cucheb: A GPU IMPLEMENTATION OF THE FILTERED LANCZOS PROCEDURE

JARED L. AURENTZ[*], VASSILIS KALANTZIS[†], AND YOUSEF SAAD[†]

Abstract. This paper describes a graphics processing unit (GPU) implementation of the Filtered Lanczos Procedure for the solution of large, sparse, symmetric eigenvalue problems, with a focus on eigenvalue problems that arise in the context of electronic structure calculations under the Density Functional Theory framework. The Filtered Lanczos Procedure uses a carefully chosen polynomial spectral transformation to accelerate the convergence of the Lanczos method when computing eigenvalues within a desired interval. This method has proven particularly effective when Matrix-Vector products can be performed efficiently. We illustrate, via example, that the Filtered Lanczos Procedure implemented on a GPU can greatly accelerate eigenvalue computations for certain classes of symmetric matrices common in electronic structure calculations. Comparisons against previously published CPU results suggest a typical speedup of a factor of $20 - 30$.

Key words. graphics processing units, symmetric Lanczos, eigenvalues, eigenvectors, large sparse matrices, quantum mechanics, graph partitioning

AMS subject classifications. 65F15, 65F50, 65Y05, 68W10

1. Introduction. This paper describes a graphics processing unit (GPU) acceleration of polynomial filtering procedures for computing all eigenvalue-eigenvector pairs (eigenpairs) of a symmetric matrix A inside a given interval $[\alpha, \beta]$. Polynomial filtering eigenvalue solvers are based on selecting a polynomial spectral transformation of the original matrix so that the eigenvalues of interest are mapped to the extreme part of the spectrum of the transformed matrix. Then, a projection method, e.g., Lanczos, is applied to the transformed matrix, leading to fast convergence [19]. These approaches can be effective for computing eigenpairs lying anywhere in the spectrum in situations where the matrix of interest has a nearly uniform spectral distribution and the Matrix-Vector multiplication can be performed efficiently in serial or parallel architectures.

Polynomial filtering techniques have been particularly effective for the eigenvalue problems that arise in electronic structure calculations [19, 38, 39, 47–49]. In the Density Functional Theory (DFT) framework, the solution of the all-electron Schrödinger equation is replaced by a one-electron Schrödinger equation with an effective potential which leads to a nonlinear eigenvalue problem, known as the Kohn-Sham equation [20, 26]:

$$\left[-\frac{\nabla^2}{2} + V_{ion}(r) + V_H(\rho(r), r) + V_{XC}(\rho(r), r) \right] \Psi_i(r) = E_i \Psi_i(r), \qquad (1.1)$$

where $\Psi_i(r)$ is a wave function and E_i is a Kohn-Sham eigenvalue. The ionic potential

V_{ion} reflects contributions from the core and depends on the position r only. Both the Hartree and the Exchange-Correlation potentials depend on the charge density:

$$\rho(r) = 2 \sum_{i=1}^{n_{occ}} |\Psi_i(r)|^2, \qquad (1.2)$$

where n_{occ} is the number of occupied states (for most systems of interest this is half the number of valence electrons). Since the total potential $V_{total} = V_{ion} + V_H + V_{XC}$ depends on $\rho(r)$ which itself depends on eigenfunctions of the Hamiltonian, Equation (1.1) can be viewed as a nonlinear eigenvalue problem or a *nonlinear eigenvector problem*. The Hatree potential V_H is obtained from ρ by solving the Poisson equation $\nabla^2 V_H(r) = -4\pi\rho(r)$ with appropriate boundary conditions. The Exchange-Correlation term V_{XC} is the key to the DFT approach and it captures the effects of reducing the problem from many particles to a one-electron problem, i.e., from replacing wavefunctions with many coordinates into ones that depend solely on space location r.

Self-consistent iterations for solving the Kohn-Sham equation start with an initial guess of the charge density $\rho(r)$, from which a guess for V_{total} is computed. Then (1.1) is solved for $\Psi_i(r)$'s and a new $\rho(r)$ is obtained from (1.2) and the potentials are updated. Then (1.1) is solved again for a new ρ obtained from the new $\Psi_i(r)$'s, and the process is repeated until the total potential has converged.

A typical electronic structure calculation with many atoms requires the calculation of a large number of eigenvalues, specifically the n_{occ} leftmost ones. In addition, calculations based on Time-Dependent Density Functional Theory [10, 12], require a substantial number of unoccupied states, states beyond the Fermi level, in addition to the occupied ones. Thus, it is not uncommon to see eigenvalue problems in the size of millions where (tens) of thousands of eigenvalues may be needed.

As a result, efficient numerical methods that can be easily parallelized in current high-performance computing environments are therefore essential in electronic structure calculations. The high computational power offered by GPUs has increased their presence in the numerical linear algebra community and they are gradually becoming an important tool of scientific codes for solving large-scale, computationally intensive eigenvalue problems. While GPUs are mostly known for their high speedups relative to CPU-bound operations*, sparse eigenvalue computations can also benefit from hybrid CPU-GPU architectures. Although published literature and scientific codes for the solution of sparse eigenvalue problems on a GPU have not been as common as those that exist for multi-CPU environments, recent studies conducted independently by some of the authors of this paper demonstrated that the combination of polynomial filtering eigenvalue solvers with GPUs can be beneficial [2, 23]. The goal of this paper is twofold :

- We provide a simple and flexible framework for implementing polynomial filtering eigenvalue solvers on a GPU and for exploiting the capabilities offered by current high-performance accelerators. We give a detailed description of our software package, called `cucheb`, which can be downloaded from
https://github.com/jaurentz/cucheb
and includes all techniques presented in this paper.

---

*See also the MAGMA project at http://icl.cs.utk.edu/magma/index.html

- We study and analyze the performance of the `cucheb` software package for solving eigenvalue problems that originate from electronic structure calculations, and show that the combination of GPUs and polynomial filtering can be quite effective for these problems.

The paper is organized as follows. Section 2 introduces polynomial filtering for eigenvalue problems and provides the basic formulation of the filters used. Section 3 discusses the proposed GPU implementation of the filtered Lanczos procedure. Section 4 presents computational results with the proposed GPU implementations. Finally, concluding remarks are presented in Section 5.

2. The Filtered Lanczos Procedure. The Lanczos algorithm and its variants [4, 11, 17, 28, 31, 43, 46] are well-established methods for computing a subset of the spectrum of a symmetric (real) matrix. These methods are especially adept at approximating eigenvalues lying at the extreme part of the spectrum [5, 29, 35, 42], e.g., eigenvalues of largest magnitude. When the desired eigenvalues are well inside the spectral interval these techniques can become ineffective and lead to large computational and memory costs. Traditionally, this is overcome by mapping interior eigenvalues to the exterior part of the spectrum, e.g., by a shift-and-invert spectral transformation (see for example [37] or [44]). While shift-and-invert can be highly effective in some situations, a matrix factorization is now necessary and this can be too costly for matrices that generate a large amount of fill-in.

The Filtered Lanczos Procedure (FLP) offers an appealing alternative for such cases. In this approach interior eigenvalues are mapped to the end parts of the spectrum of the transformed matrix using a polynomial spectral transformation. A Lanczos procedure is then used on the transformed matrix [19]. Such a transformation only requires Matrix-Vector multiplications, a task that is often easy to parallelize for sparse matrices. The procedure requires bounds on the spectrum as well as a well-selected polynomial.

2.1. Polynomial spectral transformations. Let $A \in \mathbb{R}^{n \times n}$ be symmetric and real and let

$$A = V \Lambda V^T \tag{2.1}$$

be its spectral decomposition, where $V \in \mathbb{R}^{n \times n}$ is an orthogonal matrix and $\Lambda \in \mathbb{R}^{n \times n} = \mathrm{diag}\,(\lambda_1, \ldots, \lambda_n)$ is diagonal. We consider *spectral transformations* of A, i.e., mappings of the form

$$f(A) = V f(\Lambda) V^T, \tag{2.2}$$

where $f(\Lambda) = \mathrm{diag}\,(f(\lambda_1), \ldots, f(\lambda_n))$ and f is any (real or complex) function f defined on the spectrum of A. Standard examples in eigenvalue computations include the shift-and-invert transformation $f(z) = (z - \rho)^{-1}$ and $f(z) = z^k$ for subspace iteration.

In a polynomial spectral transformation the function f is a polynomial. A desirable filter polynomial p should satisfy the following requirements: a) the desired eigenvalues of A are the largest in magnitude eigenvalues of $p(A)$, b) the construction of p requires minimal knowledge of the spectrum of A, and c) multiplying a vector by $p(A)$ is relatively inexpensive.

Our implementation of the FLP constructs polynomials that satisfy the above requirements using techniques from digital filter design. The basic idea is to construct a transforming polynomial by approximating an "ideal" filter which maps the desired eigenvalues of A to eigenvalues of largest magnitude in $p(A)$.

2.2. Constructing polynomial transformations. Throughout this section it is assumed that the spectrum of A is contained entirely in the interval $[-1, 1]$. In practice, this assumption poses no restrictions since the eigenvalues of A located inside the interval $[\lambda_{\min}, \lambda_{\max}]$, where $\lambda_{\min}$, $\lambda_{\max}$ denote the algebraically smallest and largest eigenvalues of A, respectively, can be mapped to the interval $[-1, 1]$ by the following linear transformation:

$$A := (A - cI)/e, \quad c = \frac{\lambda_{\min} + \lambda_{\max}}{2}, \quad e = \frac{\lambda_{\max} - \lambda_{\min}}{2}. \tag{2.3}$$

Since $\lambda_{\min}$ and $\lambda_{\max}$ are exterior eigenvalues of A, one can obtain very good estimates by performing a few Lanczos steps.[†] We will see in Section 4 that computing such estimates constitutes only a modest fraction of the total compute time.

Let then $[\alpha, \beta] \subset [-1, 1]$, be a subinterval and assume we wish to compute all eigenvalues of A in $[\alpha, \beta]$ along with their corresponding eigenvectors. Consider first the following spectral transformation:

$$\phi(z) = \begin{cases} 1, & z \in [\alpha, \beta], \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

The function ϕ is just an indicator function, taking the value 1 inside the interval $[\alpha, \beta]$ and zero outside. When acting on A, ϕ maps the desired eigenvalues of A to the repeated eigenvalue 1 for $\phi(A)$ and all the unwanted eigenvalues to 0. Moreover, the eigenvectors which correspond to eigenvalues of A within the interval $[\alpha, \beta]$ are identical to the eigenvectors of $\phi(A)$ which correspond to the multiple eigenvalue 1. Thus, applying Lanczos on $\phi(A)$ computes the same invariant subspace, with the difference that now the eigenvalues of interest (mapped to 1) are well-separated from the unwanted ones (mapped to zero), and rapid convergence can be established. Unfortunately, such a transformation is not practically significant as there is no cost-effective way to multiply a vector by $\phi(A)$.

A practical alternative is to replace ϕ with a polynomial p such that $p(z) \approx \phi(z)$ for all $z \in [-1, 1]$. Such a p will then map the desired eigenvalues of A to a neighborhood of 1 for $p(A)$. Moreover, since p is a polynomial, applying $p(A)$ to a vector only requires Matrix-Vector multiplication with A.

In order to quickly construct a p that is a good approximation to ϕ it is important that we choose a good basis. For functions supported on $[-1, 1]$ the obvious choice is Chebyshev polynomials of the first kind. Such representations have already been used successfully for constructing polynomial spectral transformations and for approximating matrix-valued functions in quantum mechanics (see for example [2, 6, 19, 22, 36, 39, 40, 45, 47–49]).

Recall that the Chebyshev polynomials of the first kind obey the three-term recurrence

$$T_{i+1}(z) = 2zT_i(z) - T_{i-1}(z), \ i \geq 1. \tag{2.5}$$

starting with $T_0(z) = 1$, $T_1(z) = z$. The Chebyshev polynomials satisfy the following orthogonality condition and so they form a complete orthogonal set for the Hilbert

[†]In practice, if $\bar{\lambda}_{\min}$, $\bar{\lambda}_{\max}$ are approximations to eigenvalues $\lambda_{\min}$, $\lambda_{\max}$, and $\|\rho_{\min}\|$, $\|\rho_{\max}\|$ are the residuals of the approximate eigenpairs associated with the corresponding eigenvalues, one could use the values $\bar{\lambda}_{\min} - \|\rho_{\min}\|$ and $\bar{\lambda}_{\max} + \|\rho_{\max}\|$ to scale the matrix.

space $L^2_\mu([-1,1])$, $d\mu(z) = (1-z^2)^{-1/2} dz$:

$$\int_{-1}^1 \frac{T_i(z)T_j(z)}{\sqrt{1-z^2}} dz = \begin{cases} \pi, & i = j = 0, \\ \frac{\pi}{2}, & i = j > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.6}$$

Since $\phi \in L^2_\mu([-1,1])$ it possesses a convergent Chebyshev series

$$\phi(z) = \sum_{i=0}^\infty b_i T_i(z), \tag{2.7}$$

where the $\{b_i\}_{i=0}^\infty$ are defined as follows, where δ_{ij} represents the Dirac delta symbol:

$$b_i = \frac{2 - \delta_{i0}}{\pi} \int_{-1}^1 \frac{\phi(z)T_i(z)}{\sqrt{1-z^2}} dz \tag{2.8}$$

For a given α and β the $\{b_i\}$ are known analytically (see for example [21]),

$$b_i = \begin{cases} (\arccos(\alpha) - \arccos(\beta))/\pi, & i = 0, \\ 2\left(\sin\left(i\arccos(\alpha)\right) - \sin\left(i\arccos(\beta)\right)\right)/i\pi, & i > 0. \end{cases} \tag{2.9}$$

An obvious choice for constructing p is to fix a degree m and truncate the Chebyshev series of ϕ,

$$p_m(z) = \sum_{i=0}^m b_i T_i(z). \tag{2.10}$$

Due to the discontinuities of ϕ, p_m does not converge to ϕ uniformly as $m \to \infty$. The lack of uniform convergence is not an issue as long as the filter polynomial separates the wanted and unwanted eigenvalues. Figure 2.1 illustrates two polynomial spectral transformations constructed by approximating ϕ on two different intervals. Even with the rapid oscillations near the ends of the subinterval, these polynomials are still good candidates for separating the spectrum.
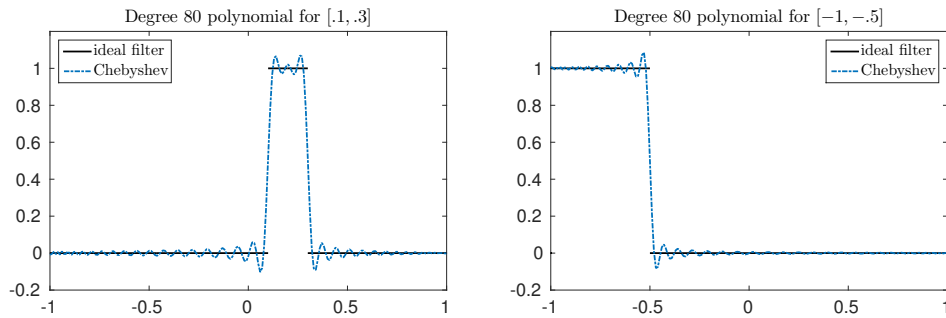
FIG. 2.1. *Chebyshev approximation of the ideal filter ϕ using a degree 80 polynomial. Left: $[\alpha, \beta] = [.1, .3]$, right: $[\alpha, \beta] = [-1, -.5]$.*

Figure 2.1 shows approximations of the ideal filter ϕ for two different subintervals of $[-1,1]$, using a fixed degree $m = 80$. In the left subfigure the interval of interest is located around the middle of the spectrum $[\alpha, \beta] = [.1, .3]$, while in the right

subfigure the interval of interest is located at the left extreme part $[\alpha, \beta] = [-1, -.5]$. Note that the oscillations near the discontinuities do not prevent the polynomials from separating the spectrum.

Since A is sparse, multiplying $p(A)$ by a vector can be done efficiently in parallel using a vectorized version of Clenshaw's algorithm [13] when p is represented in a Chebyshev basis. Clenshaw's algorithm can be run entirely in real arithmetic whenever the Chebyshev coefficients of p are real.

2.3. Filtered Lanczos as an algorithm. Assuming a transforming polynomial p, we can approximate eigenvalues of A by first approximating eigenvalues and eigenvectors of $p(A)$ using a simple version of the Lanczos method [28]. Many of the matrices arising in practical applications possess repeated eigenvalues, requiring the use of block Lanczos algorithm [17], so we describe the block version of FLP as it contains the standard algorithm as a special case.

Given a block size r and a matrix $Q \in \mathbb{R}^{n \times r}$ with orthonormal columns, the Filtered Lanzos Procedure iteratively constructs an orthonormal basis for the Krylov subspace generated by $p(A)$ and Q:

$$\mathcal{K}_k(p(A), Q) = \text{span}\{Q, p(A)Q, \ldots, p(A)^{k-1}Q\}. \tag{2.11}$$

Let us denote by $Q_k \in \mathbb{R}^{n \times rk}$ the matrix whose columns are generated by $k-1$ steps of the block Lanczos algorithm. Then, for each integer k we have $Q_k^T Q_k = I$ and $\text{range}(Q_k) = \text{span}(\mathcal{K}_k(p(A), Q))$. Since $p(A)$ is symmetric the columns of Q_k can be generated using short recurrences. This implies that there exists symmetric $\{D_i\}_{i=1}^k$ and upper-triangular $\{S_i\}_{i=1}^k$, $D_i, S_i \in \mathbb{R}^{r \times r}$, $i = 1, \ldots, k$ such that

$$p(A)Q_k = Q_{k+1}\tilde{T}_k, \tag{2.12}$$

where

$$\tilde{T}_k = \begin{bmatrix} T_k \\ S_k E_k^T \end{bmatrix}, \quad T_k = \begin{bmatrix} D_1 & S_1^T & & & \\ S_1 & D_2 & S_2^T & & \\ & S_2 & D_3 & \ddots & \\ & & \ddots & \ddots & S_{k-1}^T \\ & & & S_{k-1} & D_k \end{bmatrix}, \tag{2.13}$$

and $E_k \in \mathbb{R}^{kr \times r}$ denotes the last r columns of the identity matrix of size $kr \times kr$. Left multiplying (2.12) by Q_k^T gives the Rayleigh-Ritz projection

$$Q_k^T p(A) Q_k = T_k. \tag{2.14}$$

The matrix T_k is symmetric and banded, with a semi-bandwidth of size r. The eigenvalues of T_k are the Ritz values of $p(A)$ associated with the subspace spanned by the columns of Q_k and for sufficiently large k the dominant eigenvalues of $p(A)$ will be well approximated by these Ritz values. Of course we aren't actually interested in the eigenvalues of $p(A)$ but those of A. We can recover these eigenvalues by using the fact that $p(A)$ has the same eigenvectors as A. Assuming that an eigenvector v of $p(A)$ has been computed accurately we can recover the corresponding eigenvalue λ of A from the the Rayleigh quotient of v:

$$\lambda = \frac{v^T A v}{v^T v}. \tag{2.15}$$

In practice we will only have a (perhaps very accurate) approximation $\hat{v}$ of v. The approximate eigenvector $\hat{v}$ will be a Ritz vector of $p(A)$ associated with Q_k. To compute these Ritz vectors we first compute an eigendecomposition of T_k. Since T_k is real and symmetric there exists an orthogonal matrix $W_k \in \mathbb{R}^{rk \times rk}$ and a diagonal matrix $\Lambda_k \in \mathbb{R}^{rk \times rk}$ such that

$$T_k W_k = W_k \Lambda_k. \tag{2.16}$$

Combining (2.12) and (2.16), the Ritz vectors of $p(A)$ are formed as $\hat{V}_k = Q_k W_k$.

3. `cucheb`: a GPU implementation of the Filtered Lanczos Procedure. A key advantage of the Filtered Lanczos Procedure is that it requires only Matrix-Vector multiplication, an operation that uses relatively low memory and that is typically easy to parallelize compared to solving large linear systems. FLP and related methods have already been successfully implemented on multi-core CPUs and distributed memory machines [39].

3.1. The GPU architecture. A Graphical Processing Unit (GPU) is a Single Instruction Multiple Data (SIMD) scalable model which consists of multi-threaded Streaming Multiprocessors (SMs), each one equipped with multiple Scalar Processor cores (SPs), with each SP performing the same instruction on its local portion of data. While they were initially developed for the purposes of graphics processing, GPUs were adapted in recent years for general purpose computing. The development of the Compute Unified Device Architecture (CUDA) [16] parallel programming model by NVIDIA, an extension of the C language, provides an easy way for computational scientists to take advantage of the GPU raw power.

Under the CUDA programming model, each SP executes one thread at a time and in a parallel fashion from the other SPs. The SIMD model simplifies the hardware design of the GPU unit and allows this space to be filled by many SPs which results to a throughput-oriented model. Threads and blocks of threads are executed in an asynchronous manner and threads within a block cooperate by using explicit synchronization calls (barriers) and block-private shared memory space.

After a block of threads is assigned to an SM, it is internally divided in warps -a fixed number of threads determined in hardware- and executed warp-by-warp. Having a large number of threads per block, and thus a large number of warps, can hide latency since every time threads in a particular warp remain idle waiting for an I/O instruction, another warp can be swapped in and executed.

3.2. Implementation details of the `cucheb` software package. In this section we discuss the details of our GPU implementation of FLP. Our implementation will consist of a high-level, open source C++ library called `cucheb` [3] which depends only on the NVIDIA CUDA Toolkit [16, 30] and standard C++ libraries, allowing for easy interface with NVIDIA brand GPUs. At the user level, the `cucheb` software library consists of three basic data structures:

- `cuchebmatrix`
- `cucheblanczos`
- `cuchebpoly`

The remainder of this section is devoted to describing the role of each of these data structures.

3.3. Sparse matrices and the `cuchebmatrix` object. The first data structure, called `cuchebmatrix`, is a container for storing and manipulating sparse matrices.

This data structure consists of two sets of pointers, one for data stored in CPU memory and one for data stored in GPU memory. Such a duality of data is often necessary for GPU computations if one wishes to avoid costly memory transfers between the CPU and GPU. To initialize a `cuchebmatrix` object one simply passes the path to a symmetric matrix stored in the Matrix Market file format [9]. The following segment of `cucheb` code illustrates how to initialize a `cuchebmatrix` object using the matrix `H2O` downloaded from the University of Florida sparse matrix collection [18]:

```
#include "cucheb.h"

int main(){

  // declare cuchebmatrix variable
  cuchebmatrix ccm;

  // create string with Matrix Market file name
  string mtxfile("H2O.mtx");

  // initialize ccm using Matrix Market file
  cuchebmatrix_init(&mtxfile, &ccm);

  .
  .
  .


}
```

The function `cuchebmatrix_init` opens the data file, checks that the matrix is real and symmetric, allocates the required memory on the CPU and GPU, reads the data into CPU memory, converts it to an appropriate format for the GPU and finally copies the data into GPU memory. By appropriate format we mean that the matrix is stored on the GPU in compressed sparse row (CSR) format with no attempt to exploit the symmetry of the matrix. CSR is used as it is the most generic storage scheme for performing sparse Matrix-Vector multiplications using the GPU. See [7,34] and references therein for a discussion on the performance of sparse Matrix-Vector multiplications in the CSR and other formats. Once a `cuchebmatrix` object has been created, sparse Matrix-Vector multiplications can then be performed on the GPU using the NVIDIA CUSPARSE library [15].

3.4. Lanczos and the `cucheblanczos` object. The second data structure, called `cucheblanczos`, is a container for storing and manipulating the vectors and matrices associated with the Lanczos process. As with the `cuchebmatrix` objects, a `cucheblanczos` object possesses pointers to both CPU and GPU memory. While there is a function for initializing a `cucheblanczos` object, the average user should never do this explicitly. Instead they should call a higher level routine like `cuchebmatrix_lanczos` which takes as an argument an uninitialized `cucheblanczos` object. Such a routine will then calculate an appropriate number of Lanczos vectors based on the input matrix and initialize the `cucheblanczos` object accordingly.

Once a `cuchebmatrix` object and corresponding `cucheblanczos` object have been initialized, one of the core Lanczos algorithms can be called to iteratively construct

the Lanczos vectors. Whether iterating with A or $p(A)$, the core Lanczos routines in `cucheb` are essentially the same. The algorithm starts by constructing an orthonormal set of starting vectors (matrix Q in (2.11)). Once the vectors are initialized the algorithm expands the Krylov subspace, peridiocaly checking for convergence. To check convergence the projected problem (2.16) is copied to the CPU, the Ritz values are computed and the residuals are checked. If the algorithm has not converged the Krylov subspace is expanded further and the projected problem is solved again. For stability reasons `cucheb` uses full reorthogonalization to expand the Krylov subspace, making the algorithm more akin to the Arnoldi method [1]. Due to the full reorthogonalization, the projected matrix T_k from (2.16) will not be symmetric exactly but it will be symmetric to machine precision, which justifies the use of an efficient symmetric eigensolver (see for example [32]). The cost of solving the projected problem is negligible compared to expanding the Krylov subspace, so we can afford to check convergence often. All the operations required for reorthogonalization are performed on the GPU using the NVIDIA CUBLAS library [14]. Solving the eigenvalue problem for T_k is done on the CPU using a special purpose built banded symmetric eigensolver included in the `cucheb` library.

It is straight-forward to use selective reorthogonalization [33, 41, 42] or implicit restarts [4, 46], though we don't make use of these techniques in our code. In Section 4 we will see that the dominant cost in the algorithm is the Matrix-Vector multiplication with $p(A)$, so reducing the number of products with $p(A)$ is the easiest way to shorten the computation time. Techniques like implicit restarting can often increase the number of iterations if the size of the maximum allowed Krylov subspace is too small, meaning we would have to perform more Matrix-Vector multiplications. Our experiments suggest that the best option is to construct a good filter polynomial and then use increasingly larger Krylov subspaces until the convergence criterion is met.

All the Lanczos routines in `cucheb` are designed to compute all the eigenvalues in a user prescribed interval $[\alpha, \beta]$. When checking for convergence the Ritz values and vectors are sorted according to their proximity to $[\alpha, \beta]$ and the method is considered to be converged when all the Ritz values in $[\alpha, \beta]$ as well as a few of the nearest Ritz values outside the interval have sufficiently small residuals. If the iterations were done using A then the computation is complete and the information is copied back to the CPU. If the iterations were done with $p(A)$ the Rayleigh quotients are first computed on the GPU and then the information is copied back to the CPU.

To use Lanczos with A to compute all the eigenvalues in $[\alpha, \beta]$ a user is required to input five variables:

1. a lower bound on the desired spectrum (α)
2. an upper bound on the desired spectrum (β)
3. a block size
4. an initialized `cuchebmatrix` object
5. an uninitialized `cucheblanczos` object

The following segment of `cucheb` code illustrates how to do this using the function `cuchebmatrix_lanczos` for the interval $[\alpha, \beta] = [.5, .6]$, a block size of 3 and an already initialized `cuchebmatrix` object:

```
#include "cucheb.h"

int main(){
```

```
// initialize cuchebmatrix object
cuchebmatrix ccm;
string mtxfile("H2O.mtx");
cuchebmatrix_init(&mtxfile, &ccm);

// declare cucheblanczos variable
cucheblanczos ccl;

// compute eigenvalues in [.5,.6] using block Lanczos
cuchebmatrix_lanczos(.5, .6, 3, &ccm, &ccl);

.
.
.


}
```

This function call will first approximate the upper and lower bounds on the spectrum of the `cuchebmatrix` object. It then uses these bounds to make sure that the interval $[\alpha, \beta]$ is valid. If it is, it will adaptively build up the Krylov subspace as described above, periodically checking for convergence. For large matrices or subintervals well inside the spectrum, standard Lanczos may fail to converge all together. A better choice is to call the routine `cuchebmatrix_filteredlanczos` which automatically constructs a filter polynomial and then uses FLP to compute all the eigenvalues in $[\alpha, \beta]$.

3.5. Filter polynomials and the `cuchebpoly` object. To use FLP one needs a way to store and manipulate filter polynomials stored in a Chebyshev basis. In `cucheb` this is done with the `cuchebpoly` object. The `cuchebpoly` object contains pointers to CPU and GPU memory which can be used to construct and store filter polynomials. For the filter polynomials from Section 2 one only needs to store the degree, the Chebyshev coefficients and upper and lower bounds for the spectrum of A.

As with `cucheblanczos` objects, a user typically will not need to initialize a `cuchebpoly` object themselves as it will be handled automatically by a higher level routine. In `cuchebmatrix_filteredlanczos` for example, not only is the `cuchebpoly` object for the filter polynomial initialized but also the degree at which the Chebyshev approximation should be truncated is computed. This is done using a simple formula based on heuristics and verified by experiment. Assuming the spectrum of A is in $[-1, 1]$, a "good" degree m for $[\alpha, \beta] \subset [-1, 1]$ is computed using the following formula:

$$m = \min\{m > 0 : ||p_m - \phi|| < \epsilon ||\phi||\}, \tag{3.1}$$

where $||f||$ is the weighted Chebyshev 2-norm. The tolerance ϵ is a parameter and is chosen experimentally, with the goal of maximizing the separation power of the filter while keeping the polynomial degree and consequently the computation time low.

Figure 3.1 uses the same ideal filters from Figure 2.1 but this time computes the filter degree based on (3.1). In the left subfigure the interval of interest is located around the middle of the spectrum $[\alpha, \beta] = [.1, .3]$ and the distance between α and β is relatively small, giving a filter degree of 48. In the right subfigure the interval of interest is located at the left extreme part of the spectrum $[\alpha, \beta] = [-1, -.5]$ and the
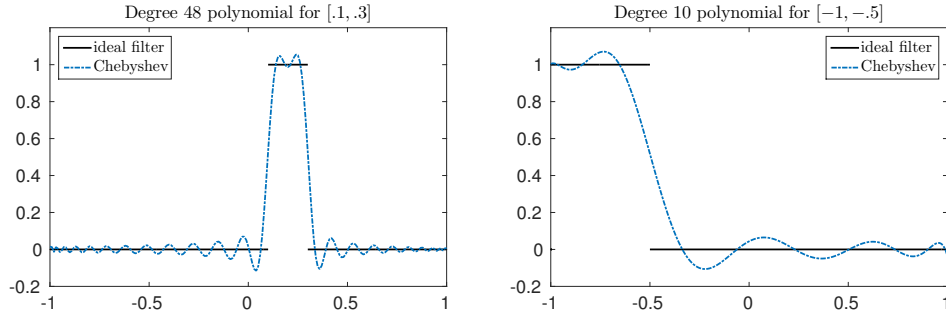
FIG. 3.1. *Chebyshev and approximation of the ideal filter ϕ. Left: $[\alpha, \beta] = [.1, .3]$ with an optimal degree of 48, right: $[\alpha, \beta] = [-1, -.5]$ with an optimal degree of 10.*

distance α and β is relatively large, giving a filter degree of 10. Although these filters seem worse approximations than those in Figure 2.1, the lower degrees lead to much shorter computation times.

The following segment of `cucheb` code illustrates how to use the function `cuchebmatrix_filteredlanczos` to compute all the eigenvalues in the interval $[\alpha, \beta] = [.5, .6]$ of an already initialized `cuchebmatrix` object using FLP with a block size of 3:

```
#include "cucheb.h"

int main(){

  // initialize cuchebmatrix object
  cuchebmatrix ccm;
  string mtxfile("H2O.mtx");
  cuchebmatrix_init(&mtxfile, &ccm);

  // declare cucheblanczos variable
  cucheblanczos ccl;

  // compute eigenvalues in [.5,.6] using block filtered Lanczos
  cuchebmatrix_filteredlanczos(.5, .6, 3, &ccm, &ccl);

  .
  .
  .

}
```

4. Experiments. In this section we illustrate the performance of our GPU implementation of the Filtered Lanczos Procedure. Our test matrices (Hamiltonians) originate from electronic structure calculations. In this setting, one is typically interested in computing a few eigenvalues around the Fermi level of each Hamiltonian. The Hamiltonians were generated using the PARSEC package [27], and can be also found
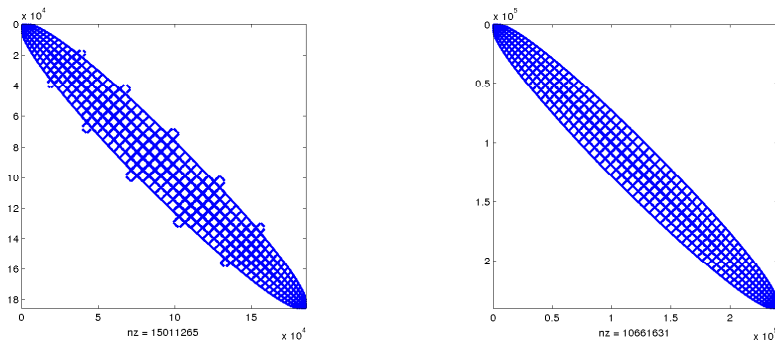
FIG. 4.1. *Sparsity pattern of the PARSEC matrices. Left: Si41Ge41H72. Right: Si87H76.*

in the University of Florida Sparse Matrix Collection [18].[‡] The Hamiltonians are real, symmetric, and have clustered, as well as multiple, eigenvalues. Table 4.1 lists the size n, the total number of non-zero entries nnz, as well as the endpoints of the spectrum of each matrix, i.e., the interval defined by the algebraically smallest/largest eigenvalues. The average number of nonzero entries per row for each Hamiltonian is quite large, a consequence of the high-order discretization and the addition of a (dense) 'non-local' term. Figure 4.1 plots the sparsity pattern of matrices Si41Ge41H72 (left) and Si87H76 (right).

All GPU experiments in this section were implemented using the `cucheb` library and performed on the same machine which has dual Intel Xeon ES-2667 v2 3.30GHz processors with 256GB of CPU RAM and two NVIDIA K40 GPUs each with 12GB of GPU RAM and 2880 compute cores. We make no attempt to access mutliple GPUs and all the experiments were performed using a single K40.

Matrix	n	nnz	nnz/n	Spectral interval
Ge87H76	$112,985$	$7,892,195$	69.9	$[-1.21e+0,\quad 3.28e+1]$
Ge99H100	$112,985$	$8,451,395$	74.8	$[-1.23e+0,\quad 3.27e+1]$
Si41Ge41H72	$185,639$	$15,011,265$	80.9	$[-1.21e+0,\quad 4.98e+1]$
Si87H76	$240,369$	$10,661,631$	44.4	$[-1.20e+0,\quad 4.31e+1]$
Ga41As41H72	$268,096$	$18,488,476$	69.0	$[-1.25e+0,\quad 1.30e+3]$

TABLE 4.1

A list of the PARSEC matrices used to evaluate our GPU implementation, where n is the dimension of the matrix, nnz is the number of nonzero entries and $[\lambda_{\min}, \lambda_{\max}]$ is the spectral interval.

Exploiting eigenvalue solvers that are based on matrix factorizations, e.g., shift-and-invert, has been shown to be impractical for matrices of the PARSEC matrix collection [24, 25]. The reason is that performing the LU factorization of each Hamiltonian results in a huge amount of fill-in in the associated triangular factors, requiring an excessive amount of memory and computations [24]. On the other hand, polynomial filtering accesses the Hamiltonians in their original form and only requires an efficient Matrix-Vector multiplication routine. Polynomial filtering has been often

[‡]https://www.cise.ufl.edu/research/sparse/matrices/

reported to be the most efficient numerical method for solving eigenvalue problems with the PARSEC matrix collection [19, 38, 39, 47–49]. This observation led to the development of FILTLAN, a C/C++ software package which implements the Filtered Lanczos Procedure with partial reorthgonalization [19] for serial architectures. The `cucheb` library featured in this paper, although implemented in CUDA, shares many similarities with FILTLAN. There are, however, a few notable differences. `cucheb` does not implement partial reorthgonalization as is the case in FILTLAN. Moreover, `cucheb` includes the ability to use block counterparts of the Lanczos method, and which can be more efficient in the case of multiple or clustered eigenvalues. In contrast to FILTLAN, which uses a least-squares filter polynomial, `cucheb` utilizes Chebyshev-type filters.

4.1. GPU benchmarking. Results are summarized in Table 4.2. Variable 'interval' for each Hamiltonian was set so that it included roughly the same number of eigenvalues from the left and right side of the Fermi level, and, in total, it included 'eigs' eigenvalues. For each matrix and interval $[\alpha, \beta]$ we repeated the same experiment three times, each time using a different degree m for the filter polynomial. Variable 'iters' shows the number of FLP iterations, while 'MV' shows the total number of Matrix-Vector products (MV) with A, which is computed using the formula 'MV'=$rm\times$ 'iters'. Through this section, the block size of the FLP will be equal to $r = 3$. Finally, variables 'time' and 'residual' show the total compute time and maximum relative residual of the computed eigenpairs. The first two rows for each matrix correspond to executions where the degree m was selected a priori. The third row corresponds to an execution where the degree was selected automatically by our implementation, using the mechanism described in (3.1). As expected, using larger values for m leads to faster convergence in terms of total iterations, since higher degree filters are better at separating the wanted and unwanted portions of the spectrum. Although larger degrees lead to less iterations, the amount of work in each filtered Lanczos iteration is also increases proportionally. This might lead to an increase of the actual computational time, an effect verified for each one of the matrices in Table 4.2.

Table 4.3 compares the percentage of total computation time required by the different subprocesses of the FLP method. We denote the preprocessing time, which consists solely of approximating the upper and lower bounds of the spectrum for A, by 'PREPROC'. We also denote the total amount of time spent in the full reorthogonalization and the total amount of time spent in performing all MV products of the form $p(A)v$ on the GPU, by 'ORTH' and 'MV' respectively. As we can verify, all matrices in this experiment devoted less than 10% of the total compute time to estimate the spectral interval (i.e. the eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$). For each one of the PARSEC test matrices, the dominant cost came from the MV products, due to their relatively large number of non-zero entries. Note that using a higher degree m will shift the cost more towards the MV products, since the Lanczos procedure will typically converge in fewer outer steps and thus the orthogonalization cost reduces.

4.2. CPU-GPU comparison. Figure 4.2 shows the speedup of the GPU FLP implementation over the CPU-based counterpart. The CPU results were obtained by executing the FILTLAN software package in a purely sequential mode on the `Mesabi` linux cluster at University of Minnesota Supercomputing Institute. `Mesabi` consists of by 741 nodes of various configurations with a total of 17,784 compute cores provided by Intel Haswell E5-2680v3 processors. Each node features two sockets, each socket with six physical cores (12 with hyper-threading), and each core with a clock speed

Matrix	interval	eigs	m	iters	MV	time	residual
Ge87H76	$[-0.645, -0.0053]$	212	50	210	$31,500$	44	$4.3e{-}14$
			100	180	$54,000$	62	$6.4e{-}13$
			49	210	$30,870$	43	$2.1e{-}13$
Ge99H100	$[-0.650, -0.0096]$	250	50	210	$31,500$	45	$3.7e{-}13$
			100	180	$54,000$	65	$4.0e{-}12$
			49	210	$30,870$	45	$5.1e{-}13$
Si41Ge41H72	$[-0.640, -0.0028]$	218	50	210	$31,500$	77	$3.2e{-}13$
			100	180	$54,000$	112	$2.7e{-}11$
			61	180	$32,940$	76	$6.3e{-}13$
Si87H76	$[-0.660, -0.3300]$	107	50	150	$22,500$	55	$1.3e{-}14$
			100	90	$27,000$	56	$3.3e{-}15$
			98	90	$26,460$	55	$1.5e{-}14$
Ga41As41H72	$[-0.640, 0.0000]$	201	300	180	$162,000$	386	$3.2e{-}15$
			400	180	$216,000$	506	$8.1e{-}15$
			308	180	$166,320$	396	$2.5e{-}15$

TABLE 4.2

Computing the eigenpairs inside an interval using FLP with various filter polynomial degrees. Times listed are in seconds.

Matrix	m	iters	PREPROC	ORTH	MV
Ge87H76	50	210	5%	15%	69%
	100	180	3%	8%	82%
	49	210	5%	15%	68%
Ge99H100	50	210	4%	15%	70%
	100	180	3%	8%	83%
	49	210	4%	15%	69%
Si41Ge41H72	50	210	7%	14%	70%
	100	180	5%	8%	83%
	61	180	7%	11%	75%
Si87H76	50	150	8%	16%	69%
	100	90	8%	8%	82%
	98	90	8%	8%	81%
Ga41As41H72	300	180	2%	3%	93%
	400	180	2%	2%	95%
	308	180	2%	3%	93%

TABLE 4.3

Percentage of total compute time required by various components of the algorithm. For all these examples the dominant computational cost is the MV multiplication.

of 2.5 GHz. Each node is also equipped with 64 GB of RAM memory.

We have divided the comparison in two parts: a "low degree" situation when $m = 50$ ($m = 300$ for Ga41As41H72), and a "high degree" situation when $m = 100$ ($m = 400$ for Ga41As41H72). In both cases, the GPU implementation obtains a

speedup which ranges between $10 - 40$ with a tendency for larger speedups when larger degrees are used.
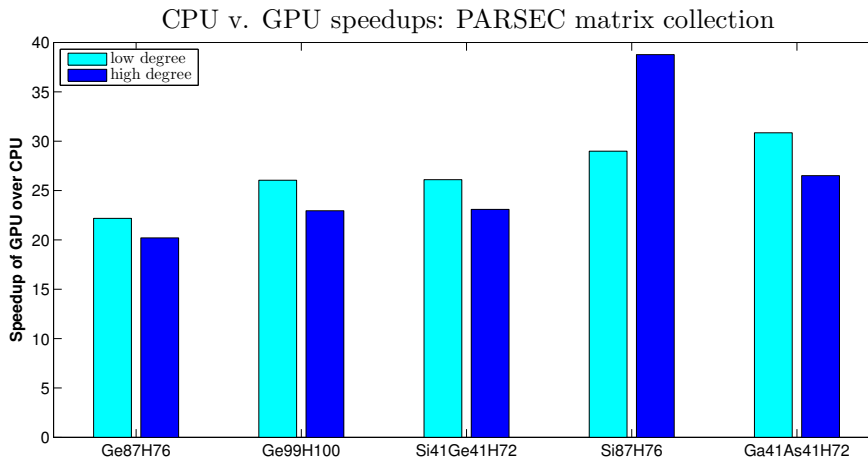

FIG. 4.2. *Speedup of the GPU FLP implementation over the CPU (FILTLAN) for the PARSEC test matrices.*

5. Conclusion. In this work we presented a GPU implementation of a Filtered Lanczos Procedure for solving large and sparse eigenvalue problems such as those that arise from real-space Density Functional Theory methods in electronic structure calculations. Our experiments indicate that the use of GPU architectures in the context of electronic structure calculations can provide a speedup of at least a factor of $20 - 30$ over CPU implementations.

Possible future research directions include the utilization of more than one GPUs to perform the filtered Lanczos procedure in computing environments with access to multiple GPUs. Each different GPU can then be used to either perform the sparse Matrix-Vector products and other operations of the FLP in parallel, or compute all eigenpairs in a sub-interval of the original interval. The implementation proposed in this paper can be used without any modifications. Another interesting extension would be to use additional customization and add support for other sparse matrix formats except the CSR format. A dense matrix version of the proposed implementation would also be of interest for solving sequences of eigenvalue problems as in [8].

REFERENCES

[1] W. E. ARNOLDI, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quart. Appl. Math., 9 (1951), pp. 17–29.
[2] J. L. AURENTZ, *GPU accelerated polynomial spectral transformation methods*, PhD thesis, Washington State University, 2014.
[3] J. L. AURENTZ AND V. KALANTZIS, *cucheb*. https://github.com/jaurentz/cucheb, 2015.
[4] J. BAGLAMA, D. CALVETTI, AND L. REICHEL, *IRBL: An implicitly restarted block-Lanczos method for large-scale hermitian eigenproblems*, SIAM J. Sci. Comp., 24 (2003), pp. 1650–1677.
[5] C. A. BEATTIE, M. EMBREE, AND D. C. SORENSEN, *Convergence of polynomial restart krylov methods for eigenvalue computations*, SIAM Rev., 47 (2005), pp. 492–515.
[6] C. BEKAS, E. KOKIOPOULOU, AND Y. SAAD, *Computation of large invariant subspaces using polynomial filtered Lanczos iterations with applications in density functional theory*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 397–418.

[7] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[8] M. Berljafa, D. Wortmann, and E. Di Napoli, *An optimized and scalable eigensolver for sequences of eigenvalue problems*, Concurrency and Computation: Practice and Experience, 27 (2015), pp. 905–922.

[9] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, *Matrix market: A web resource for test matrix collections*, in Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement, London, UK, UK, 1997, Chapman & Hall, Ltd., pp. 125–137.

[10] W. R. Burdick, Y. Saad, L. Kronik, Manish Jain, and James Chelikowsky, *Parallel implementations of time-dependent density functional theory*, Computer Physics Communications, 156 (2003), pp. 22–42.

[11] D. Calvetti, L. Reichel, and D. C. Sorensen, *An implicitly restarted Lanczos method for large symmetric eigenvalue problems*, Electron. Trans. Numer. Anal., 2 (1994), p. 21.

[12] J. R. Chelikowsky, Y. Saad, and I. Vasiliev, *Atoms and clusters*, in Time-Dependent Density Functional Theory, vol. 706 of Lecture notes in Physics, Springer-Verlag, Berlin, Heidelberg, 2006, ch. 17, pp. 259–269. DOI-10.1007/3-540-35426-3-17.

[13] C. W. Clenshaw, *A note on the summation of Chebyshev series*, Math. Tab. Wash., 9 (1955), pp. 118–120.

[14] NVIDIA Corporation, *CUBLAS Library User Guide*, v7.0 ed., October 2015.

[15] ——, *CUSPARSE Library User Guide*, v7.0 ed., October 2015.

[16] ——, *NVIDIA CUDA C Programming Guide*, v7.0 ed., October 2015.

[17] J. Cullum and W. E. Donath, *A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices*, in Decision and Control including the 13th Symposium on Adaptive Processes, 1974 IEEE Conference on, IEEE, 1974, pp. 505–509.

[18] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.

[19] H. Fang and Y. Saad, *A filtered Lanczos procedure for extreme and interior eigenvalue problems*, SIAM J. Sci. Comp., 34 (2012), pp. A2220–A2246.

[20] P. Hohenberg and W. Kohn, *Inhomogeneous electron gas*, Phys. Rev., 136 (1964), pp. B864–B871.

[21] D. Jackson, *The Theory of Approximation*, vol. 11 of Colloquium publications, AMS, New York, NY, USA, 1930.

[22] L. O. Jay, H. Kim, Y. Saad, and J. R. Chelikowsky, *Electronic structure calculations for plane-wave codes without diagonalization*, Comput. Phys. Commun., 118 (1999), pp. 21–30.

[23] V. Kalantzis, *A GPU implementation of the filtered Lanczos algorithm for interior eigenvalue problems*. 2015.

[24] V. Kalantzis, J. Kestyn, E. Polizzi, and Y. Saad, *Domain decomposition contour integration approaches for symmetric eigenvalue problems*. 2016.

[25] V. Kalantzis, R. Li, and Y. Saad, *Spectral Schur complement techniques for symmetric eigenvalue problems*, Tech. Report ys-2015-3, 2015.

[26] W. Kohn and L. J. Sham, *Self-consistent equations including exchange and correlation effects*, Phys. Rev., 140 (1965), pp. A1133–A1138.

[27] L. Kronik, A. Makmal, M. L. Tiago, M. M. G. Alemany, M. Jain, X. Huang, Y. Saad, and J. R. Chelikowsky, *PARSEC–the pseudopotential algorithm for real-space electronic structure calculations: recent advances and novel applications to nano-structures*, Phys. Status Solidi (B), 243 (2006), pp. 1063–1079.

[28] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Nat. Bur. Standards, 45 (1950), pp. 255–282.

[29] R. B. Lehoucq, *Analysis and implementation of an implicitly restarted Arnoldi iteration*, PhD thesis, Rice University, 1995.

[30] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with cuda*, Queue, 6 (2008), pp. 40–53.

[31] C. C. Paige, *The computation of eigenvalues and eigenvectors of very large sparse matrices*, PhD thesis, University of London, 1971.

[32] B. N. Parlett, *The Symmetric Eigenvalue Problem*, SIAM, Philadelphia, PA, USA, 1980.

[33] B. N. Parlett and D. S. Scott, *The Lanczos algorithm with selective orthogonalization*, Math. Comp., 33 (1979), pp. 217–238.

[34] I. Reguly and M. Giles, *Efficient sparse matrix-vector multiplication on cache-based gpus*, in Innovative Parallel Computing, IEEE, 2012, pp. 1–12.

[35] Y. SAAD, *On the rates of convergence of the Lanczos and the block-Lanczos methods*, SIAM J. Numer. Anal., 17 (1980), pp. 687–706.

[36] ——, *Filtered conjugate residual-type algorithms with applications*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 845–870.

[37] ——, *Numerical Methods for Large Eigenvalue Problems*, SIAM, Philadelphia, PA, USA, Second ed., 2011.

[38] Y. SAAD, A. STATHOPOULOS, J. CHELIKOWSKY, K. WU, AND S. ÖĞÜT, *Solution of large eigenvalue problems in electronic structure calculations*, BIT, 36 (1996), pp. 563–578.

[39] G. SCHOFIELD, J. R. CHELIKOWSKY, AND Y. SAAD, *A spectrum slicing method for the Kohn–Sham problem*, Comput. Phys. Commun., 183 (2012), pp. 497 – 505.

[40] R. N. SILVER, H. ROEDER, A. F. VOTER, AND J. D. KRESS, *Kernel polynomial approximations for densities of states and spectral functions*, J. Comput. Phys., 124 (1996), pp. 115–130.

[41] H. D. SIMON, *Analysis of the symmetric Lanczos algorithm with reorthogonalization methods*, Linear Algebra Appl., 61 (1984), pp. 101–131.

[42] H. D. SIMON, *The Lanczos algorithm with partial reorthogonalization*, Math. Comp., 42 (1984), pp. 115–142.

[43] D. C. SORENSEN, *Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations*, tech. report, 1996.

[44] D. S. WATKINS, *The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods*, SIAM, Philadelphia, PA, USA, 2007.

[45] A. WEISSE, G. WELLEIN, A. ALVERMANN, AND H. FEHSKE, *The kernel polynomial method*, Rev. Modern Phys., 78 (2006), p. 275.

[46] K. WU AND H. SIMON, *Thick-restart Lanczos method for large symmetric eigenvalue problems*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 602–616.

[47] Y. ZHOU, *A block Chebyshev-Davidson method with inner-outer restart for large eigenvalue problems*, J. Comput. Phys., 229 (2010), pp. 9188 – 9200.

[48] Y. ZHOU AND Y. SAAD, *A Chebyshev-Davidson algorithm for large symmetric eigenproblems*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 954–971.

[49] Y. ZHOU, Y. SAAD, M. L. TIAGO, AND J. R. CHELIKOWSKY, *Self-consistent-field calculations using Chebyshev-filtered subspace iteration*, J. Comput. Phys., 219 (2006), pp. 172 – 184.