tasks fit in the available cache space. In this initial version of DeepSparse though, such information is not yet used because we rely on OpenMP's default task execution algorithms, as explained next.

B. Task Executor

To represent a vertex in the task graph, TDGG uses an instance of the TaskInfo structure [listing 1] which provides all the necessary information for the Task executor to properly spawn the corresponding OpenMP task. The task executor receives an array of TaskInfo structures [listing 1] from the PCU that represents the full computational dependency graph, picks each node from this array one by one and extracts the corresponding task information. DeepSparse implements OpenMP task based functions for all computational kernels (represented by `opCode`) it supports. Based on the `opCode`, partition id of the input/output data structures and other required parameters (given by `numParamsList` and `strParamsList`) found in the TaskInfo structure at hand, Task Executor calls the necessary computational function found in the DeepSparse library, effectively spawning an OpenMP task.

Algorithm 1: SpMM Kernel

Input: X[i,j] ($\beta \times \beta$, Sparse CSB block), Y[j] ($\beta \times b$)
Output: Z[i] (Dense vector block, $\beta \times b$)

```
1  #pragma omp task depend(in: X[i,j], Y[j], Z[i])
     depend(out: Z[i])
2  {
3  foreach val ∈ X[i,j].nnz do
4  |   r = X[i,j].row_loc[val]
5  |   c = X[i,j].col_loc[val]
6  |   for k = 0 to b do
7  |   |   Z[r × b + k] = Z[r × b + k] + val × Y[c × b + k]
8  |   end
9  end
10 }
```

In DeepSparse, the master thread spawns all OpenMP tasks one after the other, and relies on OpenMP's default task scheduling algorithms for execution of these tasks. OpenMP's Runtine Environment then determines which tasks are ready to be executed based on the provided task dependency information. When ready, those tasks are executed by any available member of the current thread pool (including the master thread). Note that OpenMP supports task parallel exeuction with named dependencies, and better yet these dependencies can be specified as variables. This feature is fundamental for DeepSparse to be able to generate TDGs based on different problem sizes and matrix sparsity patterns. This is exemplified in Algorithm 1, where SpMM tasks for the compressed sparse block at row i and j is simply invoked by providing the $X[i,j]$ sparse matrix block along with $Y[j]$ input vector block and $Z[i]$ output vector block in the *depend* clause.

An important issue in a task parallel program is the data race conditions involving the output data that is being generated. Fortunately, the task-parallel execution specifications of OpenMP requires only one thread to be active among threads writing into the same output data location. While this ensures a race-condition free exeuction, it might hinder performance due to a lack of parallelism. Therefore, for data flowing into tasks with a high incoming degree, DeepSparse allocates temporary output data buffers based on the number of threads and the available memory space. Note that this also requires

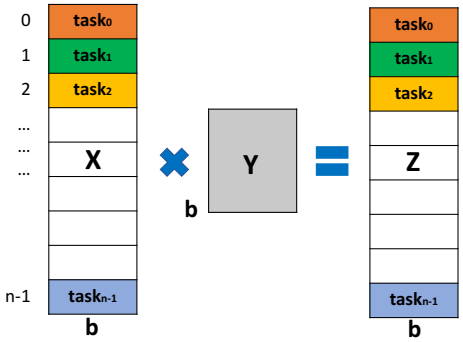

Fig. 2. Overview of input output matrices partitioning of task-based matrix multiplication kernel.

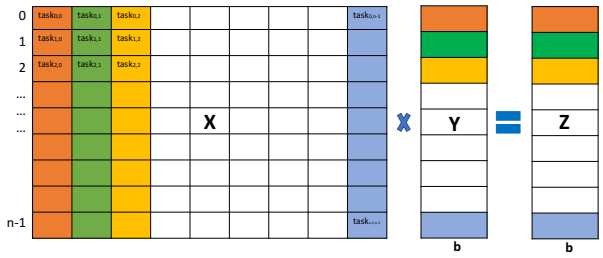

Fig. 3. Overview of matrices partitioning of task-based SpMM kernel.

the creation of an additional task to reduce the data in temporary buffer space before it is fed into the originally intended target task.

C. Illustrative Example

We provide an example to demonstrate the operation of DeepSparse using the simple code snippet provided in Listing 2. As TI parses the sample solver code, it discovers that the first `cblas_dgemm` in the solver corresponds to a linear combination operation (see Fig. 2), the second line is a sparse matrix vector block multiplication (SpMM, see Fig. 3) and the second `cblas_dgemm` at the end is an inner product of two vector blocks (see Fig. 4). These function calls, their parameters as well as dependencies are captured in the ParserMap, Keyword, and idTracker data structures as shown in Table I.

Listing 2. An example pseudocode

```
cblas_dgemm(CblasRowMajor, CblasNoTrans,
    CblasNoTrans, m, n, k, 1.0, A, k, B, n, 0,
    C, n);
SpMM(X, C, D, m, n);
cblas_dgemm(CblasRowMajor, CblasTrans,
    CblasNoTrans, n, n, m, 1.0, D, n, C, n, 0,
    E, n);
```

Task Dependency Graph (TDG) generator receives the necessary information from TI and determines the tasks corresponding to partitionings of operand data structures of each operation, as well as their origins (whether the necessary data are coming from another task or from a variable). TDGG then builds the DAG of each computational kernel and appends it to the global DAG with

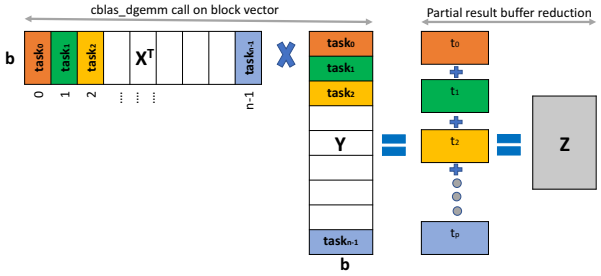

Fig. 4. Overview of matrices partitioning of task-based inner product kernel.

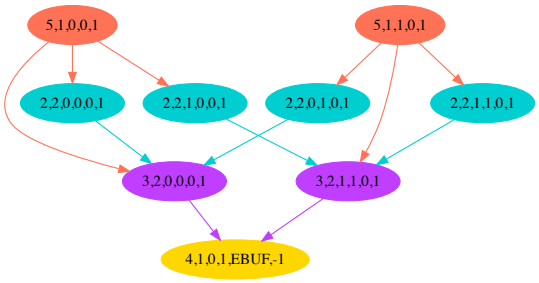

Fig. 5. Task graph for the psudocode in listing 2.

proper edge connectivity (*i.e.*, dependencies). While generating the DAG, the TDGG also encodes the value of the TaskInfo structure instance that represent each of the vertices into the vertex name. The vertex naming convention is *<opCode, numParamsCount, numParamsList, strParamsCount, strParamsList, taskID>*. Figure 5 shows the task dependency graph of the solver code in Listing 2 (assuming $m = 100$, $k = 8$, $n = 8$, $CSBtile/blocksize = 50$, so each input matrix is partitioned into 2 chunks).

The task executor receives an array of TaskInfo structures that contains the node information as shown in Figure 5. The task executor goes over each of the tasks in the array of TaskInfo structure. At first, it reads the nodes (*<5,1,0,0,1>*, *<5,1,1,0,1>*) of the first operation and spawns two matrix multiplication (xY) tasks with proper input output matrices. The task executor then reads all the task information for all SpMM tasks {*<2,2,0,0,0,1>*, *<2,2,0,1,0,1>*, *<2,2,1,0,0,1>*, *<2,2,1,1,0,1>*} and spawns four SpMM tasks with proper input/output matrix blocks. Finally, the task executor reads *<3,2,0,0,0,1>*, *<3,2,1,1,0,1>* and *<4,1,0,1,EBUF,-1>* and spawns two inner product (X^TY) tasks and one partial output buffer reduction task for the inner product operation.

D. Limitations of the Task Executor

Despite the advantages of an asynchronous task-parallel execution, the Task Executor has the following limitations:

- **Synchronization at the end of an iteration**: Most computations involving sparse matrices are based on iterative techniques. As such, the TDG generated for a single iteration can be reused over several steps (until the algorithm reaches convergence). However,

Data Structure	Content
ParserMap	<{XY, 1, 1},{<A, B>, <C>, <m, n, k>}>
	<{SpMM, 1, 2},{<X, C>, <D>, <m, m, n>}>
	<{XTY, 1, 3},{<D, C>, <E>, <m, n, n>}>
keyword	<XY, SpMM, XTY>
idTracker	<1, 1, 1>

TABLE I
MAJOR DATA STRUCTURES AFTER PARSING THIRD LINE.

it is necessary to introduce a `#pragma omp taskwait` at the end of each solver iteration and force all tasks of the current iteration to be completed to ensure computational consistency among different iterations of the solver. For relatively simple solvers, the `taskwait` clause adds some overhead to the total execution time due to threads idling at `taskwait`s.

- **Limited number of temporary buffers**: While OpenMP allows the use of program variables in the dependency clauses, it does not allow dynamically changing the variable lists of the *depend* clauses. As such, the number of buffer lists in the partial output reduction tasks need to be fixed to overcome this issue. Depending on the available memory, there are at most $nbuf$ number of partial output buffers for a reduction operation. If $nbuf$ is less than the total number of threads, then there might be frequent read after write (RAW) contentions on partial output buffers. This could be have been potentially avoided, if the list of variables in the *depend* clause could have been dynamically changed.

IV. Benchmark Applications

We demonstrate the performance of the DeepSparse framework on two important eigensolvers widely used in large-scale scientific computing applications: Lanczos eigensolver [34] and Locally Optimal Block Preconditioned Conjugate Gradient algorithm (LOBPCG) [35].

a) Lanczos: Lanczos algorithm finds eigenvalues of a symmetric matrix by building a matrix $Q_k = [q_1, \ldots, q_k]$ of orthogonal Lanczos vectors [36]. The eigenvalues of the sparse matrix A is then approximated by the Ritz values. As shown in Algorithm 2, it is a relatively simple algorithm consisting of an Sparse Matrix Vector Multiplication (SpMV) along with some vector inner products for orthonormalization.

b) LOBPCG: LOBPCG is a commonly used block eigensolver based on the SpMM kernel [35], see Figure 3 for a pseudocode. Compared to Lanczos, LOBPCG comprises high arithmetic intensity operations (SpMM and Level-3 BLAS). In terms of memory, while the $\widehat{H}$ matrix takes up considerable space, when a large number of eigenpairs are needed (e.g. dimensionality reduction, spectral clustering or quantum many-body problems), memory needed for block vector Ψ can be comparable to or even greater than that of $\widehat{H}$. In addition, other block vectors (residual R, preconditioned residual W, previous direction P), block vectors from the previous iteration and the preconditioning matrix T must be stored, and accessed at each iteration. Figure 6 shows a sample task graph for LOBPCG generated by TDGG using a very small matrix. Clearly, orchestrating the data movement in a deep memory hierarchy to obtain an efficient LOBPCG implementation is non-trivial.

Algorithm 2: Lanczos Algorithm in Exact Arithmetic

$q_1 = b/||b||_2, \beta_0 = 0, q_0 = 0$
for $j = 1$ *to* k **do**
  $z = Aq_j$
  $\alpha_j = {q_j}^T z$
  $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  $\beta_j = ||z||_z$
  if $\beta_j = 0$, quit
  $q_{j+1} = z/\beta_j$
  Compute eigenvalues, eigenvectors, and error bounds of T_k
end

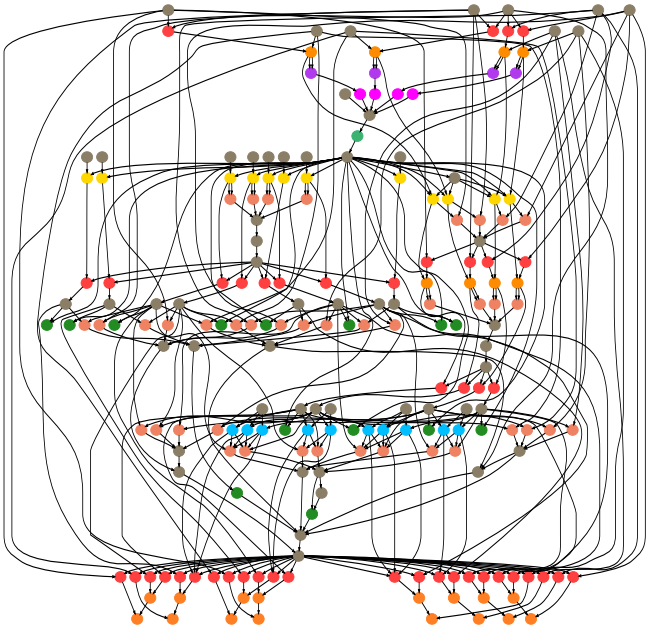

Fig. 6. A sample task graph for the LOBPCG algorithm using a small sparse matrix.

V. Performance Evaluation

A. Experimental Setup

We conducted all our experiments on Cori Phase I, a Cray XC40 supercomputer at NERSC, mainly using the GNU compiler. Each Cori Phase I node has two sockets with a 16-core Intel Xeon Processor E5-2698 v3 Haswell CPUs. Each core has a 64 KB private L1 cache (32 KB instruction and 32 KB data cache) and a 256 KB private L2 cache. Each CPU has a 40 MB shared L3 cache (LLC).

Algorithm 3: 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: Ψ_0, a block of vectors of dimensions of $N \times m$
Output: Ψ and E such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I_m$

1 Orthonormalize the columns of Ψ_0
2 $P_0 \leftarrow 0$
3 **for** *i = 0, 1, . . . , until convergence* **do**
4 $\quad E_i = \Psi_i^T \hat{H} \Psi_i$
5 $\quad R_i \leftarrow \hat{H}\Psi_i - \Psi_i E_i$
6 $\quad$ Apply the Rayleigh–Ritz procedure on $\text{span}\{\Psi_i, R_i, P_i\}$
7 $\quad \Psi_{i+1} \leftarrow \underset{S \in \text{span}\{\Psi_i . R_i, P_i\},\ S^T S = I_m}{\text{argmin}} \text{trace}(S^T \hat{H} S)$
8 $\quad P_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$
9 $\quad$ Check convergence
10 **end**

We use thread affinity to bind threads to cores and use a maximum of 16 threads to avoid NUMA issues. We test DeepSparse using five matrices with different size, sparsity patterns and domains (see Table II). The first 4 matrices are from The SuitSparse Matrix Collection and the Nm7 matrix is from nuclear no-core shell model code MFDn.

We compare the performance of DeepSparse with two other library implementations: i) **libcsr** is implementation of the benchmark solvers using thread-parallel Intel MKL Library calls (including SpMV/SpMM) with CSR storage of the sparse matrix, ii) **libcsb** is an implementation again using Intel MKL calls, but with the matrix being stored in the CSB format. Performance data for LOBPCG is averaged over 10 iterations, while the number of iterations is set to 50 for Lanczos runs. Our performance comparison criteria are L1, L2, LLC misses and execution times for both solvers. All cache miss data was obtained using the Intel VTune software.

Performance of the DeepSparse and libcsb implementations depends on the CSB block sizes used. Choosing a small block size creates a large number of small tasks. While this is preferable on a highly parallel architecture, the large number of tasks may lead to significant task execution overheads, in terms of both cache misses and execution times. Increasing the block size reduces such overheads, but this may then lead to increased thread idle times and load imbalances. Therefore, the CSB block size is a parameter to be optimized based on the specific problem. Different block sizes we experimented with have been 1K, 2K , 4K, 8K and 16K.

B. LOBPCG Evaluation

In Fig. 7, we show the number of cache misses at all three levels (L1, L2 and L3) and execution time comparison between all three versions of the LOBPCG algorithm compiled using the GNU compiler. LOBPCG is a complex algorithm with a number of different kernel types; its task graph results in millions of tasks for a single iteration. As shown in Fig. 7, except for the Nm7 matrix, libcsb and libcsr versions achieve similar number of cache misses; for Nm7, libcsb has important cache miss reductions over the libcsr version. On the other hand, DeepSparse achieves 2.5× - 10.7× fewer L1 misses, 6.5× - 16.2× fewer L2 misses and 2× - 7× fewer L3 cache misses compared to the libcsr version. As the last row of Fig. 7 shows, even with the implicit task graph creation and execution overheads of DeepSparse, the significant reduction in cache misses leads to 1.2× - 3.9× speedup over the execution times of libcsr. Given the highly complex DAG of LOBPCG and abundant data re-use opportunities available, we attribute these improvements to the pipelined execution of tasks which belong to different computational kernels (see Fig. 8) but use the same data structures. We note that the Task Executor in DeepSparse solely relies on the default scheduling algorithm used in the OpenMP runtime environment. By making use of the availability of the entire global task graph and labeling information on vertices/edges, it might be possible to improve the performance of DeepSparse even further.

TABLE II
MATRICES USED IN OUR EVALUATION.

Matrix	Rows	Columns	Nonzeros
inline1	503,712	503,712	36,816,170
dielFilterV3real	1,102,824	1,102,824	89,306,020
HV15R	2,017,169	2,017,169	283,073,458
Queen4147	4,147,110	4,147,110	316,548,962
Nm7	4,985,422	4,985,422	647,663,919

C. Lanczos Evaluation

In Fig. 9, cache misses and execution time comparisons for different Lanczos versions are shown. Lanczos algorithm is much simpler than LOBPCG, it has much fewer types and numbers of tasks than LOBPCG (basically, one SpMV and one inner product kernel at each iteration). As such, there are not many opportunities for data re-use. In fact, we observe that DeepSparse sometimes leads to increases in cache misses for smaller matrices. However, for the Nm7 and HV15R matrices, which are the largest matrices among our benchmark set, we observe an improvement in cache misses, achieving up to 2.4× fewer L1 cache misses, 3.1× fewer L2 misses and 4.5× fewer L3 misses than libcsr. But most importantly, DeepSparse achieves up to 1.8× improvement in terms of execution time. We attribute the execution time improvement observed across the board to the increased degrees of parallelism exposed by the global task graph of DeepSparse, which is in fact highly critical for smaller matrices.

D. Compiler Comparison

For all of our experiments, we use OpenMP as our backend. To explore the impact of different task schedul-

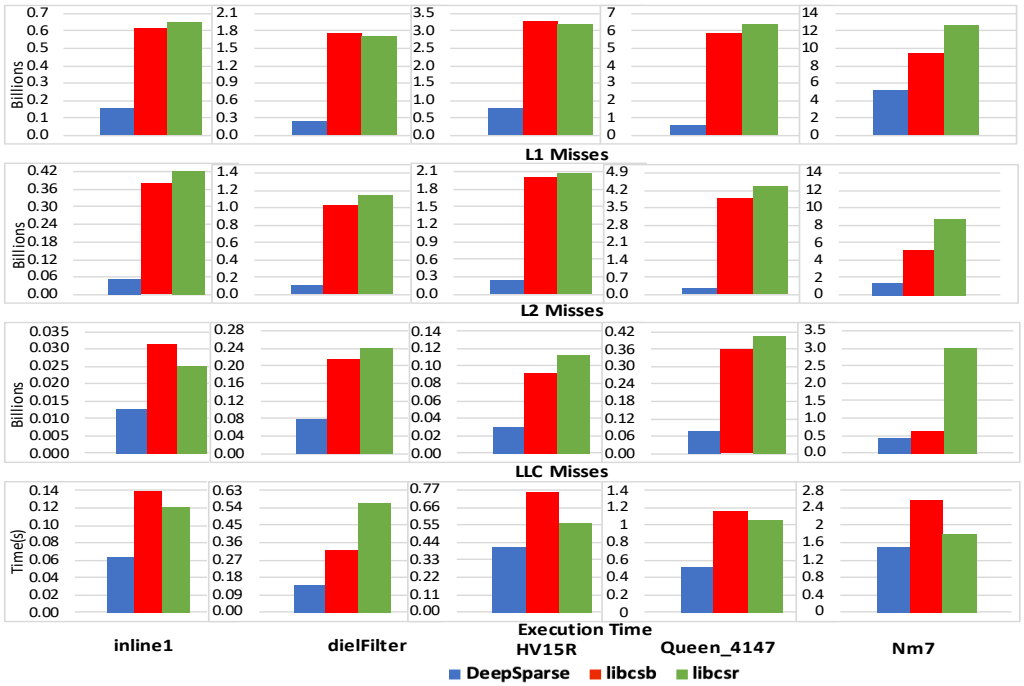

Fig. 7. Comparison of L1, L2, LLC misses and execution times between Deepsparse, libcsb and libcsr for the LOBPCG solver.

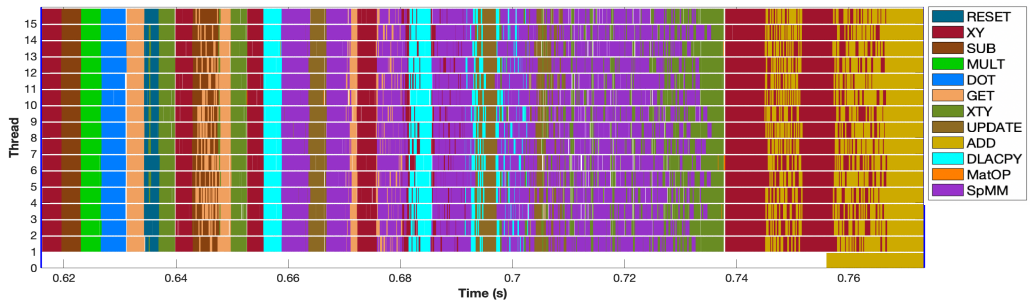

Fig. 8. LOBPCG single iteration execution flow graph of dielFilterV3real.

ing approaches in different OpenMP implementations, we have experimented with three compilers: Intel, GNU and Clang/LLVM compilers. In Figure 10, we show the comparison in execution time among different compilers for the three implementations. We see that the execution time for the Clang/LLVM compiler is significantly higher compared to GNU and Intel compilers for all matrices. However, cache misses stay pretty much the same when one moves to a different compiler. We show the cache miss comparison between the three compilers in Figure 11 for one matrix, HV15R. All other matrices follow a similar cache miss pattern like HV15R. Here, we can clearly see that regardless of the compiler, DeepSparse achieves fewer cache misses over libcsb and libcsr implementations. We can see that Clang/LLVM shows fewer cache misses for DeepSparse as well, but it eventually has a poor running time. We believe that this is because Clang/LLVM is not able to schedule tasks as efficiently as GNU and Intel. Compared to Intel compiler , GNU compiler sometimes shows more L1 and L2 misses. But the execution time is higher in Intel. This may be due to the scheduling strategy and and the implementation of task scheduling points in the compilers. Overall, GNU does best with running times among the three compilers, and Intel compilers do not do well with the library based solver implementations.

VI. Conclusion

This work introduces a novel task-parallel sparse solver framework which targets all computational steps in sparse solvers. We show that our approach achieves significantly fewer cache misses across different cache layers and also improves the execution time over the library versions. Future works will be in the direction of further reducing the cache misses and execution time over the current versions by experimenting with more advanced partitioning and scheduling algorithms compared to the default schemes in OpenMP.

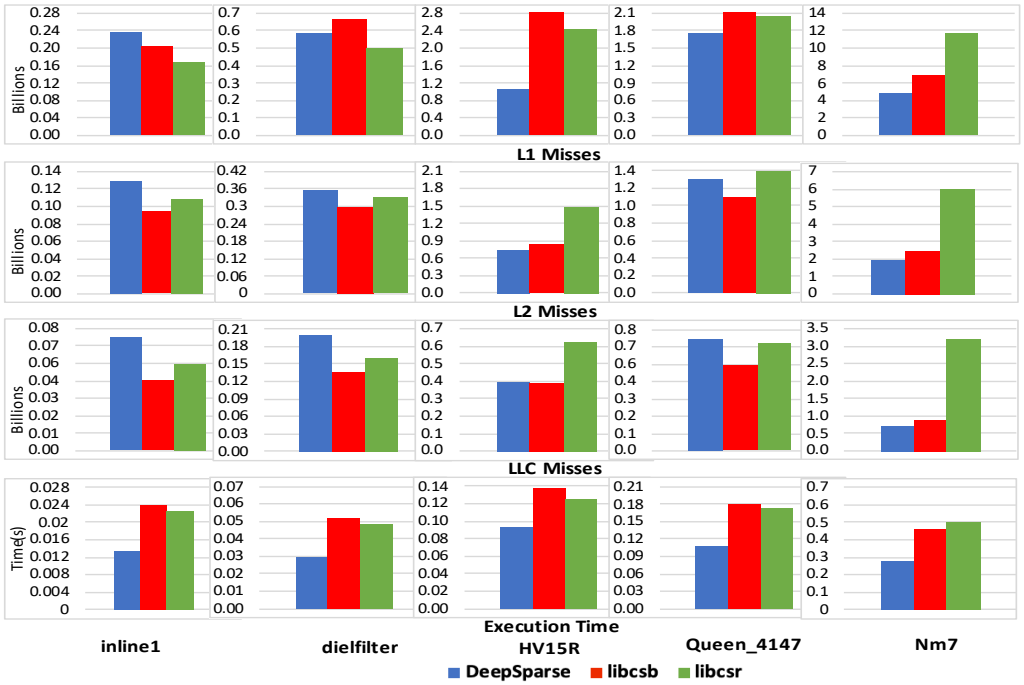

Fig. 9. Comparison of L1, L2, LLC misses and execution times between Deepsparse, libcsb and libcsr for the Lanczos solver.

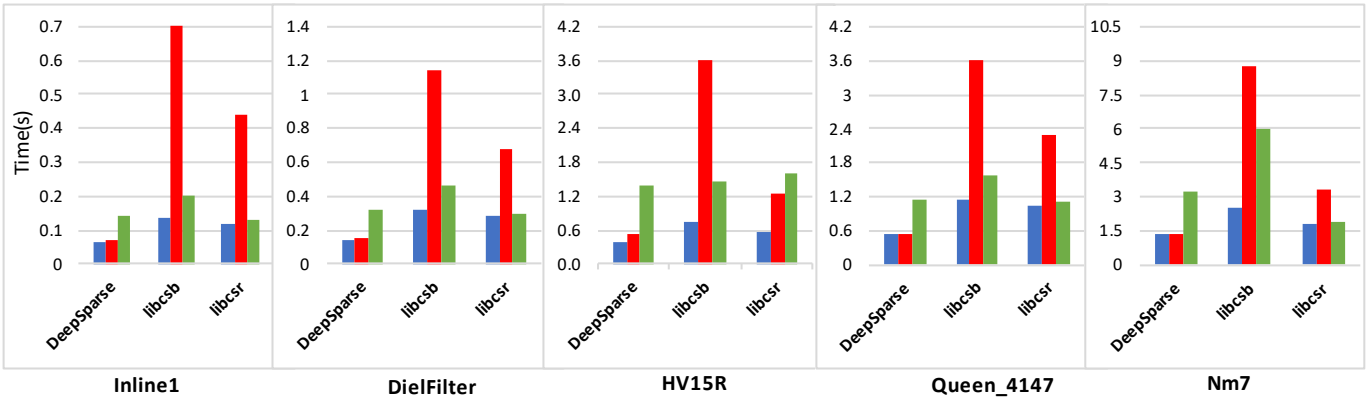

Fig. 10. Comparison of execution time for different compilers between Deepsparse , libcsb and libcsr for Lanczos Algorithm. (Blue/Left: GNU, Red/Middle: Intel, Green/Right: Clang compiler.)

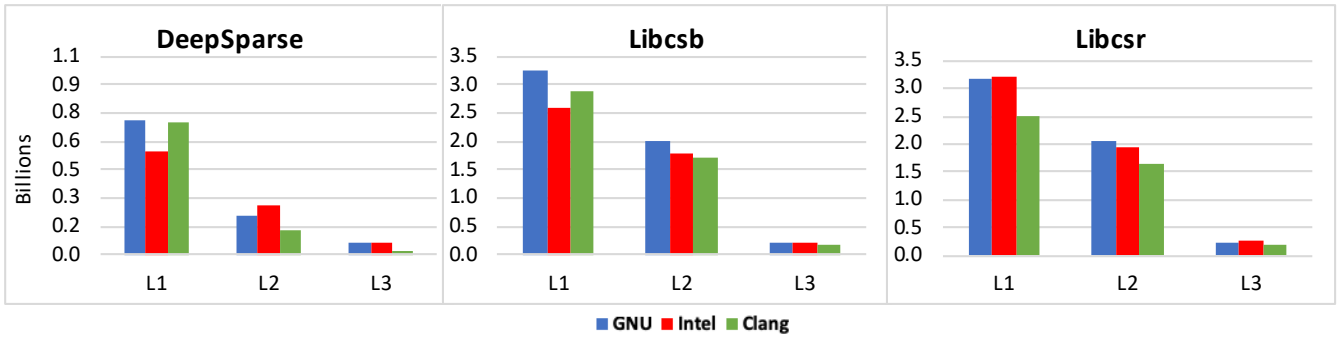

Fig. 11. Cache Miss comparison between compilers for HV15R

Acknowledgments

This work was in part supported by the NSF under awards CCF-1822932 and OAC-1845208, as well as the US Department of Energy, Office of Science under the award DE-SC0018083 (NUCLEI SciDAC-4 Collaboration). Computational resources were provided by the National Energy Research Scientific Computing Center (NERSC).

References

[1] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the" new normal" for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[2] J. Kepner, D. Bade, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney *et al.*, "Lapack users' guide, vol. 9," *Society for Industrial Mathematics*, vol. 39, 1999.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[5] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[6] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.

[7] M. Feit, J. Fleck Jr, and A. Steiger, "Solution of the schrödinger equation by a spectral method," *Journal of Computational Physics*, vol. 47, no. 3, pp. 412–433, 1982.

[8] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp. 849–856.

[9] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the Association for Computing Machinery*, 2009.

[12] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.

[13] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 233–244.

[14] E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[15] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *Parallel Processing, 2004. ICPP 2004. International Conference on*. IEEE, 2004, pp. 169–176.

[16] A. El Guennouni, K. Jbilou, and A. Riquet, "Block krylov subspace methods for solving large sylvester equations," *Numerical Algorithms*, vol. 29, no. 1-3, pp. 75–96, 2002.

[17] 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.

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

[19] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, "Improving the performance of dynamical simulations via multiple right-hand sides," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 36–47.

[20] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Regularizing graph centrality computations," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106–119, 2015.

[21] M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske, "Increasing the performance of the jacobi–davidson method by blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C697–C722, 2015.

[22] Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and U. V. Catalyürek, "An out-of-core eigensolver on ssd-equipped clusters," in *2012 IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 248–256.

[23] H. M. Aktulga, M. Afibuzzaman, S. Williams, A. Buluç, M. Shao, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "A high performance block eigensolver for nuclear configuration interaction calculations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1550–1563, 2016.

[24] 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," *Computer Physics Communications*, vol. 222, pp. 1–13, 2018.

[25] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[26] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[27] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[28] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[29] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[30] I. S. Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas, "Reducing data movement on large shared memory systems by exploiting computation dependencies," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 207–217.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par - 15th International Conference on Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874. [Online]. Available: http://hal.inria.fr/inria-00384363

[32] E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, and Ü. V. Çatalyürek, "An out-of-core task-based middleware for data-intensive scientific computing," in *Handbook on Data Centers*. Springer, 2015, pp. 647–667.

[33] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[34] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[35] 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.

[36] J. W. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997.