Parallel Shift-Invert Spectrum Slicing on Distributed Architectures with GPU Accelerators

David B. Williams-Young
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, United States of America
dbwy@lbl.gov

Chao Yang
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, United States of America
cyang@lbl.gov

ABSTRACT

The solution of large scale eigenvalue problems (EVP) is often the computational bottleneck for many scientific and engineering applications. Traditional eigensolvers, such as direct (e.g. ScaLAPACK) and Krylov subspace (e.g. Lanczos) methods, have struggled in achieving high scalability on large computing resources due to communication and synchronization bottlenecks which are inherent in their implementation. This includes a difficulty in developing well-performing ports of these algorithms to architectures which rely on the use of accelerators, such as graphics processing units (GPU), for the majority of their floating point operations. Recently, there has been significant research into the development of eigensolvers based on spectrum slicing, in particular shift-invert spectrum slicing, to alleviate the communication and synchronization bottlenecks of traditional eigensolvers. In general, spectrum slicing trades the global EVP for many smaller, independent EVPs which may be combined to assemble some desired subset of the entire eigenspectrum. The result is a method which utilizes more floating point operations than traditional eigensolvers, but in a way which allows for the expression of massive concurrency leading to an overall improvement in time-to-solution on large computing resources. In this work, we will examine the performance of parallel shift-invert spectrum slicing on modern GPU clusters using state-of-the-art linear algebra software.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Applied computing** → *Physics.*

ACM Reference Format:
David B. Williams-Young and Chao Yang. 2020. Parallel Shift-Invert Spectrum Slicing on Distributed Architectures with GPU Accelerators. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404416

1 INTRODUCTION

Large-scale eigenvalue problems (EVP) have become ubiquitous in many areas of computational science and engineering such as quantum physics and chemistry simulations, materials design, particle accelerator modeling, structural engineering [46] and machine learning [4]. A prominent example of this type of EVP, and the primary target of this work, is the non-linear self-consistent field EVP commonly encountered in large scale electronic structure calculations such as those based on density functional theory. There has been a tremendous amount of effort to develop efficient parallel algorithms for computing all or a subset of eigenvalues and the corresponding eigenvectors of large dense or sparse matrices [3]. Many of these algorithms have been implemented efficiently, and can scale to hundreds or thousands of processors on traditional high performance computers. Eigenvalue problems that required hours to solve 10 years ago on a workstation can now be solved in a few minutes or seconds on distributed memory many-core parallel computers. However, the demand for even faster eigensolvers still remains.

The recent surge in the number of computing clusters equipped with accelerators such as graphics processing units (GPU) has completely transformed the landscape of high performance computing. Many linear algebra algorithms have been shown to benefit from such architectures and achieve significant performance improvements [12, 19, 28]. A natural question one would ask is whether similar performance gains can be achieved for eigensolvers on distributed memory architectures equipped with GPUs. The complete answer to this question is yet to be determined. In this paper, we report our latest effort to develop a symmetric eigensolver which is capable of leveraging the computational resources of large computing clusters with GPU accelerators.

GPUs exhibit a number of characteristics one must consider in developing efficient algorithms:

- high bandwidth (O(900 GB/s)) but low capacity (O(16 GB)) memory directly accessible from the device,
- low bandwidth (O(50 GB/s)) data transfers between host and device memory,
- and a large number of computational threads which allows for the expression of massive concurrency in comparison with modern CPU architectures (O(16x) FLOP/s per node over Intel Knight's Landing).

As a result, algorithms which are capable of expressing large amounts of concurrency with minimal data movement between host and device are best suited for GPU implementation. However, due to its low capacity, it is often the case that the memory requirement of a particular algorithm exceeds the capacity of a single GPU. As such, optimal implementations of these algorithms require minimization of the size and frequency of data transfers between host and device to ensure data locality and thus amortize the impact

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404416

of these low bandwidth transfers on the ability of the device to express concurrency.

Over the years, significant progress has been made in the development of efficient CPU implementations of dense eigensolvers for shared memory (e.g. LAPACK [1, 10]) and distributed memory architectures (e.g. ScaLAPACK [5], ELPA [30, 37], QDWH [36]). More recently, a sizable research effort has been afforded to the extension of these algorithms to single GPU [20] and multi-GPU [29, 42] implementations. As will be shown in this work, for matrices that can fit in the device memory of a single GPU, an efficient GPU implementation of a symmetric dense eigensolver can achieve a 2x-3x speedup over state of the art CPU implementations. However, distributed GPU implementations of dense eigensolvers demonstrate poor strong scaling in comparison with their CPU counterparts. This makes further improvements by utilizing more GPUs in a distributed environment more difficult.

The poor strong scaling of dense symmetric eigensolvers based on the reduction of the symmetric matrix to tridiagonal form through successive orthogonal transformations is fundamentally attributed to the communication and synchronization bottlenecks which are inherent in their implementation. Communication reducing techniques can be used in the QR-based Dynamically Weight Halley (QDWH) algorithm [39] which is a spectral divide-and-conquer approach that constructs a sequence of spectral projectors using approximations to matrix sign functions [2]. Alternative approximations to the matrix sign function via a more general Zolotov function [38] can also be used. But synchronization bottleneck still exists, at least at the top level of the divide-and-conquer tree. For GPU implementations of these algorithms, low capacity device memory and low bandwidth memory transfers between host and device further exacerbate these problems by impeding the concurrency which may be expressed in any one step of the reduction. As a result, strong scaling often stagnates (or inverts) at relatively low amounts of computational resources.

In this paper, we examine an alternative approach to the solution of the symmetric EVP which addresses the communication and synchronization bottlenecks of these types of dense eigensolvers. Our approach partitions the eigenvalues to be computed into several subintervals and computes eigenvalues and corresponding eigenvectors within each subinterval independently and such that the majority of the computation may be performed concurrently. We will refer to this approach as a *spectrum slicing* algorithm. Note that, there is some similarity between spectrum slicing and the spectral divide-and-conquer approach used in QWDH. However, unlike the recursive partition used in QWDH, the partition used in spectrum slicing is made all at once. Over the years, spectrum slicing has been examined by several research groups [21–23, 32, 44, 47]. The algorithms developed by these groups differ in how the spectrum is partitioned and what algorithm is used to compute eigenvalues within a subinterval.

In this work, we focus on the spectrum partition scheme developed in [44] due to its minimal communication overhead. We use the shift-invert subspace iteration (SISUBIT) to compute eigenvalues within each subinterval, thus the resulting method is referred to as shift-invert spectrum slicing (SISS). This approach has been

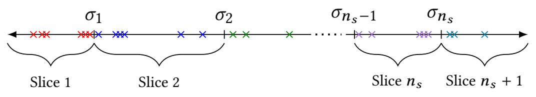

Figure 1: Partitioning the spectrum of interest into several slices or subintervals which may be computed simultaneously.

demonstrated to exhibit scalable performance in distributed memory architectures [21, 23, 44, 47], and its performance characteristics on many-core CPU architectures have been well documented. Further, due to its leverage of linear system solvers rather than orthogonal transformations, SISS may be effectively applied to sparse matrices as well. The ability to take advantage of the sparsity of the matrix is absent in a traditional dense eigensolver or the QWHD algorithm because the required QR factorization tends to destroy the sparsity of the original matrix. To achieve scalable performance on GPU accelerated systems (as with the analogous CPU implementations) we rely on efficient implementations of a dense or sparse linear equation solvers for GPU architectures to carry out the bulk of the computational work. Several GPU implementations of such solvers are available for shared (e.g. NVIDIA's cuSOLVER) and distributed memory (e.g. SuperLU_DIST [40], PaStiX [18], and the Watson Sparse Matrix Package (WSMP) [17] for sparse problems and SLATE [27] for dense problems) architectures.

To demonstrate its performance on GPU architectures, we will examine the application of our SISS method to both sparse and dense matrices of varying size, spanning problems for which SISUBIT may be accommodated by a single device and those which require SISUBIT to be distributed across several GPUs or computational nodes. The performance of this method will be compared to various state of the art shared memory and distributed memory dense eigensolvers. It will be demonstrated that for dense problems in which SISUBIT may be accommodated by a single GPU, the proposed GPU implementation of SISS outperforms the fastest dense eigensolver by a factor of 9. The case when SISUBIT must be distributed is slightly more complicated, however it will be demonstrated that there is a promising avenue for future improvement, especially in the context of sparse GPU solvers.

We should note that our comparison excludes a large class of iterative eigensolvers such as the implicitly restarted Lanczos [31], locally optimal block preconditioned conjugate gradient (LOBPCG) [11, 24], Jacobi-Davidson [41], and polynomial filtering [32] methods. The reason for this omittance is that the performance of these algorithms largely depends on the implementation of efficient (often sparse) matrix-vector products and the number of eigenpairs desired. This is often an application dependent issue that is difficult to be generalized. Furthermore, when the number of eigenpairs to be computed is relatively large, the performance of these algorithms often depends on how efficient the projected dense EVP is solved.

This paper is organized as follows. In Sec. 2, we give a brief outline of the major steps of the SISS algorithm which are pertinent to the discussion of its performance characteristics. In Sec. 3, we examine our implementation strategy of SISS on hybrid CPU/GPU computing architectures. The performance of these implementations will be compared to analogous CPU implementations of SISS and dense eigensolvers available in cuSOLVER, ELPA and vendor optimized implementations of LAPACK and ScaLAPACK in Sec. 4. Some discussions on the efficacy of the presented GPU implementations and Sec. avenues for further performance improvements are provided in 5.

2 ALGORITHM

Consider the partial eigendecomposition of a system matrix A,

$$AX = X\Lambda,$$

where $A \in \mathbb{R}^{N\times N}$ is symmetric, $X \in \mathbb{R}^{N\times M}$ is the orthogonal matrix of desired eigenvectors, $\Lambda \in \mathbb{R}^{M\times M}$ is the diagonal matrix of desired eigenvalues, with $M \leq N$. The general idea of SISS is to divide the spectrum into several slices, each of which contains a small subset of eigenvalues as shown in Figure 1. Two adjacent slices, e.g. j and $j+1$, are separated by a shift σ_j chosen by some spectrum partition scheme [21, 32, 44, 47]. A subspace iteration is applied to the shift-invert transformed matrix, $(A - \sigma_j I)^{-1}$ to compute eigenvalues near σ_j and their corresponding eigenvectors. Note that a subset of eigenvalues in both slice j and slice $j+1$ are computed in this subspace iteration. A validation step is performed upon the completion of the subspace iteration for each σ_j to decide which eigen value and eigen vector approximations to keep.

Because each subspace iteration associated with each shift σ_j is independent from those associated with other shifts, the SISS algorithm can be easily distributed and parallelized among different sets of computational resources. Alg. 1 outlines the major steps of a distributed parallel SISS algorithm that assumes a set of shifts that partition the spectrum into n_s+1 slices, $\{\sigma_j\}_{j=1}^{n_s}$, has been given. The algorithm distributes the computational workload into n_s separate tasks (one for each σ_j) mapped to some subsets of the available computational resources. We will refer to this resource subset as an *execution context* in the following. Each independent task obtains a set of approximate eigenpairs in the spectral neighborhood of its associated σ_j by constructing a basis of dimension K which spans the local eigenspace. In practice, n_s is chosen so that the number of eigenvalues near each shift, which is approximately M/n_s is relatively small. We have found that setting $K \approx 10\lceil M/n_s\rceil$ is often sufficient to ensure rapid convergence in most cases [44]. Once all tasks have completed, there is a *single* synchronization step which allows for the desired eigenpairs to be extracted.

Once this validation has been completed, the desired eigenpairs are irregularly distributed among the compute ranks due to the fact that there is no guarantee that all spectrum slices contain the *exact* same number of eigenvalues. As such, one may optionally gather or replicate the combined set of desired eigenpairs using `MPI_(All)gatherv`. However, we note that this is not always a necessary synchronization step, as it is often the case that one may extract the required information of the eigenvectors in this irregular format to avoid the communication of the eigenvectors directly.

There are four primary computational kernels in Alg. 1. The first kernel (Line 1.4) performs a triangular factorization (TRF) of the shifted system matrix into either LU or LDLT triangular factors. For dense system matrices, this factorization scales $O(N^3)$ with a much lower prefactor than direct dense eigensolvers. For sparse problems, it is possible to achieve $O(N^2)$ or even $O(N)$ scaling depending on the sparsity pattern [14]. The second kernel (Line 1.5) solves a set of linear systems using the triangular factors (TRS) produced by TRF. This process formally scales $O(KN^2)$ for dense system matrices and possibly as low as $O(NK)$ for sparse matrices. Line 1.6 produces an orthonormal subspace from a non-orthogonal input. Several algorithmic choices exist for the production of this subspace, however in this work we will employ the Cholesky QR (CholQR) (Alg. 2) due to its low communication overhead and leverage of level-3 BLAS primitives. CholQR scales as $O(NK^2)$. Line **??** are collectively known as the shift-invert subspace iteration (SISUBIT),

Algorithm 1: Parallel Shift-Invert Spectrum Slicing

```
Input  : Symmetric $A \in \mathbb{R}^{N\times N}$, shift partition $\{\sigma_j\}_{j=1}^{n_s}$,
         number of desired eigenpairs $M$, basis dimension
         $K$, and max iteration $n_{iter}$.
Output : Eigenvectors $X \in \mathbb{R}^{N\times M}$, and eigenvalues
         $\Lambda \in \mathbb{R}^{M\times M}$.
1.1 Distribute work over $j$.
1.2 for $j$ assigned to this execution context do
1.3     Form initial guess $V_j \in \mathbb{R}^{N\times K}$
1.4     Factorize $(A - \sigma_j I)$                         (TRF)
        for $i = 1 : n_{iter}$ do
1.5         $V_j \leftarrow (A - \sigma_j I)^{-1} V_j$          (TRS)
1.6         $V_j \leftarrow \mathrm{orth}(V_j)$                 (CholQR)
        end
1.7     $(X_j, \Lambda_j, \vec{r}_i) \leftarrow \mathrm{RayleighRitz}(A, V_j)$   (RR)
    end
1.8 $(X, \Lambda) \leftarrow \mathrm{DistValidate}(\{(X_j, \Lambda_j, \vec{r}_j)\})$
```

Algorithm 2: Cholesky QR Algorithm (CholQR)

```
Input  : $V \in \mathbb{R}^{N\times K}$
Output : $Q \in \mathbb{R}^{N\times K}$ such that $Q^T Q = I$.
2.1 $Y \leftarrow V^T V$
2.2 $L \leftarrow Y = LL^T$ (Cholesky)
2.3 $Q \leftarrow VL^{-T}$
```

Algorithm 3: Rayleigh-Ritz Procedure (RR)

```
Input  : Symmetric $A \in \mathbb{R}^{N\times N}$, $V \in \mathbb{R}^{N\times K}$
Output : Approximate eigenvalues $\Lambda \in \mathbb{R}^{K\times K}$, eigenvectors
         $X \in \mathbb{R}^{N\times K}$ such that $X^T X = I$, and residual norms
         $\vec{r} \in \mathbb{R}^K$
3.1 $W \leftarrow V^T A V$
3.2 Solve $WC = C\Lambda$ (EVP)
3.3 $X \leftarrow VC$
3.4 $R \leftarrow AX - X\Lambda$
    for $k = 1 : K$ do
3.5     $\vec{r}_k = ||R(:,k)||$
    end
```

and must be performed n_{iter} times for each shift. Note that we will refer to TRF and the n_{iter} invocations of SISUBIT collectively as "SISUBIT" in following for brevity. Once the basis has been produced, Line 1.7 constructs a set of eigenpair approximations in the spectral neighborhood of each shift and their associated residual norms through the Rayleigh-Ritz (RR) procedure (Alg. 3). The complexity of RR is $O(KN^2)$ for dense system matrices and can be as low as $O(NK)$ for sparse matrices. However, due to the task dependency pattern of matrix-matrix multiplication vs TRS, RR admits a much lower prefactor than TRS despite admitting the same formal scaling (as will also be demonstrated in Sec. 4). In the following sections, we examine the implementation and performance of these kernels, as well as their integration into parallel SISS.

3 IMPLEMENTATION

In this section, we examine implementation strategies for SISUBIT and parallel SISS for distributed computing environments with GPU accelerators. The variations of parallel SISS implementations primarily differ in their choice for how to select $\{\sigma_j\}$, how to distribute the work (Line 1.1) and how to validate and combine the distributed eigenpair results (Line 1.8). We will not explicitly treat these aspects in this work as they are highly problem dependent. Instead, we will focus on the common aspects of all SISS implementations which are pertinent to performance, namely the concurrent execution of SISUBIT and synchronization of spectral slice metadata to allow for extraction of the desired eigenpairs.

In the following, we will define an execution context to be a heterogeneous compute platform consisting of some number of CPU and GPUs. The total set of available computational resources will be defined as n_p of these execution contexts, and each context will be assumed to be identical. Further, we will make the following assumptions to simplify the remainder of the discussion:

- n_{iter} is a constant and the same for each shift.
- the available computing resources allow for a balanced distribution of n_s computational tasks, i.e. $n_s \geq n_p, n_s \bmod n_p = 0$, and each execution context will perform SISUBIT for n_s/n_p shifts.
- n_s may be chosen to allow for a shift distribution which permits rapid convergence of the desired eigenpairs and such that the memory available to each execution context is able to store at least $O(NM/n_s)$ data in addition to the $O(N^2)$ storage of the system matrix for dense problems and $O(N)$ for sparse problems.
- $\{\sigma_j\}$ has been chosen as to have each of the spectral slices contain *roughly* the same number of eigenvalues (i.e. such as the *a priori* spectral density estimation scheme from [34, 45]). We note for clarity that these estimation schemes may be replicated and require little to no communication, thus not affecting the overall scalability of the proposed method.
- and the validation / combination scheme only requires the communication of $O(M)$ data, e.g. eigenvalue approximations, residual norms, etc. This is generally possible if full factorizations of the shifted system matrices are available through comparison of the inertial counts of adjacent shifts [7, 16, 44] (see Fig. 2).

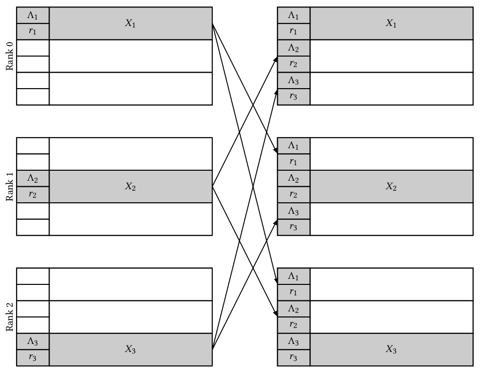

Figure 2: Example parallel SISS synchronization of slice metadata for 3 shifts distributed across 3 execution contexts. The boxes on the left (right) represent the status of the slice metadata pre-(post-) synchronization, and the arrows indicate the communication of data. The size of each box represents its relative memory footprint. Note that only the small boxes are communicated.

Due to the fact that, by design, SISS places shifts near clusters of eigenvalues, the condition numbers of the resulting shifted linear systems are often quite large ($O(10^8)$). As a result, standard mixed precision schemes for linear system solution [6] are often too numerically unstable for use with SISS. Thus, for this work, we will limit the discussion to purely double precision arithmetic.

3.1 SISUBIT Implementation

One of the hallmarks of parallel SISS is its ability to quickly resolve the desired eigenpairs for EVPs that are too large to be solved efficiently with traditional eigensolvers. As such, it will often be the case that the memory requirements for SISUBIT cannot be accommodated by a single compute node. Thus, we consider two regimes of concurrency for SISUBIT: the regime where the memory requirement of SISUBIT is below the capacity of the shared memory accessible to a single compute node (i.e. no network communication is incurred within its execution context), and the regime where matrices and vectors must be stored in a distributed fashion amongst several compute nodes. We will refer to these regimes as SM-SISUBIT (shared memory) and DM-SISUBIT (distributed memory), respectively. We note for clarity that shared memory is meant to indicate memory which is directly accessible from *all* of the resources of an execution context, thus for GPU implementations of SM-SISUBIT, this means that these matrices may be stored in the device memory of a *single* GPU rather than distributed amongst the possibly several devices available to a single compute node. Implementations of SISUBIT which utilize the distributed storage of the system matrices across several GPUs will be considered DM-SISUBIT, even if those GPUs are associated with the same compute note. In the following, we discuss the CPU and GPU implementations of SM- and DM-SISUBIT in terms of state of the art numerical linear algebra software.

Dense SM-SISUBIT Implementation. For dense problems which are able to reside in the memory of a single compute node, the SM-SISUBIT kernels may be composed of standard linear algebra primitives provided by vendor optimized BLAS and LAPACK libraries.

Due to its prevalence in contemporary HPC and the maturity of its dense linear algebra software stack, we will limit our discussion to NVIDIA GPUs and thus the cuBLAS and cuSOLVER libraries for accelerated, single-device implementations of BLAS and LAPACK primitives, respectively. For the GPU implementation of dense SM-SISUBIT, we will only consider the case where the memory requirement of a *single* SISUBIT may be accommodated in the device memory of a single GPU, i.e. we do not consider batched BLAS/LAPACK implementations for the concurrent execution of multiple instances of SISUBIT on a single device. For problems small enough in which batch implementations could be utilized, the GPU direct dense eigensolver implemented in cuSOLVER (`cusolverDnDsyevd`) would likely be more performant than SISS.

While cuSOLVER does implement a LDLT factorization, as of this work it does not offer an analogous backsolve for this factorization. Thus, for TRF and TRS we will utilize `cusolverDnDgetrf` (LU) and `cusolverDnDgetrs` (LU solve) from the cuSOLVER library for the GPU implementation of dense SM-SISUBIT, respectively. Analogously, we will utilize the vendor implementations of LAPACK `DGETRF` and `DGETRS` for the respective CPU implementations.

For shared memory CholQR, the inner product (Line 2.1) has been implemented using `cublasDgemm` and `DGEMM` for the CPU and GPU implementations, respectively. For Line 2.2 (Line 2.3), the GPU and CPU implementations have utilized `cusolverDnDpotrf` (`cublasDtrsm`) and `DPOTRF` (`DTRSM`), respectively. Similarly, the application of the system matrix to the basis as required by RR (Line 3.1) has been implemented by `cublasDgemm` and `DGEMM` for the shared memory GPU and CPU implementations, respectively. For Line 3.2 (Line 3.3), the shared memory CPU and CPU implementations have utilized `cusolverDnDsyevd` (`cublasDgemm`) and `DSYEVD` (`DGEMM`), respectively. We note that the residual calculation in Line 3.4 may be efficiently implemented by `cublasDdgmm` on the GPU with no analogy in the standard LAPACK API.

As the memory requirement for SM-SISUBIT is assumed to fit in the device memory of a single GPU, we are able to minimize data movement between host and device by enforcing data locality on the device throughout the computation. As TRF and TRS are performed in place, there are a maximum of three data transfers that would need to occur in the GPU implementation of dense SM-SISUBIT: a single host-to-device (H2D) transfer of $(A - \sigma_j I)$ and V_j prior to TRF, a H2D transfer of the A (overwriting the triangular factors produced on the GPU) before RR, and a device-to-host (D2H) transfer of $(X_j, V_j, \vec{r}_j)$ after RR. If the device memory capacity is such that $O(2N^2)$ data may be accommodated (in addition to the basis), the second H2D transfer of A may be avoided by copying A instead of $(A - \sigma_j I)$ in the first step, performing a device-to-device (D2D) copy of A locally in the high-bandwidth memory (fast), and performing the identity shift in place on the copy of A which may be overwritten by the triangular factors in TRF. The first H2D transfer of A and V_j may be avoided completely if they are able to be generated directly on the device.

Dense DM-SISUBIT Implementation. Much like the dense implementation of SM-SISUBIT, we may compose the performance critical kernels of dense DM-SISUBIT using analogous implementations of distributed memory linear algebra primitives. However, in contrast to SM-SISUBIT, there does not exist a purely GPU distributed memory linear algebra library as of this work. Currently, the state-of-the-art for GPU accelerated distributed memory dense linear algebra is the SLATE library [13]. As a hybrid GPU/CPU library, SLATE utilizes both vendor optimized CPU and GPU accelerated implementations of BLAS/LAPACK primitives to achieve its performance. For the CPU implementations of DM-SISUBIT, we will utilize the ScaLAPACK library as it currently outperforms the CPU-only implementation of performance critical primitives (TRF/TRS) in SLATE [26, 27].

In analogy with SM-SISUBIT, Line **??** have been implemented using `slate::gemm` and `PDGEMM`, and Line 2.2 has been implemented `slate::potrf` and `PDPOTRF` for SLATE and ScaLAPACK implementations, respectively. As SLATE does not currently implement an eigensolver capable of returning eigenvectors, both ScaLAPACK and SLATE implementations of DM-SISUBIT utilize `PDSYEVD` for Line 3.2, with the latter performing a conversion to ScaLAPACK format prior to invocation. The SLATE (ScaLAPACK) implementations of distributed memory TRF and TRS in DM-SISUBIT have been implemented by `slate::getrf` (`PDGETRF`) and `slate::getrs` (`PDGETRS`), respectively.

Due to the fact that SLATE is a hybrid CPU/GPU distributed memory library, ensuring data locality on the device throughout the computation as we have proposed in SM-SISUBIT is generally not possible. Instead, SLATE offers a mechanism which pins the "origin" of its matrices to either the host or device such that the specified origin locality is returned post function invocation (although it may or may not be respected internally). For SLATE implementations of DM-SISUBIT, we have pinned the triangular factors and basis to the device to incur minimal data movement throughout the SISUBIT invocation. We examine the magnitude of the data movement costs incurred by SLATE in Sec. 4

Sparse SISUBIT. For sparse TRF and TRS, we must utilize sparse direct linear system solvers such as PARDISO [9, 25, 43] for shared memory and SuperLU_DIST [15, 33, 35] for distributed memory implementations, respectively. However, the status of GPU accelerated sparse linear algebra is much less mature than its dense counterpart. Vendor implementations of sparse BLAS primitives have been implemented in libraries such as NVIDIA cuSPARSE for NVIDIA GPUs, but GPU implementations for operations for TRF and TRS are few and far between. There are several publicly available implementations of GPU sparse direct linear solvers, including SuperLU_DIST, PaStiX, ans WSMP. In this work, we limit our experiments with sparse solvers to only include SuperLU_DIST. The use of other GPU sparse solvers will be explored in future work. In SuperLU_DIST, GPU acceleration is limited to its implementation of sparse TRF: `pdgstrf`. At this time, SuperLU_DIST does not provide a GPU accelerated implementation of sparse TRS. The GPU acceleration of `pdgstrf` in SuperLU_DIST is an offloaded computation, i.e. the affinity of the data both before and after execution must be the host, and the data movement and device computation are performed internally. However, because the SuperLU_DIST implementation of TRS (`pdgstrs`) is CPU-only as of this time, data movement is only required before and after the TRF step and all subsequent computation (TRS, CholQR and RR) is performed on the host to preserve data locality despite the fact that CholQR and RR may be more efficiently implemented on the GPU. As SuperLU_DIST is a distributed memory solver, the only distinction between sparse implementations of TRF or TRS in SM- and DM-SISUBIT is in the scope of its communication context. To have a more complete comparison with the current status of CPU implementations of sparse-direct linear system solvers, we also consider the implementation of SM-SISUBIT with PARDISO implementations of TRF and TRS (both provided by the `pardiso` driver with differing input parameters). We note that while SuperLU_DIST performs a sparse LU factorization, PARDISO is a symmetric solver which returns a sparse LDLT factorization.

For sparse SISUBIT, only the system matrices and triangular factors are stored as sparse matrices: the basis constructed by SISUBIT is still generally a dense matrix. As such, we may use the same CholQR kernels as the dense implementations of SM- and DM-SISUBIT described above. With the exception of the application of the system matrix to the basis in Line 3.1, which would be implemented using sparse matrix-vector products (SpMV) in the case of sparse matrices, the remainder of RR is also identical to the dense implementations described previously. Due to the fact SpMV is highly problem dependent and that RR will be performed on the host for sparse implementations of sparse SISUBIT in this work, we do not consider its implementation at this time. However, because RR comprises only a small percentage of the overall execution time of SISS, it will be demonstrated that this omittance does not yield any measurable impact on the results of this work.

3.2 Parallel SISS Implementation

Given an implementation of either SM- or DM-SISUBIT, parallel SISS is agnostic as to whether the computation was performed on the CPU or the GPU or whether the basis was produced using dense or sparse linear algebra. As has been demonstrated in the work of [44], it is possible to validate and select eigenpair approximations in parallel SISS through communication of only $O(M)$ data, e.g. the eigenvalue approximations and residual norms. As such, we will characterize the implementation of parallel SISS as concurrent execution of n_s SISUBIT invocations on n_p independent computing resources followed by a collective gather of the eigenvalue approximations and residual norms computed for each shift (see Fig. 2). This collective gather will be implemented by `MPI_Allgather` for parallel SISS based on either SM- and DM-SISUBIT. As such, its performance is only based on n_p, the interconnect of the computing cluster and K, and is independent of the problem dimension N. We note that in the case of SM-SISUBIT the `MPI_Allgather` is a global collective of all n_p MPI ranks being used for the SISS calculation, whereas for DM-SISUBIT, this collective may be limited to include only the n_p ranks which correspond to the same processor coordinate in the remote execution contexts. As the synchronization of slice data is taken to be the sole synchronization point between independent execution contexts, the weak scaling of SISS in this paradigm is thus limited only by the weak scaling of `MPI_Allgather` on arrays of length K. We note that in the case of DM-SISUBIT one must also consider the scaling of the distributed linear solver, however, in this work, the size of the execution context for each individual SISUBIT will be taken to be a constant, and achieving optimal performance is very dependent on the solver and the compute platform itself. Further, given the assumption that the SISS calculation may be conducted using enough resources as to perform a load balanced calculation, the scaling of SISS is independent on M in this context.

4 NUMERICAL EXPERIMENTS

In this section, we perform a number of numerical experiments to compare the performance of the CPU and GPU implementations of parallel SISS based on the various schemes for SISUBIT discussed in the previous section. In turn, we will examine the CPU and GPU implementations of each of the various compute intensive kernels, as well as the combined implementations of SISUBIT and SISS, on a representative set of modern computing architectures. Experiments will be performed using the resources of the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) and the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC).

Each Summit node consists of a two IBM POWER9 processors and 6 V100s interconnected via NVLink (50 GB/s bidirectional peak bandwidth) with 512 GB DDR4 memory. Each V100 has 16 GB of high-bandwidth memory, leading to a total of 96 GB device memory on each node. Each POWER9 processor consists of 21 cores (2x21 / node @ 3.8 GHz) with a maximum 4 hardware threads (HWT) per core. From the Cori supercomputer, experiments will be performed using two sets of CPU architectures: Intel Xeon Phi Knight's Landing (KNL) and Intel Xeon Gold 6148 (XG). Each KNL node consists of 68 cores (@ 1.4 GHz) with a maximum 4 HWT per core, and 96 GB DDR4 memory and 16 GB high-bandwidth MCDRAM. Each XG processor consists of 20 cores (2x20 / node @ 2.40 GHz) with a maximum of 2 HWT per core.

As was discussed in Sec. 3, the performance of SISUBIT is heavily reliant on the existence of vendor optimized BLAS/LAPACK libraries for the target architectures. For single GPU dense linear algebra, we have used the cuBLAS and cuSOLVER libraries provided by NVIDIA (CUDA version 10.1.168). For POWER9 CPUs, we have used the BLAS implementation featured in the Engineering Scientific Software Library (ESSL version 6.1.0) provided by IBM. As ESSL does not feature a full LAPACK implementation, the missing functionality has been resolved by the reference LAPACK implementation provided by NETLIB. For Intel CPUs, we have utilized the BLAS and LAPACK implementations provided by the Intel Math Kernel Library (MKL version 19.0.1). The SLATE and ScaLAPACK (version 2.0.2) libraries used for distributed memory BLAS/LAPACK have been linked to the vendor BLAS/LAPACK libraries of their respective architectures for their single node performance. Unless otherwise noted, SLATE will utilize a 7-to-1 MPI-to-GPU affinity in its implementations of DM-SISUBIT. For sparse TRF and TRS, we have used SuperLU_DIST (version 6.3.0) and PARDISO (as implemented in Intel MKL 19.0.1).

As all of the considered CPU architectures feature multiple HWT per physical core, there is some variability in BLAS/LAPACK performance from differing ratios of HWT. All calculations using POWER9 and KNL CPUs will be performed using 2 HWT / core, and those using on XG will be performed using 1 HWT / core. Calculations performed on POWER9 CPUs will utilize 42 cores, KNL CPUs will utilize 64 cores (4 reserved for OS processes), and XG will utilize 40 cores, unless otherwise noted.

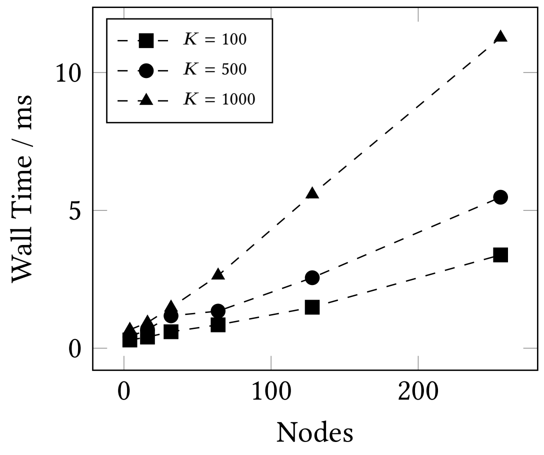

Figure 3: Scaling of slice metadata (eigenvalue approximations and residual norms) synchronization for various values of K on the Summit supercomputer. Calculations were performed with 6 MPI ranks per Summit node.

For experiments involving dense SISUBIT, test matrices were randomly generated according to the normal distribution $\mathcal{N}(0., 1.)$, and the system matrices were diagonally shifted by a constant factor proportional to their order. For sparse experiments, we have used the Ga10As10H30 test matrix ($N = 113,081$, $NNZ = 6,115,633$) from the SuiteSparse Matrix Collection [8]. The Ga10As10H30 matrix is a discretized Kohn-Sham Hamiltonian generated by PARSEC: a finite-element electronic structure software based on density functional theory.

In the following, we compare the performance of the proposed SISS implementation with dense eigensolvers in the cuSOLVER, LAPACK, ScaLAPACK and ELPA libraries. All dense eigensolver results were obtained on the Summit supercomputer and use a blocking factor of $NB = 128$. All distributed eigensolver times reported represent the minimum time to solution at the extent of the strong scaling of these methods.

To compare the performance of SISS implementations with dense eigensolvers, we examine the case when each execution context performs SISUBIT for a single shift, i.e. $n_p = n_s$. Figure 3 shows the scaling of slice synchronization (per the discussion in Sec. 2) for various values of K on the Summit supercomputer. Even for large K and large processor counts, the amount of time spent in synchronization is under 1 second. As this is negligible to the overall computation, it will be approximated in the following by adding 1 second to the execution time of a single execution of SM- or DM-SISUBIT. We will refer to this scheme as the proxy application for SISS in the following.

4.1 Dense SM-SISUBIT + SISS

For shared-memory execution contexts, we have compared V100, POWER9, KNL and XG implementations of dense SM-SISUBIT and their integration into parallel SISS per the proxy application discussed in the previous section. A summary of these results is given in Tab. 1. For TRF and TRS, there was considerable variability in the performance of the CPU implementations between the different architectures for different problem sizes. However, the GPU implementation was demonstrated to be consistently more performant. Similarly, the GPU implementations of CholQR and RR drastically outperformed their CPU counterparts, however their low percentage of total SISUBIT compute time yields these speedups to have a smaller impact on overall time-to-solution.

Figure 4 shows performance comparisons for dense SM-SISUBIT as a function of problem dimension and Fig. 5 is a breakdown of the total SISUBIT computation time into its dominant components.

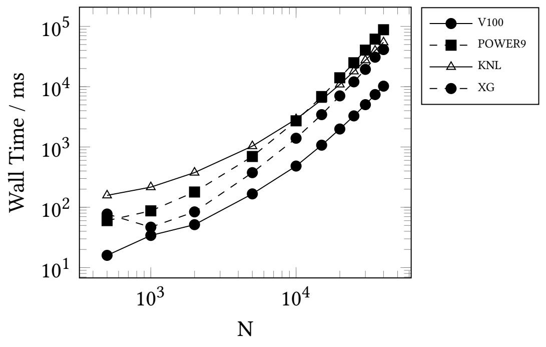

Figure 4: Wall time comparison for CPU and GPU implementations of dense SM-SISUBIT as a function of problem dimension (N). Times are given in milliseconds and include data transfers between host and device. All calculations use $K = 100$.

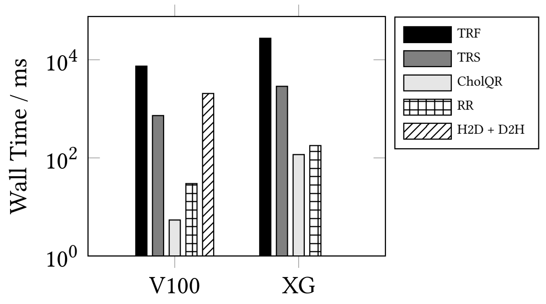

Figure 5: Wall time comparison for dominant costs in the CPU and GPU implementations of dense SM-SISUBIT. Times are given for $N = 40,000$, $K = 100$, and $n_{iter} = 4$. Data transfer times are given for the GPU implementation and represent total times (H2D + D2H).

Table 1: Speedups for GPU implementations of compute intensive kernels for dense SM-SISUBIT. Speedups are relative to the fastest CPU implementation from the considered architectures (in parentheses) and are calculated as t_{cpu}/t_{gpu}. Except for SISUBIT, GPU times do not include host-device transfer times.

Kernel	Speedup	
	$N \leq 1,000$	$N \geq 10,000$
TRF	6x (XG)	3x (KNL)
TRS	1.5x (POWER9)	4-5x (XG)
CholQR	50x (POWER9)	20x (POWER9)
RR	1.5-2x (XG)	6x (XG)
SISUBIT	1.5-2x (XG)	4x (XG)

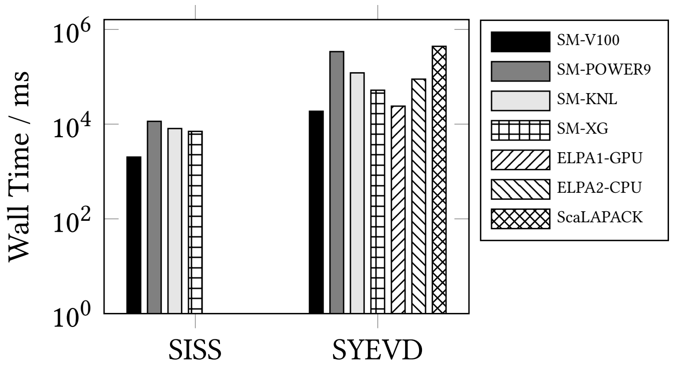

Figure 6: Wall time comparison for a representative SM-SISS calculation with state-of-the-art shared and distributed memory dense direct eigensolvers (SYEVD) for $N = 18,000$. Times for SISS are given by SISUBIT + synchronization with $K = 100$ and $n_{iter} = 4$.

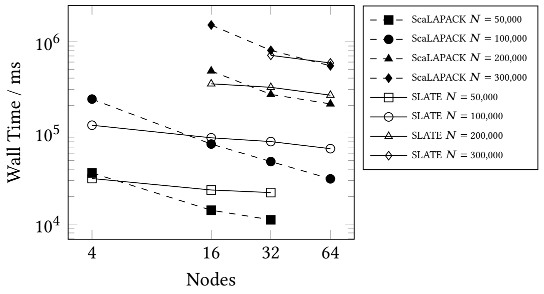

Figure 7: Strong scaling comparison of ScaLAPACK and SLATE implementations of DM-SISUBIT with $K = 100$ and $n_{iter} = 4$.

Component comparison was made only for XG as it was demonstrated to be the fastest CPU implementation of SM-SISUBIT of the considered architectures for all problem dimensions. Figure 6 compares the performance of the proxy SISS calculation with various shared memory and distributed memory dense eigensolvers for $N = 18,000$. The extent of strong scaling for both ScaLAPACK and ELPA was found to be a 6x6 process grid on a single Summit node

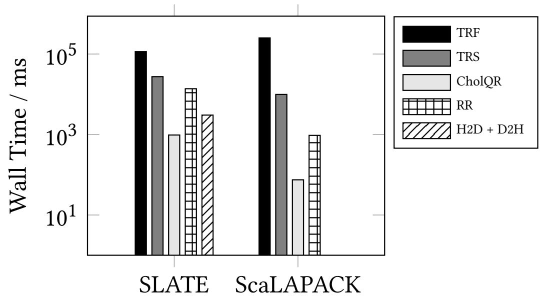

Figure 8: Wall time comparison for dominant costs in the CPU (ScaLAPACK) and GPU (SLATE) implementations of dense DM-SISUBIT. Times are given for $N = 100,000$, $K = 100$, and $n_{iter} = 4$. The H2D+D2H time for the SLATE implementation is the aggregate for the entire SISUBIT on the first MPI rank.

Table 2: Speedups for the SLATE implementation of compute intensive kernels for dense DM-SISUBIT over ScaLAPACK. Values are calculated as $t_{scalapack}/t_{slate}$. Values less than 1.0 represent a performance degradation.

Kernel	Speedup			
	$N = 100,000$		$N = 300,000$	
	4 nodes	64 nodes	32 nodes	64 nodes
TRF	2.1x	0.8x	1.7x	1.8x
TRS	0.4x	0.2x	0.1x	0.1x
CholQR	0.07x	0.04x	0.07x	0.04x
RR	0.07x	0.02x	0.02x	0.01x
SISUBIT	2.3x	0.5x	1.1x	0.9x

for this problem dimension. ELPA eigensolver times were obtained for the lowest 9,000 eigenpairs and the GPU eigensolver therein used a 6-to-1 MPI-to-GPU affinity. The GPU implementation of dense SM-SISUBIT outperformed the shared-memory eigensolver in cuSOLVER by a factor of 9x, the fastest shared-memory CPU eigensolver (XG) by a factor of 25x, the fastest distributed-memory CPU eigensolver (ELPA) by a factor of 44x and the distributed-memory GPU eigensolver in ELPA by a factor of 11x.

4.2 Dense DM-SISUBIT + SISS

For distributed memory execution contexts, we have compared SLATE and ScaLAPACK implementations of DM-SISUBIT and their integration into parallel SISS per the proxy application discussed in the previous section. In particular, we have examined the strong scaling of the individual DM-SISUBIT kernels for several problem dimensions across 4, 16, 32 and 64 Summit nodes corresponding to 12x14, 24x28, 23,42 and 48x56 process grids, respectively, for both ScaLAPACK and SLATE implementations. NB was chosen to be 128 for ScaLAPACK calculations and 512 for SLATE calculations. A summary of these results is given in Tab. 2

TRF using SLATE outperforms the ScaLAPACK implementation at low processor counts for all considered problem dimensions, but

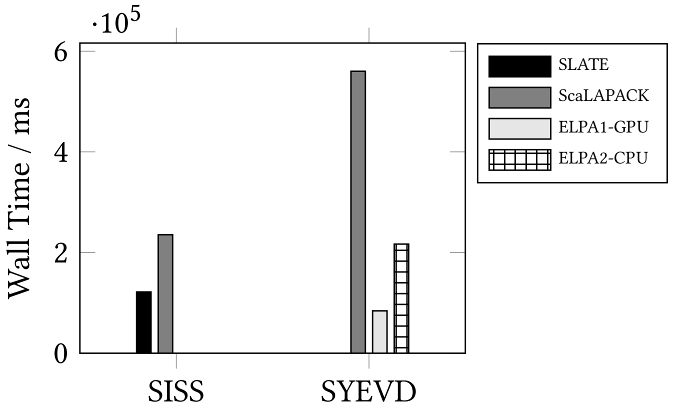

Figure 9: Wall time comparison for a representative DM-SISS calculation with state-of-the-art distributed memory dense direct eigensolvers (SYEVD) for $N = 100,000$. Times for SISS are given by SISUBIT + synchronization with $K = 100$ and $n_{iter} = 4$.

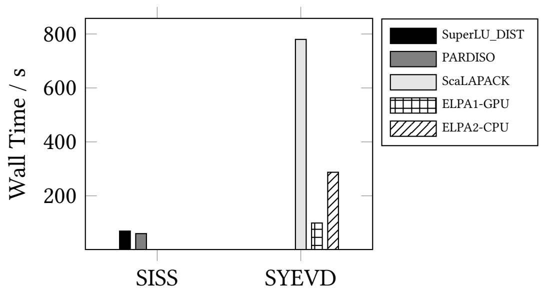

Figure 10: Wall time comparison between a sparse SISUBIT calculation and distributed memory direct eigensolvers (SYEVD) for Ga10As10H30. Times for SISUBIT are given for $K = 100$ and $n_{iter} = 4$ and do not include CholQR or RR.

ScaLAPACK exhibits better strong scaling and is able to outperform SLATE at larger processor counts for most problem dimensions. As of this work, the SLATE implementation of TRS is drastically outperformed by ScaLAPACK across all problem dimensions and processor counts. The case is more drastic for CholQR and RR. We note for clarity that the node counts for the timings of CholQR and RR in Tab. 2 are representative of the larger SISUBIT invocation, i.e. CholQR and RR are only distributed across a subset of those processors based on the blocking factors. Despite the ill-performant implementations of TRS, CholQR and RR, the SLATE implementation of DM-SISUBIT is able to outperform the ScaLAPACK implementation for low processor counts as is illustrated in Fig. 7. This is due to the fact that these kernels are not the dominant cost of DM-SISUBIT. For a breakdown of the total DM-SISUBIT calculation in this regime into its component kernels, see Fig. 8.

Figure 9 compares the performance of the SLATE and ScaLAPACK implementations of the proxy SISS calculation with various distributed memory dense eigensolvers for $N = 100,000$. The SLATE and ScaLAPACK SISS calculations were performed on 4 nodes with the previously described processor grids, and the eigensolver calculations were performed on a 32x36 process grid across 32 Summit nodes. This was found to be the extent of the scaling for these eigensolvers for the minimum time to solution. ELPA times represent obtaining the lowest 20,000 eigenpairs and the GPU eigensolver therein used a 6-to-1 MPI-to-GPU affinity. The SLATE implementation of DM-SISUBIT outperforms the fastest distributed memory CPU eigensolver (ELPA) by a factor of 1.7x, but is outperformed by the distributed memory GPU eigensolver in ELPA by a factor of 1.4x.

4.3 Sparse SISUBIT + SISS

As was discussed in Sec. 3, the current state of GPU implementations of sparse kernels for SISUBIT is rather limited as of this work. However, it is instructive to examine how these implementations compare to the current state of dense eigensolvers. Figure 11 shows the strong scaling of CPU and GPU sparse TRF for the Ga10As10H30 test matrix using SuperLU_DIST and compares these to the analogous, symmetric operations in PARDISO. We note that the SuperLU_DIST calculations were performed on the Summit supercomputer while the PARDISO calculation was performed on

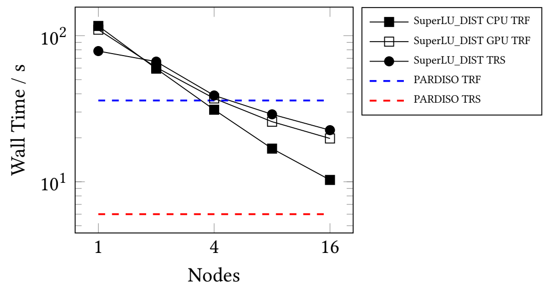

Figure 11: Wall time comparison of shared and distributed memory implementations of sparse TRF and TRS for Ga10As10H30. TRS times are given for $K = 100$.

KNL. Much like the dense implementations of distributed memory TRF, the GPU offloaded implementation of sparse TRF outperforms the CPU implementation with low processor counts (1.1x). The CPU implementation exhibits better strong scaling and is able to outperform the GPU implementation for large processor counts. Further, the distributed SuperLU_DIST TRF is able to obtain a lower time-solution over PARDISO, but the symmetric TRS implemented by PARDISO is far more performant than the analogous implementation in SuperLU_DIST.

Figure 10 compares the performance of SuperLU_DIST and PARDISO implementations of the proxy SISS calculation with various distributed memory dense eigensolvers. CholQR and RR times are not included in the SISUBIT times, but per the previous results in the dense implementation of SISUBIT, these contributions are negligible. Dense eigensolver calculations were performed with a 32x26 processor grid. We note that the SuperLU_DIST times are for the distributed CPU implementation at the extent of its strong scaling (32x42 processor grid on 32 Summit nodes) and the PARDISO times are for SM-SISUBIT as we have not considered the PARDISO cluster interface in this work. The SuperLU_DIST (PARDISO) implementations of SISUBIT were about to outperform the fastest (ELPA) distributed CPU eigensolver by a factor of 4.1x (4.8x) and the distributed GPU eigensolver in ELPA by a factor of 1.4x (1.6x).

5 CONCLUSIONS

In this work, we have presented several implementations strategies for parallel SISS for distributed architectures with GPU accelerators. As was discussed in the introduction, the primary factor to consider in the implementation of GPU-based linear algebra algorithms is optimally utilizing the low capacity device memory to avoid costly transfers between host and device. For small to medium sized dense problems ($N < 40,000$), the memory requirement of SISUBIT may be accommodated by a single GPU. To demonstrate the efficacy of the proposed GPU implementation of SISS, we have compared shared memory CPU and GPU implementations of SISUBIT on a representative set of modern CPU architectures. We have demonstrated that we may obtain very high-performance implementations of SISS by offloading the performance critical kernels to the GPU via vendor optimized dense linear algebra libraries such a cuBLAS and cuSOLVER from NVIDIA and ensuring data locality to prevent costly data transfers between host and device. We have demonstrated that it is possible to achieve performance improvements upwards of 9x over state of the art shared and distributed memory eigensolver with this scheme. We note for posterity that the data transfer times are non-negligible in this scheme ((Fig. 5), thus these performance improvements would be further enhanced for problems in which it could be eliminated (per the discussion in Sec. 2).

For large dense problems which must be distributed, the data movement incurred by current distributed GPU linear algebra libraries such as SLATE yield a large impact on the performance of SISUBIT. For relatively low processor counts, SLATE is able to outperform analogous ScaLAPACK implementations of compute intensive kernels. However, we have demonstrated that the ScaLAPACK implementations of these kernels exhibit better strong scaling than the SLATE implementations, leading to ScaLAPACK outperforming SLATE at large processor counts. Further, we have demonstrated that the implementations of TRS, CholQR and RR are currently ill-suited for SLATE implementation. Although not expressly considered in this work, the conversion utilities between SLATE and ScaLAPACK storage formats offered by the SLATE library would allow for utilization of the ScaLAPACK implementations of these ill-performance kernels, thus only utilizing SLATE for the accelerated implementation of TRF. As TRF is the dominant cost in SISUBIT, this would likely offer a more performant implementation scheme than the one presented in this work. However, this performance improvement would likely be minimal due to the fact that these kernels are not dominant in cost.

Due to space and time limitation, we have not performed comparisons with spectral divide-and-conquer methods based on QDWH and Zolotov (ZOLO) approximation [38] polar decomposition. Although efficient parallel implementations of QDWH and ZOLO polar decomposition are available [36, 42], some effort is needed to ensure the recursive partition of the spectrum is well load balance in the divide-and-conquer procedure, and data communication required in the partition and back transformation is minimized. We will perform a careful comparison with this type of solver for large dense eigenvalue problems in future work.

For sparse problems, the current state of GPU accelerated linear algebra software for kernels relevant to the implementation of SISUBIT is far less mature than its dense counterpart. We have demonstrated that the current state of the art for distributed CPU/GPU implementations of sparse TRF (SuperLU_DIST) exhibits far better strong scaling in its CPU implementation than its GPU implementation. As such, for large processor counts, it is more beneficial to use the CPU implementations at this time. We note however that the superior performance of the PARDISO implementations of TRF and TRS indicate that GPU acceleration of sparse symmetric solvers would be further beneficial to achieving better performing implementations of sparse SISUBIT. As was aforementioned, integration of other GPU sparse solvers such as PaStiX ans WSMP will be explored in future work.

Despite these results for distributed sparse and dense implementations of SISUBIT, we remain optimistic for the future. The development of GPU accelerated distributed memory linear algebra is still very much in its infancy relative to its CPU counterparts, and is currently a very active research topic in the fields of high-performance computing and numerical linear algebra. As these implementations improve, so will the performance of SISUBIT. The fact that even the CPU implementation of DM-SISUBIT outperforms the current state of the art for CPU and GPU dense eigensolvers (ELPA) indicates a bright future for the GPU acceleration of parallel SISS.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. This research was also partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors thank the National Energy Research Scientific Computing (NERSC) center and the Oak Ridge Leadership Computing Facilities (OLCF) for making computational resources available to them.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719604

[2] Zhaojun Bai and James Demmel. 1998. Using the Matrix Sign Function to Compute Invariant Subspaces. *SIAM J. Matrix Anal. Appl.* 19, 1 (1998), 205–225. https://doi.org/10.1137/S0895479896297719 arXiv:https://doi.org/10.1137/S0895479896297719

[3] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. 2000. *Templates for the Solution of Algebraic Eigenvalue Problems.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719581 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898719581

[4] Mohamed-Ali Belabbas and Patrick J Wolfe. 2009. Spectral methods in machine learning and new strategies for very large datasets. *Proceedings of the National Academy of Sciences* 106, 2 (2009), 369–374. https://doi.org/10.1073/pnas.0810600105

[5] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. 1997. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719642

[6] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. 2007. Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *The International Journal of High Performance Computing Applications* 21, 4 (2007), 457–466. https://doi.org/10.1177/1094342007084026 arXiv:https://doi.org/10.1177/1094342007084026

[7] Carmen Campos and Jose E Roman. 2012. Strategies for spectrum slicing based on restarted Lanczos methods. *Numerical Algorithms* 60, 2 (2012), 279–295.

[8] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[9] Arne De Coninck, Bernard De Baets, Drosos Kourounis, Fabio Verbosio, Olaf Schenk, Steven Maenhout, and Jan Fostier. 2016. Needles: Toward Large-Scale Genomic Prediction with Marker-by-Environment Interaction. 203, 1 (2016), 543–555. https://doi.org/10.1534/genetics.115.179887 arXiv:http://www.genetics.org/content/203/1/543.full.pdf

[10] James W. Demmel, Osni A. Marques, Beresford N. Parlett, and Christof VÃƒÂ¶mel. 2007. *A Testing Infrastructure for LAPACK's Symmetric Eigensolvers.* Technical Report 182. LAPACK Working Note. http://www.netlib.org/lapack/lawnspdf/lawn182.pdf

[11] J. Duersch, M. Shao, C. Yang, and M. Gu. 2018. A Robust and Efficient Implementation of LOBPCG. *SIAM Journal on Scientific Computing* 40, 5 (2018), C655–C676. https://doi.org/10.1137/17M1129830

[12] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 133–137.

[13] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19).* Association for Computing Machinery, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

[14] Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. https://doi.org/10.1137/0710032 arXiv:https://doi.org/10.1137/0710032

[15] Laura Grigori, James W. Demmel, and Xiaoye S. Li. 2007. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing* 29, 3 (2007), 1289–1314. https://doi.org/10.1137/050638102 arXiv:https://doi.org/10.1137/050638102

[16] Roger G Grimes, John G Lewis, and Horst D Simon. 1994. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl.* 15, 1 (1994), 228–272.

[17] Anshul Gupta, Natalia Gimelshein, Seid Koric, and Steven Rennich. 2016. Effective minimally-invasive GPU acceleration of distributed sparse matrix factorization. In *European Conference on Parallel Processing*. Springer, 672–683. https://doi.org/10.1007/978-3-319-43659-3_49

[18] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[19] Thomas Herault, Yves Robert, George Bosilca, and Jack Dongarra. 2019. Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PARSEC. In *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 33–41.

[20] Yulu Jia, Piotr Luszczek, , and Jack Dongarra. 2013. *Transient Error Resilient Hessenberg Reduction on GPU-based Hybrid Architectures*. Technical Report 279. LAPACK Working Note. http://www.netlib.org/lapack/lawnspdf/lawn279.pdf

[21] Murat Keçeli, Hong Zhang, Peter Zapol, David A Dixon, and Albert F Wagner. 2016. Shift-and-invert parallel spectral transformation eigensolver: Massively parallel performance for density-functional based tight-binding. *Journal of computational chemistry* 37, 4 (2016), 448–459.

[22] J. Kesyn, V. Kalantzis, E. Polizzi, and Y. Saad. 2016. PFEAST: A High Performance Sparse Eigenvalue Solver Using Distributed-Memory Linear Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE Supercomputing Conference (SC16)*. 16:1–16:12.

[23] Murat Keçeli, Fabiano Corsetti, Carmen Campos, Jose E. Roman, Hong Zhang, Álvaro Vázquez-Mayagoitia, Peter Zapol, and Albert F. Wagner. 2018. SIESTA-SIPs: Massively parallel spectrum-slicing eigensolver for an ab initio molecular dynamics package. *Journal of Computational Chemistry* 39, 22 (2018), 1806–1814. https://doi.org/10.1002/jcc.25350

[24] A. Knyazev. 2001. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. *SIAM Journal on Scientific Computing* 23, 2 (2001), 517–541. https://doi.org/10.1137/S1064827500366124

[25] D. Kourounis, A. Fuchs, and O. Schenk. 2018. Towards the Next Generation of Multiperiod Optimal Power Flow Solvers. *IEEE Transactions on Power Systems* PP, 99 (2018), 1–10. https://doi.org/10.1109/TPWRS.2017.2789187

[26] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. *SLATE Working Note 12: Implementing Matrix Inversions*. Technical Report ICL-UT-19-04. Innovative Computing Laboratory, University of Tennessee. revision 07-2019.

[27] Jakub Kurzak, Mark Gates, Ichitaro Yamazaki, Ali Charara, Asim YarKhan, Jamie Finney, Gerald Ragghianti, Piotr Luszczek, and Jack Dongarra. 2018. *SLATE Working Note 8: Linear Systems Performance Report*. Technical Report ICL-UT-18-08. Innovative Computing Laboratory, University of Tennessee. revision 09-2018.

[28] Jakub Kurzak, Piotr Luszczek, Mathieu Faverge, and Jack Dongarra. 2012. *LU Factorization with Partial Pivoting for a Multi-CPU, Multi-GPU Shared Memory System*. Technical Report 266. LAPACK Working Note. http://www.netlib.org/lapack/lawnspdf/lawn266.pdf

[29] Pavel Kus, Hermann Lederer, and Andreas Marek. 2017. GPU optimization of large-scale eigenvalue solver. In *European Conference on Numerical Mathematics and Advanced Applications*. Springer, 123–131.

[30] P Kus, A Marek, SS Koecher, H-H Kowalski, Christian Carbogno, Ch Scheurer, Karsten Reuter, Matthias Scheffler, and H Lederer. 2019. Optimizations of the eigensolvers in the ELPA library. *Parallel Comput.* 85 (2019), 167–177.

[31] R. B. Lehoucq, D. C. Sorensen, and C. Yang. 1998. *ARPACK Users' Guide*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719628

[32] Ruipeng Li, Yuanzhe Xi, Lucas Erlandson, and Yousef Saad. 2019. The eigenvalues slicing library (EVSL): Algorithms, implementation, and software. *SIAM Journal on Scientific Computing* 41, 4 (2019), C393–C415.

[33] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.* 29, 2 (2003), 110–140. https://doi.org/10.1145/779359.779361

[34] Lin Lin, Yousef Saad, and Chao Yang. 2016. Approximating spectral densities of large matrices. *SIAM Rev.* 58, 1 (2016), 34–65.

[35] Yang Liu, Mathias Jacquelin, Pieter Ghysels, and Xiaoye S. Li. [n.d.]. *Highly scalable distributed-memory sparse triangular solution algorithms*. 87–96. https://doi.org/10.1137/1.9781611975215.9 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611975215.9

[36] Hatem Ltaief, Dalal Sukkari, Aniello Esposito, Yuji Nakatsukasa, and David Keyes. 2019. Massively Parallel Polar Decomposition on Distributed-Memory Systems. *ACM Trans. Parallel Comput.* 6, 1, Article 4 (June 2019), 15 pages. https://doi.org/10.1145/3328723

[37] Andreas Marek, Volker Blum, Rainer Johanni, Ville Havu, Bruno Lang, Thomas Auckenthaler, Alexander Heinecke, Hans-Joachim Bungartz, and Hermann Lederer. 2014. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter* 26, 21 (2014), 213201. https://doi.org/10.1088/0953-8984/26/21/213201

[38] Yuji Nakatsukasa and Roland W. Freund. 2016. Computing Fundamental Matrix Decompositions Accurately via the Matrix Sign Function in Two Iterations: The Power of Zolotarev's Functions. *SIAM Rev.* 58, 3 (2016), 461–493. https://doi.org/10.1137/140990334 arXiv:https://doi.org/10.1137/140990334

[39] Yuji Nakatsukasa and Nicholas J. Higham. 2013. Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD. *SIAM Journal on Scientific Computing* 35, 3 (2013), A1325–A1349. https://doi.org/10.1137/120876605 arXiv:https://doi.org/10.1137/120876605

[40] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. 2014. A Distributed CPU-GPU Sparse Direct Solver. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 487–498.

[41] Gerard LG Sleijpen and Henk A Van der Vorst. 2000. A Jacobi–Davidson iteration method for linear eigenvalue problems. *SIAM review* 42, 2 (2000), 267–293.

[42] Dalal Sukkari, Hatem Ltaief, and David Keyes. 2016. A High Performance QDWH-SVD Solver Using Hardware Accelerators. *ACM Trans. Math. Softw.* 43, 1, Article 6 (Aug. 2016), 25 pages. https://doi.org/10.1145/2894747

[43] Fabio Verbosio, Arne De Coninck, Drosos Kourounis, and Olaf Schenk. 2017. Enhancing the scalability of selected inversion factorization algorithms in genomic prediction. *Journal of Computational Science* 22, Supplement C (2017), 99 – 108. https://doi.org/10.1016/j.jocs.2017.08.013

[44] David B. Williams-Young, Paul G. Beckman, and Chao Yang. 2019. A Shift Selection Strategy for Parallel Shift-Invert Spectrum Slicing in Symmetric Self-Consistent Eigenvalue Computation. *arXiv preprint arXiv:1908.06043* (2019).

[45] Yuanzhe Xi, Ruipeng Li, and Yousef Saad. 2018. Fast Computation of Spectral Densities for Generalized Eigenvalue Problems. *SIAM Journal on Scientific Computing* 40, 4 (2018), A2749–A2773. https://doi.org/10.1137/17M1135542 arXiv:https://doi.org/10.1137/17M1135542

[46] Chao Yang. 2005. Solving large-scale eigenvalue problems in SciDAC applications. *Journal of Physics: Conference Series* 16 (jan 2005), 425–434. https://doi.org/10.1088/1742-6596/16/1/058

[47] Hong Zhang, Barry Smith, Michael Sternberg, and Peter Zapol. 2007. SIPs: Shift-and-invert parallel spectral transformations. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (2007), 9.