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. 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 in parallel. 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 and other applications. Comparisons against previously published CPU results suggest a typical speedup of a factor of 10.

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.

The solution of large and sparse eigenvalue problems holds a prominent role in a variety of scientific fields such as electronic structure calculations [36], network analysis [18], statistical analysis and machine learning [1], and recommender systems [27] to name a few. The large size of the underlying matrices as well as the need to compute a large number of eigenvalues, sometimes deep inside the spectrum, call for efficient numerical methods which can be also easily parallelized in current high-performance computing environments. The high computational power offered by graphics processing units (GPUs) has increased their presence in the numerical linear algebra community and they are gradually becoming an important ingredient of scientific codes for solving large-scale, computationally intensive eigenvalue problems. While GPUs are mostly known to offer high speedups when performing CPU-bound operations[1], 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 are certainly much scarcer than those that exist for multi-CPU environments, recent efforts try to bridge this gap. See [32] for a comparison of different eigenvalue solvers ported to multi-GPU environments for the solution of eigenvalue problems stemming from nanostructure simulations, or [42] for a GPU-based tridiagonal eigenvalue solver.

In this paper we focus on a GPU acceleration of the *Filtered Lanczos Procedure* for computing all eigenvalues of a symmetric matrix A inside a given interval $[\alpha, \beta]$ [17]. The Filtered Lanczos Procedure uses a carefully selected polynomial spectral

[*]This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/20072013)/ERC grant agreement no. 291068. The views expressed in this article are not those of the ERC or the European Commission, and the European Union is not liable for any use that may be made of the information contained here; and by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences DE-SC0008877.

[†]Mathematical Institute, University of Oxford, Andrew Wiles Building, Woodstock Road, OX2 6GG, Oxford, UK. (`aurentz@maths.ox.ac.uk`).

[‡]Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Bldg., 200 Union Street S.E., Minneapolis, Minnesota, 55455, USA. (`{kalantzi,saad}@cs.umn.edu`).

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

transformation of the original matrix so that the eigenvalues of interest are mapped to the extreme part of the spectrum while, at the same time, the gap region between the wanted and unwanted eigenvalues in the transformed space is enhanced. The Lanczos method (or any Krylov subspace method) applied to the transformed matrix then converges much faster. When the matrix of interest has a nearly uniform spectral distribution and matrix-vector multiplication can be performed efficiently in parallel, the polynomial filtering approach can be well suited for eigenvalue computations, especially for interior eigenvalue problems where the computational and memory cost required by Krylov subspace methods operating directly on A becomes prohibitive. Previous experimental studies performed by some of the authors of this article suggest that the utilization of GPUs can seriously enhance the performance of the filtered Lanczos procedure, with speedups ranging from factors of 10 [22] to factors of 100 [3].

The goal of this paper is twofold. First, we provide a flexible framework for implementing the Filtered Lanczos Procedure (FLP) on a GPU and exploiting the capabilities offered by current high-performance accelerators. Second, we study and analyze the performance of the proposed implementation on a variety of test matrices stemming from different applications, such as electronic structure calculations and graph theory, and show that the combination of GPUs and polynomial filtering can be very effective at efficiently solving sparse symmetric eigenvalue problems.

The paper is organized as follows. In Section 2 we discuss the concept of polynomial filtering in eigenvalue problems and provide the basic formulation of the filters used. In Section 3 we discuss in detail the proposed GPU implementation of the filtered Lanczos procedure. In Section 4 we present computational results with the proposed GPU implementations on matrices arising from different application fields such as electronic structure calculations or graph theory. Finally, in Section 5 we present our concluding remarks.

2. The Filtered Lanczos Procedure.

The Lanczos method and its variants [5,10,15,24,28,41,45] are established tools for computing a subset of the spectrum of a large symmetric matrix. These methods are especially adept at approximating eigenvalues near the periphery of the spectrum [6,25,33,40]. When the desired eigenvalues are well inside the spectral interval these techniques can suffer from slow convergence or possibly fail to converge at all. Traditionally this is overcome by moving interior eigenvalues to the exterior using a shift and invert spectral transformation (see for example [35] or [43]). While such transformations can be very effective, performing the required system solves can be prohibitively difficult, especially for large indefinite matrices. The Filtered Lanczos Procedure (FLP) overcomes this by moving interior eigenvalues to the exterior using a polynomial spectral transformation [17]. Such a transformation only requires matrix-vector multiplication, a task that is often easy to parallelize for sparse matrices. The drawback is that one requires bounds on the spectrum as well as a good transforming polynomial. For certain classes of matrices both of these challenges can be overcome, making FLP a good candidate for modern parallel architectures like the GPU.

2.1. Polynomial spectral transformations.

We begin our discussion of FLP with a brief overview of polynomial spectral transformations. Let $A \in \mathbb{R}^{n\times n}$ be symmetric. Since A is real and symmetric there exists an orthogonal matrix $V \in \mathbb{R}^{n\times n}$ and diagonal matrix $\Lambda \in \mathbb{R}^{n\times n}$ such that

$$AV = V\Lambda, \tag{2.1}$$

where the diagonal entries of Λ are the eigenvalues of A,

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}. \tag{2.2}$$

A *spectral transformation* of A is any complex-valued function f defined on the spectrum of A. The matrix $f(A)$ is defined as

$$f(A) = Vf(\Lambda)V^T, \tag{2.3}$$

where

$$f(\Lambda) = \begin{bmatrix} f(\lambda_1) & & \\ & \ddots & \\ & & f(\lambda_n) \end{bmatrix}. \tag{2.4}$$

Standard examples in eigenvalue computations include the shift and invert transformation $f(z) = (z - \rho)^{-1}$ and $f(z) = z^k$ for the power method and subspace iteration.

For FLP we will be interested in a special type of spectral transformation known as a *polynomial spectral transformation*. A polynomial spectral transformation is any spectral transformation that is also a polynomial.[2] In order for FLP to be a competitive method, a good polynomial spectral transformation p must satisfy the following four requirements:

1. The desired eigenvalues of A are peripheral eigenvalues of $p(A)$.
2. Construction of p requires minimal knowledge of the spectrum of A.
3. Multiplying a vector by $p(A)$ is easy to parallelize.
4. $p(A)$ is symmetric if A is symmetric.

In our implementation of the FLP we construct 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. For symmetric eigenvalue computations the notion of ideal is characterized by requirement 1, namely move the desired eigenvalues of A to the periphery of the spectrum of $p(A)$. While one could imagine many ways of doing this, our implementation aims to provide a simple construction technique for an important class of problems.

2.2. Constructing simple transforming polynomials. For now we assume that the spectrum of A is contained entirely in the interval $[-1, 1]$. Given a subinterval $[\alpha, \beta] \subset [-1, 1]$ of modest size, we wish to compute all the eigenvalues and corresponding eigenvectors within that interval. To do so we will construct a polynomial spectral transformation by approximating the following ideal filter ϕ:

$$\phi(z) = \begin{cases} 1, & z \in [\alpha, \beta], \\ 0, & \text{otherwise}. \end{cases} \tag{2.5}$$

Such a filter maps the desired eigenvalues of A to the repeated eigenvalue 1 for $\phi(A)$ and all the unwanted eigenvalues to 0. Since 1 is on the periphery of the spectrum

[2]In fact any spectral transformation of a symmetric matrix is equivalent to a polynomial spectral transformation [19], though using this equivalence in practice would require precise knowledge of the spectrum of A.

of $\phi(A)$ requirement 1 is met. Unfortunately such a transformation is not practically significant as there is no way to multiply a vector by $\phi(A)$ (requirement 3) without first knowing the spectrum of A. To get around this we will 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 $p(A)$ and requirement 1 will still be satisfied, albeit in a looser sense. Moreover, since p is a polynomial, applying $p(A)$ to a vector only requires matrix-vector multiplication with A. If A is sparse this can be done efficiently in parallel, which means that requirement 3 is also met.

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 [3, 7, 17, 21, 34, 37, 38, 44, 46–48]). For easy reference we define the first kind Chebyshev polynomials below.

Definition 2.1. *Let $T_0(z) = 1$, $T_1(z) = z$ and*

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

The functions $\{T_i(z)\}_{i=0}^{\infty}$ are called the Chebyshev polynomials of the first kind.

The Chebyshev polynomials satisfy the following orthogonality condition and consequently form a complete orthogonal set for the Hilbert space $L^2_\mu([-1, 1])$, $d\mu(z) = \left(1 - z^2\right)^{-1/2} dz$:

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

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:

$$\tilde{b}_i = \frac{1}{\pi} \int_{-1}^{1} \frac{\phi(z)T_i(z)}{\sqrt{1 - z^2}} dz, \quad b_i = \begin{cases} \tilde{b}_i, & i = 0, \\ 2\tilde{b}_i, & i > 0. \end{cases} \tag{2.8}$$

For a given α and β the $\{b_i\}$ are known analytically (see for example [20]),

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

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

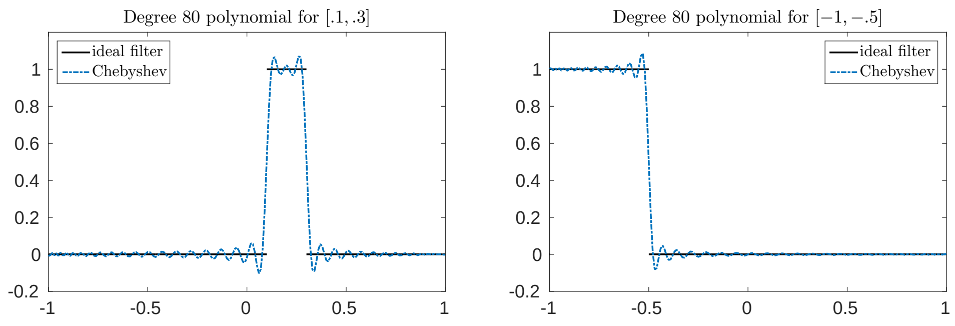

FIG. 2.1. *Chebyshev approximation of the ideal filter ϕ using a degree 80 polynomial. Left:* $[\alpha, \beta] = [.1, .3]$*, right:* $[\alpha, \beta] = [-1, -.5]$.

the wanted and unwanted eigenvalues. Figure 2.1 illustrates some possible 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.

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.

As we mentioned earlier, constructing polynomial spectral transformations this way automatically satisfies requirement 1. When A is sparse requirement 3 is also satisfied. This is due to the fact that multiplying $p(A)$ times a vector can be done efficiently in parallel using a vectorized version of Clenshaw's algorithm [11] 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. This means that $p(A)$ is a strictly real linear combination of real symmetric matrices and requirement 4 is also met. Satisfying requirement 2 is a bit more challenging. In general we will not know bounds for the spectrum of A a priori and indeed this is an important preprocessing step in our implementation. Assuming one could cheaply compute estimates for $\lambda_{\min}$ and $\lambda_{\max}$, respectively the smallest and largest eigenvalues of A, one could shift and scale A so that its spectrum is contained in $[-1, 1]$ using the following transformation:

$$A \to (\lambda_{\max} - \lambda_{\min})^{-1} \left(2A - (\lambda_{\max} + \lambda_{\min})I\right). \tag{2.11}$$

Since $\lambda_{\min}$ and $\lambda_{\max}$ are on the periphery of the spectrum of A, one can obtain very good estimates using standard Lanczos. We will see in Section 4 that computing such estimates constitutes only a modest fraction of the total compute time.

2.3. Filtered Lanczos as an algorithm. Assuming a transforming polynomial p that satisfies requirements 1-4 is already constructed, we can approximate eigenvalues of A by first approximating eigenvalues and eigenvectors of $p(A)$ using a simple version of the Lanczos method [24]. Many of the matrices arising in practice possess repeated eigenvalues requiring the use of block Lanczos [15], 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:

$$\mathcal{K}_k(p(A), q) = \text{span}\{q, p(A)q, \ldots, p(A)^{k-1}q\}. \tag{2.12}$$

Let $Q_k \in \mathbb{R}^{n\times rk}$ be the matrix whose columns are generated by $k-1$ steps of the Lanczos algorithm, then for each integer k we have $Q_k^T Q_k = I$ and

$$\text{range}(Q_k) = \text{span}\{q, p(A)q, \ldots, p(A)^{k-1}q\}. \tag{2.13}$$

Since $p(A)$ is symmetric the columns of Q_k can be generated using short recurrences. This implies 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

$$p(A)Q_k = Q_{k+1}\tilde{T}_k, \tag{2.14}$$

where

$$\tilde{T}_k = \begin{bmatrix} d_1 & s_1^T & & & \\ s_1 & d_2 & s_2^T & & \\ & s_2 & d_3 & \ddots & \\ & & \ddots & \ddots & s_{k-1}^T \\ & & & s_{k-1} & d_k \\ & & & & s_k \end{bmatrix}. \tag{2.15}$$

Left multiplying (2.14) by Q_k^T gives the standard Rayleigh-Ritz projection result

$$Q_k^T p(A) Q_k = T_k, \tag{2.16}$$

where

$$T_k = \begin{bmatrix} d_1 & s_1^T & & & \\ s_1 & d_2 & s_2^T & & \\ & s_2 & d_3 & \ddots & \\ & & \ddots & \ddots & s_{k-1}^T \\ & & & s_{k-1} & d_k \end{bmatrix}. \tag{2.17}$$

The matrix T_k is symmetric and has exactly $2r+1$ nonzero diagonals. 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 extremal 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 rather the eigenvalues of A. We can recover these eigenvalues by using the fact that transforming the eigenvalues of A does not change the corresponding eigenvectors. Thus if v is an eigenvector of A corresponding to λ then v is also an eigenvector of $p(A)$ corresponding to $p(\lambda)$,

$$Av = \lambda v \Rightarrow p(A)v = p(\lambda)v.$$

Furthermore, if $\lambda_i \neq \lambda_j \Rightarrow p(\lambda_i) \neq p(\lambda_j)$ for all λ_i in the spectrum of A then the eigenvectors of $p(A)$ are also eigenvectors of A,

$$p(A)v = p(\lambda)v \Rightarrow Av = \lambda v.$$

In practice p will always separate eigenvalues of A, though if the separation is small there is a danger of loss of accuracy in the computed eigenvectors of $p(A)$.

Assuming that the eigenvectors of $p(A)$ can be computed accurately we can recover the eigenvalues of A by first computing an approximation to v, call it $\hat{v}$, using $p(A)$ and then approximate λ using Rayleigh quotients,

$$\hat{\lambda} = \frac{\hat{v}^T A \hat{v}}{\hat{v}^T \hat{v}}. \tag{2.18}$$

It is well known that if $\hat{v}$ is a good approximation to v then $\hat{\lambda}$ is an excellent approximation to λ when A is symmetric [29]. 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

$$T_k W_k = W_k \Lambda_k. \tag{2.19}$$

Combining (2.14) and (2.19) we get the following:

$$p(A)V_k = V_{k+1}\tilde{\Lambda}_k, \tag{2.20}$$

where $V_k = Q_k W_k$,

$$V_{k+1} = Q_{k+1}\begin{bmatrix} W_k & \\ & I \end{bmatrix}$$

and

$$\tilde{\Lambda}_k = \begin{bmatrix} \Lambda_k \\ R_k \end{bmatrix}$$

with $R_k \in \mathbb{R}^{r\times rk}$. The columns of V_k are Ritz vectors of $p(A)$ and the diagonals of Λ_k the corresponding Ritz values. The i^{th} column of V_k is considered a good approximation to an eigenvector of $p(A)$ if the norm of the i^{th} column of R_k is small relative to $\|p(A)\|_2$. If the columns of V_k are sorted so that the diagonals of Λ_k are ordered from largest to smallest, i.e. if

$$\Lambda_k = \begin{bmatrix} \lambda_1^{(k)} & & \\ & \ddots & \\ & & \lambda_{rk}^{(k)} \end{bmatrix}$$

implies $\lambda_1^{(k)} \geq \ldots \geq \lambda_{rk}^{(k)}$, then the leading columns of V_k will correspond to the eigenvectors of A belonging to the eigenvalues in $[\alpha, \beta]$. This is true since p was constructed to move the eigenvalues of A in $[\alpha, \beta]$ to 1 which corresponds to the largest eigenvalues of $p(A)$. If the leading columns of R_k are sufficiently small we will have computed an approximate invariant subspace of $p(A)$ and consequently A. We can then recover the corresponding eigenvalues of A using (2.18).

3. `cucheb`: a GPU implementation of the Filtered Lanczos Procedure.

The key benefit of using the Filtered Lanczos Procedure is that it requires only matrix-vector multiplication, an operation that uses relatively low memory and is typically easy to parallelize compared to solving a large linear system. FLP and related methods have already been successfully implemented on multi-core CPUs and distributed memory machines [17, 37].

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` [4] which depends only on the NVIDIA CUDA Toolkit [14,26] 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.

3.1. Sparse matrices and the `cuchebmatrix` object.

The first data structure, called a `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 [16]:

```
#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 [8,31] 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 [13].

3.2. 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 (2.12)). Once the vectors are initialized the algorithm expands the Krylov subspace, peridiocally checking for convergence. To check convergence the projected problem (2.19) 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 [2]. Due to the full reorthogonalization, the projected matrix T_k from (2.19) 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 [29]). 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 [12]. Solving the eigenvalue problem for T_k is done on the CPU using a purpose built banded symmetric eigensolver included in the `cucheb` library.

It is straight forward to use selective reorthogonalization [30, 39, 40] or implicit restarts [5,45], 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 $[\alpha, \beta]$.

3.3. 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.

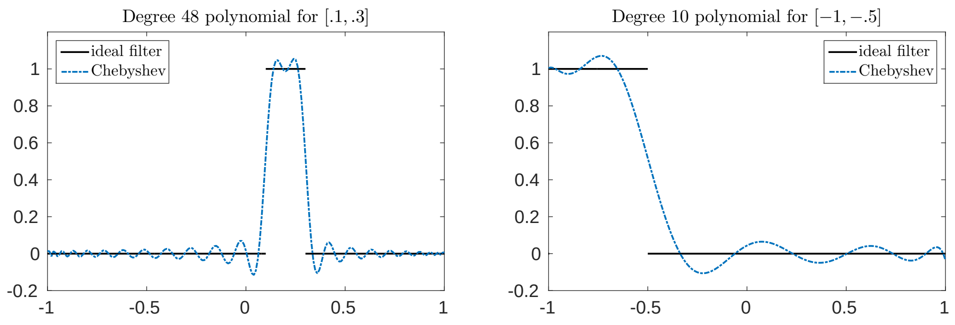

FIG. 3.1. *Chebyshev and approximation of the ideal filter* ϕ. *Left:* $[\alpha, \beta] = [.1, .3]$ *with an optimal degree of* 48, *right:* $[\alpha, \beta] = [-1, -.5]$ *with an optimal degree of* 5.

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 distance α and β is relatively large, giving a filter degree of 10. Although these filters look like 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 effectiveness of accelerating the Filtered Lanczos Procedure using a GPU. All the 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.

Table 4.1 gives a description of the basic properties of all our test matrices. All of these matrices can be found on the University of Florida sparse matrix collection [16]. The test matrices cover a wide range of characteristics, with some of them being fairly

Matrix	n	nnz	nnz/n	Spectral interval
`Ge87H76`	$112,985$	$7,892,195$	69.9	$[-1.21e{+}0,\ 3.28e{+}1]$
`Ge99H100`	$112,985$	$8,451,395$	74.8	$[-1.23e{+}0,\ 3.27e{+}1]$
`Si41Ge41H72`	$185,639$	$15,011,265$	80.9	$[-1.21e{+}0,\ 4.98e{+}1]$
`Si87H76`	$240,369$	$10,661,631$	44.4	$[-1.20e{+}0,\ 4.31e{+}1]$
`Ga41As41H72`	$268,096$	$18,488,476$	69.0	$[-1.25e{+}0,\ 1.30e{+}3]$
`fe_ocean`	$143,437$	$819,186$	5.7	$[-5.97e{+}0,\ 5.97e{+}0]$
`144`	$144,649$	$2,148,786$	14.9	$[-5.73e{+}0,\ 1.59e{+}1]$
`m14b`	$214,765$	$3,358,036$	15.6	$[-6.68e{+}0,\ 1.71e{+}1]$
`auto`	$448,695$	$6,629,222$	14.8	$[-6.60e{+}0,\ 1.70e{+}1]$
`caidaRouterLevel`	$192,244$	$1,218,132$	6.3	$[-1.08e{+}2,\ 1.09e{+}2]$
`mn2010`	$259,777$	$1,227,102$	4.7	$[-1.67e{+}7,\ 1.68e{+}7]$
`coPapersDBLP`	$540,486$	$30,491,458$	56.4	$[-6.41e{+}1,\ 3.63e{+}2]$
`ca2010`	$710,145$	$3,489,366$	4.9	$[-1.38e{+}7,\ 1.39e{+}7]$
`delaunay_n20`	$1,048,576$	$6,291,372$	6.0	$[-4.78e{+}0,\ 7.53e{+}0]$
`rgg_n_2_20_s0`	$1,048,576$	$13,783,240$	13.1	$[-6.56e{+}0,\ 2.64e{+}1]$

Table 4.1
A list of the 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.

dense, while others have a very wide spectrum. In the following, all timings are listed in seconds.

4.1. CPU-GPU comparison. The first experiment follows closely the experiment in [17, Sec. 5.2], where the authors used a multi-core CPU implementation of

FLP to compute all the eigenvalues of a matrix within a prescribed interval $[\alpha, \beta]$. Such problems are common in electronic structure calculations and the test matrices were generated using the software package PARSEC [23]. The purpose of this experiment is to illustrate the potential speedups one can achieve when using a GPU to accelerate the FLP method. The intervals $[\alpha, \beta]$ for each matrix were chosen as in [17]. The convergence of the FLP method was checked every 30 iterations.

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

Results are summarized in Table 4.2. 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. For each matrix we report the desired interval $[\alpha, \beta]$, the number of eigenvalues in the interval, the degree of the filter polynomial, the number of FLP iterations, the total number of matrix-vector products (MV) with A, the total compute time and the maximum relative residual of the computed eigenpairs. The block size of the FLP method was set to $r = 3$. 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 raised proportionally. The latter might lead to an increase of the actual computational time, an effect verified for each one of the matrices in Table 4.2. The same effect was also observed for the CPU-based FLP in [17] when tested on the same matrices.

Table 4.3 compares the percentage of total compute 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 on performing the full reorthogonalization and the total amount of time spent on performing all MV products of the form $p(A)v$ on the GPU, by ORTH and MV respectively. As we can verify, all the 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 nonzeros.

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 matrix-vector multiplication.

Finally, Figure 4.1 shows the speedup of the GPU FLP implementation over the CPU-based counterpart, for the PARSEC test matrices, where the CPU timings were extracted directly from [17, Sec. 5.2]. 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.

4.2. Lanczos versus Filtered Lanczos. The next set of examples illustrates the potential benefits of using FLP instead of standard Lanczos, when both schemes are implemented on the GPU. The key observation is that a good filter polynomial can lead to less Lanczos iterations which can both reduce computation time and memory requirements. We compare these methods by computing all the eigenvalues in an interval that contains the upper bound of the spectrum. Note that since the desired eigenvalues lie on the periphery of the spectrum, they are still computable using standard Lanczos. This type of computation is also common in graph partitioning algorithms where a percentage of the largest eigenvalues and eigenvectors are required (see [18] and the references therein).

We run our code on two different classes of matrices. The first class stems from problems in engineering, where the matrices possess a large number of eigenvalues

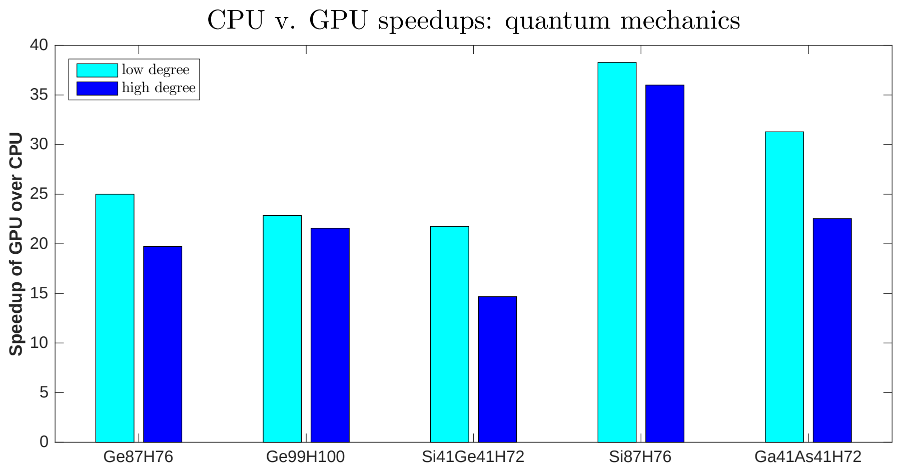

FIG. 4.1. *Speedup of the GPU FLP implementation over the CPU for the PARSEC test matrices. The reference timings for the CPU are extracted directly from [17].*

near the upper bound of the spectrum. For each matrix in the engineering class, Table 4.4 reports the fraction of the spectral interval that was computed, the number of eigenvalues in that interval, the degree of the filter polynomial ($m = 1$ implies standard Lanczos), the number of iterations required by the FLP method and standard Lanczos, the total number of MV products with A, the total compute time in seconds and the maximum relative residual of the computed eigenvalues. By fraction of the spectral interval we mean the ratio $(\beta - \alpha)/(\lambda_{\max} - \lambda_{\min})$ with $\beta \equiv \lambda_{\max}$. For both standard Lanczos and FLP we used a block size of $r = 1$. In the standard Lanczos computations ($m = 1$) we checked convergence of the approximate eigenpairs every 300 iterations. This reduced the number of projected eigenvalue problems that were solved to check convergence, ultimately lowering the total compute time for standard Lanczos. For FLP we checked convergence every 30 iterations.

Matrix	fraction	eigs	m	iters	MV	time	residual
`fe_ocean`	2%	217	47	480	22,560	19	$1.3e{-12}$
			1	4,200	4,200	803	$2.9e{-14}$
`144`	4%	195	33	450	14,850	19	$1.7e{-12}$
			1	2,700	2,700	252	$4.1e{-13}$
`m14b`	3%	235	38	510	19,380	37	$1.3e{-12}$
			1	3,600	3,600	833	$3.2e{-14}$
`auto`	4%	172	33	390	12,870	56	$1.8e{-11}$
			1	2,700	2,700	961	$3.5e{-14}$

TABLE 4.4
Comparison of Lanczos ($m = 1$) and FLP method for matrices arising in engineering. For each matrix we computed all the eigenvalues in the specified fraction of the total spectral interval that included the upper bound.

Table 4.5 shows the total number of iterations required for convergence as well as the percentage of total time required to perform the full reorthogonalization and MV products for the engineering class of matrices. Notation is kept the same as in previous tables. Standard Lanczos required significantly more iterations before convergence was achieved. This increased number of iterations meant more inner products and ultimately slower runtimes. For standard Lanczos, the largest portion of the compute time was spent performing the full reorthogonalizations. This is a direct consequence of the (relatively) high sparsity of the test matrices, as well as the large number of iterations needed for standard Lanczos to converge. On the other hand, FLP shifted the computational cost towards the MV products, an operation whose cost is only linearly dependented on the number of Lanczos iterations. Indeed, when counting the number of actual MV products with matrix A, FLP performs a much larger number of such products compared to standard Lanczos. However, the effect of these additional MV products is tiny compared to that of full reorthogonalization, if both operations are performed on a GPU. Thus, FLP can perform much better than standard Lanczos even when seeking eigenvalues on the periphery of the spectrum.

Matrix	m	iters	ORTH	MV
`fe_ocean`	47	480	29%	25%
	1	$4,200$	35%	<1%
`144`	33	450	27%	43%
	1	$2,700$	47%	1%
`m14b`	38	510	25%	48%
	1	$3,600$	56%	<1%
`auto`	33	390	23%	56%
	1	$2,700$	73%	1%

Table 4.5

Comparison of Lanczos ($m = 1$) and FLP method for matrices arising in engineering. For these matrices standard Lanczos required more iterations to converge and spent a higher percentage on the orthogonalization procedure.

Figure 4.2 shows the ratio of iterations and compute times for both methods. For each of the test matrices in the engineering class, standard Lanczos required at least 5 times as many iterations and ran 10 times slower than the FLP method.

The second class of matrices stems from applications in network analysis. The test matrices possess relatively few eigenvalues near the upper periphery of the spectrum. For each test matrix, Table 4.6 reports the fraction of the spectral interval that was computed, the number of eigenvalues in that interval, the degree of the filter polynomial (as previously, $m = 1$ denotes standard Lanczos), the number of total iterations required, the total number of MV products with A, the total compute time in seconds and the maximum relative residual of the computed eigenvalues. For both the FLP method and standard Lanczos we checked convergence of the approximate eigenpairs every 30 iterations. For the FLP method we used a block size of $r = 1$.

Table 4.7 shows the total number of iterations required for convergence as well as the percentage of total time required to perform the full reorthogonalization and MV products for the network analysis class of matrices. We can observe that the performance gap between the FLP method and standard Lanczos is now much narrower than what it was for the previous class of matrices, with the main reason lying in the small number of steps being necessary by standard Lanczos. In contrast with the matrices in the engineering class, the computed eigenvalues were now much bet-

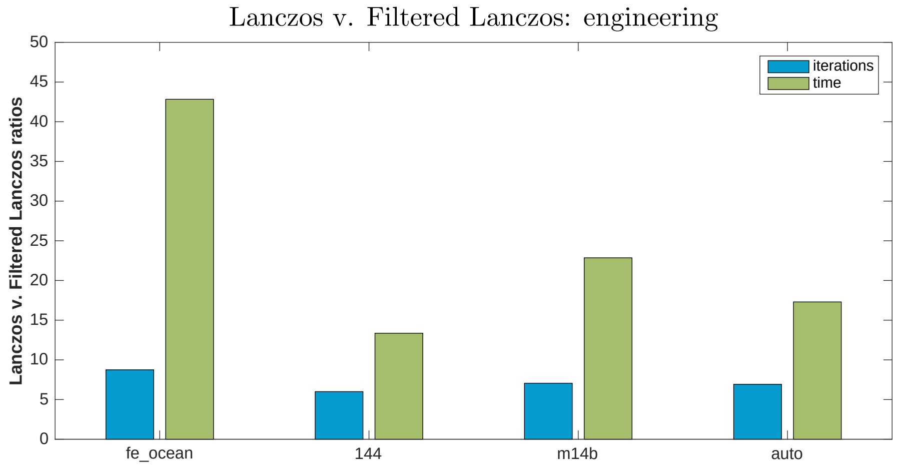

FIG. 4.2. *Comparison of Lanczos ($m = 1$) and FLP method for matrices arising in engineering. For these matrices Filtered Lanczos outperformed standard Lanczos in terms of number of iterations and total compute time.*

Matrix	fraction	eigs	m	iters	MV	time	residual
`caidaRouterLevel`	42%	137	10	570	5,700	21	$1.2e{-12}$
			1	780	780	41	$1.8e{-14}$
`mn2010`	42%	79	10	360	3,600	13	$3.0e{-13}$
			1	510	510	21	$1.9e{-14}$
`coPapersDBLP`	50%	134	9	360	3,240	44	$2.1e{-11}$
			1	630	630	57	$5.8e{-14}$
`ca2010`	38%	103	10	360	3,600	34	$5.2e{-12}$
			1	570	570	52	$2.3e{-14}$
`delaunay_n20`	9%	35	22	150	3,300	27	$1.8e{-13}$
			1	630	630	82	$9.8e{-14}$
`rgg_n_2_20_s0`	15%	24	17	120	2,040	26	$3.2e{-10}$
			1	420	420	44	$4.1e{-8}$

TABLE 4.6
Comparison of Lanczos ($m = 1$) and FLP for matrices arising in network analysis. For each matrix we computed all the eigenvalues in the specified fraction of the total spectral interval that included the upper bound.

ter separated and Lanczos could converge relatively quickly. Note also the difference in the degree m of the filter polynomials for this class of matrices, compared with the degree m in the previous (engineering) class. Here, $\beta - \alpha$ is comparable with $\lambda_{\max} - \lambda_{\min}$, which means that the optimal degree should not be too high.

Figure 4.3 shows the ratio of iterations and compute times for both methods. For each of these matrices standard Lanczos required more iterations and ran slower than Filtered Lanczos.

5. Conclusions.

In this work we presented a GPU implementation of the FLP method for the solution of large and sparse eigenvalue problems. We described the proposed implementation in detail and performed a series of experiments on problems

Matrix	m	iters	ORTH	MV
`caidaRouterLevel`	10	570	47%	15%
	1	780	38%	1%
`mn2010`	10	360	52%	12%
	1	510	54%	1%
`coPapersDBLP`	9	360	32%	41%
	1	630	56%	6%
`ca2010`	10	360	54%	12%
	1	570	68%	1%
`delaunay_n20`	22	150	33%	19%
	1	630	69%	1%
`rgg_n_2_20_s0`	17	120	26%	19%
	1	420	72%	2%

Table 4.7
Comparison of Lanczos ($m = 1$) and FLP for matrices arising in network analysis. For these matrices standard Lanczos required more iterations to converge and spent a higher percentage of time performing the orthogonalization procedure.

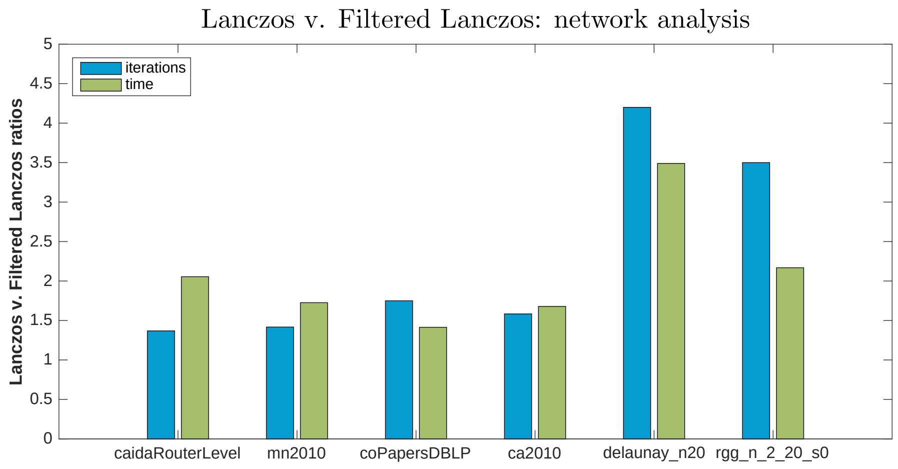

Fig. 4.3. *Comparison of Lanczos ($m = 1$) and FLP for matrices arising in network analysis. For these matrices Filtered Lanczos outperformed standard Lanczos in terms of number of iterations and total compute time.*

arising from electronic structure calculations, engineering and network analysis. Our experiments confirm that the use of GPU architectures in the context of electronic structure calculations can provide a speedup of at least a factor of 10 over a multicore-CPU implementation. In a similar manner, the GPU implementation of FLP can outperform Lanczos even when searching for a few eigenvalues on the periphery of the spectrum.

REFERENCES

[1] H. Abdi and L. J. Williams, *Principal component analysis*, Wiley Interdiscip. Rev. Comput. Stat., 2 (2010), pp. 433–459.
[2] W. E. Arnoldi, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quart. Appl. Math., 9 (1951), pp. 17–29.
[3] J. L. Aurentz, *GPU accelerated polynomial spectral transformation methods*, PhD thesis, Washington State University, 2014.
[4] J. L. Aurentz and V. Kalantzis, *cucheb*. `https://github.com/jaurentz/cucheb`, 2015.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[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] 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.
[11] C. W. Clenshaw, *A note on the summation of Chebyshev series*, Math. Tab. Wash., 9 (1955), pp. 118–120.
[12] NVIDIA Corporation, *CUBLAS Library User Guide*, v7.0 ed., October 2015.
[13] ———, *CUSPARSE Library User Guide*, v7.0 ed., October 2015.
[14] ———, *NVIDIA CUDA C Programming Guide*, v7.0 ed., October 2015.
[15] 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.
[16] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.
[17] H. Fang and Y. Saad, *A filtered Lanczos procedure for extreme and interior eigenvalue problems*, SIAM J. Sci. Comp., 34 (2012), pp. A2220–A2246.
[18] C. Fenu, D. Martin, L. Reichel, and G. Rodriguez, *Network analysis via partial spectral factorization and gauss quadrature*, SIAM J. Sci. Comp., 35 (2013), pp. A2046–A2068.
[19] N. J. Higham, *Functions of Matrices: Theory and Computation*, SIAM, Philadelphia, PA, USA, 2008.
[20] D. Jackson, *The Theory of Approximation*, vol. 11 of Colloquium publications, AMS, New York, NY, USA, 1930.
[21] 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.
[22] V. Kalantzis, *A GPU implementation of the filtered Lanczos algorithm for interior eigenvalue problems.* 2015.
[23] 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.
[24] 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.
[25] R. B. Lehoucq, *Analysis and implementation of an implicitly restarted Arnoldi iteration*, PhD thesis, Rice University, 1995.
[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with cuda*, Queue, 6 (2008), pp. 40–53.
[27] A. Nikolakopoulos, V. Kalantzis, and J. Garofalakis, *EIGENREC: An efficient and scalable latent factor family for top-N recommendation*, tech. report, 2015.
[28] C. C. Paige, *The computation of eigenvalues and eigenvectors of very large sparse matrices*,

PhD thesis, University of London, 1971.

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

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

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

[32] W. Rodrigues, A. Pecchia, M. Auf der Maur, and A. Di Carlo, *A comprehensive study of popular eigenvalue methods employed for quantum calculation of energy eigenstates in nanostructures using gpus*, J. Comput. Electron., 14 (2015), pp. 593–603.

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

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

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

[36] 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.

[37] 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.

[38] 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.

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

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

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

[42] V. Volkov and J. Demmel, *Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices*, Tech. Report UCB/EECS-2007-179, EECS Department, University of California, Berkeley, 2007.

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

[44] A. Weisse, G. Wellein, A. Alvermann, and H. Fehske, *The kernel polynomial method*, Rev. Modern Phys., 78 (2006), p. 275.

[45] K. Wu and H. Simon, *Thick-restart Lanczos method for large symmetric eigenvalue problems*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 602–616.

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

[47] Y. Zhou and Y. Saad, *A Chebyshev-Davidson algorithm for large symmetric eigenproblems*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 954–971.

[48] 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.