# A GPU implementation of the filtered Lanczos algorithm for the solution of symmetric interior eigenvalue problems

Vasileios Kalantzis
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
Email: kalantzi@cs.umn.edu

***Abstract*—In this paper we propose a GPU implementation of the filtered Lanczos algorithm for the solution of symmetric, interior eigenvalue problems. Some illustrative experiments are presented.**

## I. Introduction

In this paper we are interested in the solution of the standard symmetric eigenvalue problem

$$Au = \lambda u, \tag{1}$$

where $A$ is an $n \times n$ sparse and symmetric matrix, $\lambda \in \mathbb{R}$ is an eigenvalue and $u \in \mathbb{R}^n$ is the associated eigenvector. There are $n$ such eigenvalue pairs $(\lambda, u)$ but we are interested in computing only a few of them, say those in an interval $[\alpha, \beta]$ where $\lambda_1 \leq \alpha$, $\beta \leq \lambda_n$, and the eigenvalues of $A$ are sorted as $\lambda_1 \leq \lambda_2, \ldots, \lambda_n$. We will generally refer to the above problem as *interior eigenvalue problem*.

Applications of interior eigenvalue problems can be found in many different fields in Physics and Mechanics. For example, in electronic states of nanomaterials one is typically interested in computing a few interior eigenvalues where the exact interval of interest is determined by the properties of the material under study [3]. Another application is electronic structure calculations, and more specifically, the solution of the Dirac-Kohn-Sham equation in Density Functional Theory, where at each self-consisent field iteration one must typically compute a few hundreds of eigenvalues [2].

Albeit on the algorithmic side, one more application is the solution of indefinite linear systems with multiple right-hand sides where computation and deflation of an invariant subspace associated with the eigenvalues closest to zero can significantly accelerate the convergence of linear systems with the yet unknown right-hand sides (time-dependent problems).

The present report is the result of a semester project for the class "Applied Parallel Programming" EE5351, Spring 2014, University of Minnesota. The author would kindly like to acknowledge the support offered 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.

An independent research effort, which includes the main results of the present report, and is also accompanied with with the release of the CUDA source files, is presented in the following technical report: J. L. Aurentz, V. Kalantzis, and Y. Saad, "A GPU implementation of the filtered Lanczos procedure", Preprint, Dept. of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 2015.

The most straightforward technique to solve an interior eigenvalue problem is to utilize the Lanczos algorithm [1] and allow it to perform a large number of steps, till all eigenvalues and eigenvectors of interest are computed. For one thing, this approach will not work well because of the extensive memory requirements. The most common algorithmic approach is based on transforming the interior problem to an exterior one by means of a shift-and-invert technique. Assuming a shift $\sigma \in [\alpha, \beta]$, we then search for the largest eigenpairs of matrix $\rho(A) = (A - \sigma I)^{-1}$. The shift-and-invert technique can be extremely powerful but assumes that $A$ is given explicitly and can be factorized in a relatively low cost. The later can seriously limit the practicability of the shift-and-invert technique on large problems. The algorithmic approach followed in this paper also transforms the spectrum of $A$ but $\rho(A)$, or, more accurately, an approximation of it, is applied only through Matrix-Vector (MV) products. Furthermore, in this paper we study the acceleration of the solution of eigenvalue problems using Graphical Processing Units (GPUs).

### A. Organization of the paper

The paper is organized as follows. In section 2 we briefly describe the GPU architecture. In section 3 we describe the filtered Lanczos method for the solution of interior eigenvalue problems. In section 4 we describe in detail our GPU implementation of the filtered Lanczos method and we concentrate on the kernel MV routine. In section 5 we present numerical experiments with our implementation. Finally, in section 6 we discuss our concluding remarks.

## II. 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 (SPs) cores. Each SP performs the same instruction on its local portion of data. Using the CUDA framework, each SP executes one thread per time and in a parallel fashion. Threads are organized in blocks and each block has a pre-specified number of threads. In turn, blocks of threads form a grid. Both grid and blocks can be 1D, 2D or 3D objects.

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.

Each block of threads is assigned to an SM and is executed therein. Internally, each SM further devides the threads of a block in warps -a fixed number of threads determined in hardware- and executes one warp per time. Having a large number of threads per block, and thus a large number of warps, can hide latency since every time threads in a warp remain idle waiting for an I/O instruction, a different warp can be swapped in and executed. In practice, the number of threads that can be simultaneously supported is limited by the GPU and the same applies to the number of blocks scheduled for execution to an SM. For example, a Fermi SM can simultaneously execute up to 1536 threads, and they must be organized in at most 8 blocks of threads per SM for simultaneous scheduling.

## III. The Lanczos algorithm and the polynomial filtering technique

The Lanczos algorithm [1] is an iterative method which builds an orthonormal representation $V \in \mathbb{R}^{n\times m}$ of the Krylov subspace

$$\mathcal{K}_m(A; v) = \{v, Av, A^2v, \ldots, A^{m-1}v\}, \tag{2}$$

of a Hermitian matrix $A$, for a starting vector $v \in \mathbb{R}^n$.

At each iteration, the Lanczos algorithm performs a Matrix-Vector (MV) product between $A$ and the current Krylov vector and then orthogonalizes it against the Krylov subspace generated up to that point. In exact arithmetic, only a three-term relationship is necessary for the reorthogonalization part and thus Lanczos algorithm requires a minimal amount of storage.

Approximate eigenpairs $(\theta, x)$ of $A$ can be derived by solving the projected eigenvalue

$$T\theta = \theta y, \qquad x = Vy \tag{3}$$

where $y \in \mathbb{R}^m$ denotes an eigenvector of $T$.

**Algorithm 1** The Lanczos algorithm with polynomial transformation.

**input** : $\rho(.),\ A \in \mathbb{R}^{n\times n}, v_0 \in \mathbb{R}^n\ s.t.\ \|v_0\|_2 = 1,\ m$
**output** : $T \in \mathbb{R}^{m\times m},\ V \in \mathbb{R}^{n\times m}$

$v_{-1} = 0$

**for** $i = 0, 2, \ldots, m-1$ **do**
  $w_i = \rho(A)v_i$
  $\alpha_i =< v_i, w_i >$
  $w_i = w_i - \alpha_i v_i - \beta_i v_{i-1}$
  $\beta_{i+1} = \|w_i\|_2$
  $v_{i+1} = w_i/\beta_{i+1}$
**end for**

In theory, Lanczos algorithm provides an elegant form of tridiagonalizing a matrix operator and, given the characteristics of the underlying spectrum, the generated Krylov subspace can converge very fast towards a few extremal eigenpairs. In practice, however, where only finite precision arithmetic models exist, algorithm 1 will return a non-orthonormal basis of the Krylov subspace $\mathcal{K}_m(A; v_0)$. For that reason, some form of explicit re-orthogonalization must be included in the algorithm. The most straightforward scheme is *full reorthogonalization* in which the third line of algorithm 1 is replaced by

$$w_j = (I - VV^\top)w_j.$$

Full reorthogonalization forces Lanczos algorithm to store the entire Lanczos basis in memory and moreover, for large number of steps $m$, the reorthogonalization cost, which scales as $O(nm^2)$, will start dominating the overall cost.

### A. The polynomial filtering technique

Lanczos algorithm converges naturally to the extremal eigenpairs of $A$ and it might take a lot of steps before interior eigenpairs are well approximated (see [5] for a detailed discussion). The latter introduces not only a large number of MV products but also an excessive memory overhead.

A remedy is to transform the spectrum of $A$ by a function $\rho(.)$, so that eigenvalues in $[\alpha, \beta]$ become the largest ones in $\rho(A)$. The standard approach would be to use a shift-and-invert technique. In this context, let $\sigma$ be a real value and assume that $\alpha \leq \sigma \leq \beta$. Then the eigenvalues of $A$ inside the interval $[\alpha, \beta]$ become the dominant ones of the matrix $(A-\sigma I)^{-1}$ and applying Lanczos to that matrix should give (much) faster convergence, which, however, comes at the expense of a factorization of $A$ which is not always practical. For example, $A$ might be stemming from a 3D discretization of a domain, or might not even be given in an explicit form.

The idea behind polynomial filtering is to replace the shift-and-invert technique by a technique that also transforms the spectrum of $A$ in a more favorable one (for Lanczos) but at the same time $A$ is accessed only through MV products. Let $\rho(.)$ be a function for which $\rho(t) = 1$ if $t \in [\alpha, \beta]$ and $\rho(t) = 0$ if $t \notin [\alpha, \beta]$. The polynomial filtering technique approximates the step function $\rho(t)$ by the Chebyshev polynomial series expansion

$$\rho(t) \approx \psi_p(t) = \sum_{j=0}^{\delta} \gamma_j T_j(t)$$

where $T_j$ is the $j$-th degree Chebyshev polynomials of the first kind, $\delta$ is the degree of $\rho(t)$ and the coefficients $\gamma_j$ are defined as:

$$\gamma_j = \begin{cases} \dfrac{1}{\pi}(arccos(\alpha) - arccos(\beta)) & j = 0 \\ \dfrac{2(sin(jarccos(\alpha))) - sin(jarccos(\beta)))}{\pi j} & j > 0 \end{cases}$$

Because of the properties of Chebyshev polynomials, the definition is meaningful only when the spectrum of $A$ lies in the interval $[-1, 1]$. Since this is most probably not the case for a general symmetric matrix $A$, we can treat the general case by a simple linear transformation which maps the interval $[\lambda_{min},\ \lambda_{max}]$ to that of $[-1, 1]$. This transformation is given by the formula

$$\zeta(\lambda) = \frac{\lambda - (\lambda_n + \lambda_1)/2}{(\lambda_n - \lambda_1)/2}.$$

Although we do not know $\lambda_1$ and $\lambda_n$ a-priori, we can compute an accurate approximation by performing a few Lanczos steps.

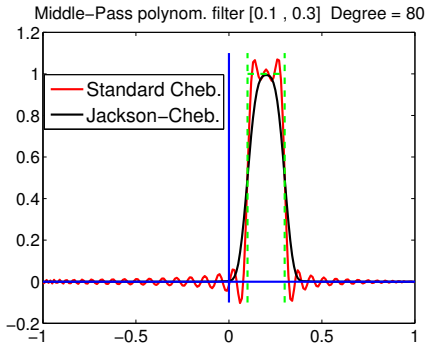


Fig. 1. Middle-Pass Chebyshev and Jackson-Chebyshev approximation of the step function in $[0.1, 0.3]$. Degree of the polynomial set to $\delta = 80$.

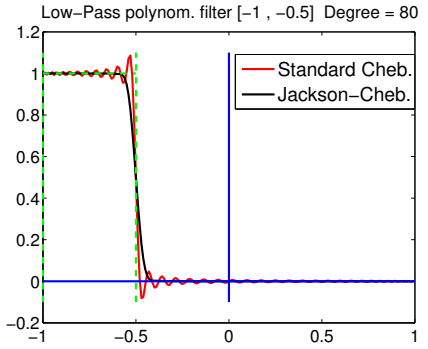


Fig. 2. Low-Pass Chebyshev and Jackson-Chebyshev approximation of the step function in $[-1.0, -0.5]$. Degree of the polynomial set to $\delta = 80$.

Polynomial approximation through Chebyshev polynomials is known to oscillate around the boundaries of the approximatio interval; these oscillations are know as Gibbs oscillations. To remedy this effect it is suggested to add damping multipliers in (III-A) so that $\rho(t)$ is approximated as

$$\rho(t) \approx \psi_p(t) = \sum_{j=0}^{p} \gamma_j g_j T_j(t)$$

where

$$g_j = \frac{\frac{1}{\mu+2} sin(\alpha_\mu) cos(j\alpha_\mu) + \frac{1}{\mu+2} cos(\alpha_\mu) sin(j\alpha_\mu)}{sin(\alpha_\mu)},$$

and

$$\alpha_\mu = \frac{\pi}{\mu+2}.$$

The multipliers $g_j$ are known as Jackson coefficients. Figures 1 and 2 show approximations of the step function in different intervals inside $[-1, 1]$ and for a fixed degree. We can easily notice the better smoothness of Jackson-Chebyshev polynomial approximation in contrast with using plain Chebyshev approximation.

Figures 3 and 4 show a comparison of the Chebyshev and Jackson-Chebyshev polynomials for a fixed degree $\delta = 100$, as well as a comparison of the Jackson-Chebyshev polynomials for two different degrees. We measure how well the polynomials suppress eigencomponents associated with eigenvalues outside the range of our interest.

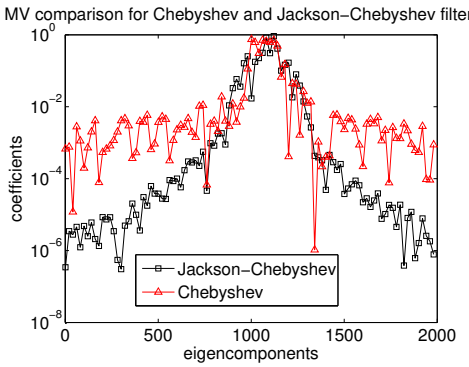


Fig. 3. Comparing Chebyshev and Jackson-Chebyshev polynomials on interval $[-0.1, 0.1]$ for matrix Trefethen2000. Degree of the polynomial set to $\delta = 100$.

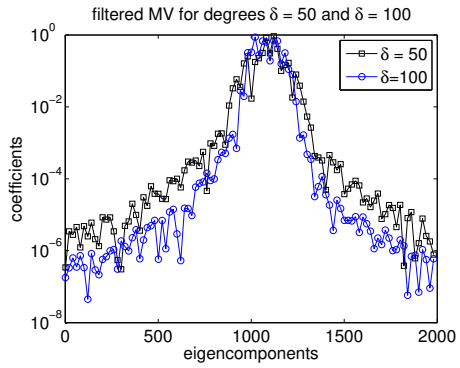


Fig. 4. Comparing Jackson-Chebyshev polynomials on interval $[-0.1, 0.1]$ for matrix Trefethen2000 and for two different degrees, $\delta = 50$ and $\delta = 100$.

## IV. Implementation on a GPU

We start this section by describing the top-level idea of our GPU implementation of the filtered Lanczos method. The main iteration loop of the procedure is controlled by the CPU and is implemented in C. The host code is responsible for initializing the CPU and GPU data structures, and also copying any initial data values from the CPU to the GPU. All numerical operations take place on the GPU and the CPU is mainly enhanced with the role of updating simple data structures and keep iterating the iteration loop of the filtered Lanczos method.

**Algorithm 2** The filtered Lanczos method from the GPU point-of-view. The implementation is linked to CUBLAS [8].

**input** : $\rho(.),\ A \in \mathbb{R}^{n \times n}, v_0 \in \mathbb{R}^n\ s.t.\ \|v_0\|_2 = 1,\ m,\ \delta,\ \gamma$
**output** : $T \in \mathbb{R}^{m \times m},\ V \in \mathbb{R}^{n \times m}$

**for** $i = 0, \ldots, m-1$ **do**
  $w_i$ = FILTERED_MV$(A,\ v_i,\ \delta,\ w,\ c,\ \gamma)$
  $\alpha_i$ = CUDOT$(n,\ w_i,\ 1,\ v_i,\ 1)$
  CUGEMV$(t,\ n,\ i+1,\ 1,\ V,\ n,\ w_i,\ 1,\ 0.0,\ u,\ 1)$
  CUGEMV$(n,\ n,\ i+1,\ -1,\ V,\ n,\ u,\ 1,\ 0.0,\ w_i,\ 1)$
  $\beta_i$ = CUDOT$(n,\ w_i,\ 1,\ w_i,\ 1)$
  DSCAL$(n,\ 1/\beta,\ w_i)$
  MEMCPY$(v_{i+1},\ w_i)$
**end for**

Algorithm 2 lists the major parts of the implementation from the CPU point-of-view. To keep presentation simple we

do not show any device memory declarations and copying procedures from the CPU to the GPU and vice versa.

Before we proceed with a more detailed description of the main modules of our implementation, we briefly describe the CSR format, a matrix format for sparse matrices. The CSR format is the most general format for storing and handling sparse matrices for it makes absolutely no assumption about the non-zero pattern of $A$. The latter, however, comes at the expense of indirect addressing during the MV product.

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{pmatrix}, a = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 4 \end{pmatrix}, ja = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \end{pmatrix} ia = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 5 \end{pmatrix} \quad (4)$$

For the sake of generality, in (4) we provide a toy example to demonstrate how a $3 \times 3$ matrix $A$ would appear in the CSR format. Vectors $a,\ ja$ store the values and corresponding column indices of $A$ while vector $ia$ shows from what value of $a$ each particular row starts. Note that $a$ and $ja$ have length equal to the number of non-zero entries of $A$ while $ia$ has length equal to $n + 1$.

### A. The Matrix-Vector product

Arguably, the filtered MV product $x = \rho(A)v$ is the most important part of our filtered Lanczos implementation. It is thus crucial to implement this part efficiently. Algorithm 3 shows the CUDA-like pseudocode of the filtered MV product $x = \rho(A)v$. The main component of algorithm 3 is the MV product $y = Av_k$ and we have to perform exactly $\delta + 1$ such MVs.

**Algorithm 3** FILTERED_MV : The filtered MV used in algorithm 2.

**input** : $A \in \mathbb{R}^{n \times n}, v_0 \in \mathbb{R}^n\ (\|v_0\|_2 = 1),\ \delta,\ w,\ c,\ \gamma$
**output** : $x \in \mathbb{R}^n$

**for** $k = 0, \ldots, \delta$ **do**
  DAXPY($n,\ \delta[i],\ v_k,\ 1,\ x,\ 1$)
  $scal = 2/w$ (or $scal = 1/w$ if $i == 0$)
  $y$ = KERNEL_MV($A, v_k$)
  $v_p = scal * (y - c * v_k) - v_m$
  $v_m = v_k$
  $v_k = v_p$
**end for**

The performance of the filtered MV product is practically dominated by the performance of the MV product with $A$, $y = Av_k$. Implementing a high-performance sparse MV routine on a GPU is a highly challenging task because of the non-optimal memory accesses by the different warps within a block. For example, consider the case where the CSR format is used and each thread computes one entry of the product $y = Av_k$. Then, the $i$'th entry of $y$, $y_i$, can be computed in trivial parallelism. However, the memory accesses are not coalesced because each thread reads elements of its assigned row serially, and consecutive threads will not load consecutive addresses. An alternative to that bottleneck is to assign more threads, e.g., an entire warp per entry of $y$, and in this case

```
__global__
void spmv_csr_scalar_kernel( const int num_rows ,
const int * ptr, const int * indices, const double *
data, const double * x, double * y) {

// An inefficient MV, one thread per row

int row = blockDim.x * blockIdx.x + threadIdx.x ;
if( row < num_rows ){

  float dot = 0;
  int row_start = ptr [ row ];
  int row_end = ptr [ row +1];
  for (int jj = row_start ; jj < row_end ; jj ++)
     dot += data [ jj ] * x[ indices [ jj ]];
  y[ row ] += dot ;
  }
}
```

TABLE I. SPARSE MV BY USING ONE THREAD PER ROW. THE ROUTINE CAN BE FOUND IN [11].

entries of $A$ are accessed contiguously and memory loads are coalesced. Later on, in the numerical experiments' section, we will demonstrate the differences of the above concepts. For an extensive discussion on sparse MV product on GPUs, see [11].

Let us now focus on the implementation of the sparse MV kernel. Table I shows an implementation of the MV kernel when using one thread per row. The algorithm is pretty simple and each thread essentially performs a sparse inner product. As noted above, however, someone should not expect this routine to be efficient except for matrices of special structure (for example diagonal matrices).

Table II shows the sparse MV kernel for the case where an entire warp, thirty-two threads, is used per row. The main difference from the single-thread kernel in I is that now threads that handle a specific row must collaborate in order to reduce the per-thread partial sums in a single value. This is achieved by using a reduction tree for each warp.

Table III is very similar to Table II, however, now each entry of the result is computed by using sixteen threads instead of thirty-two. Before we conclude this section, it is important to stretch one important detail. If $A$ is highly sparse, then if we use more threads per row we improve coalescing but, on the other hand, we might waist resources. Indeed, if $A$ has only five non-zero entries per row, then launching a kernel with thirty-two threads per row will waste twenty-seven of them which corresponds to an under-utilization of the GPU of the order of 75%! The waste takes place because only 5 threads perform a computation while the others remain idle. Thus, the number of non-zero entries per row plays an important role. What is also important is the spy pattern of matrix $A$ in general. Ideally, we would like the same exactly number of non-zero entries per row so that each row demands the same computational effort. Highly irregular non-zero patterns across the different rows of $A$ should degrade performance. We shall verify these observations in the numerical experiments' section.

```
__global__
void spmv_csr_vector_kernel(const int num_rows,
const int * ptr, const int * indices, const double
* data, const double * x, double * y) {

__shared__ float vals[];
int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
int warp_id = thread_id / 32;
int lane = thread_id & (32 - 1);
int row = warp_id;

if (row < num_rows) {

  int row_start = ptr[row];
  int row_end = ptr[row+1];

  // compute running sum per thread
  vals[threadIdx.x] = 0;

  for(int jj = row_start + lane; jj < row_end; jj += 32)
     vals[threadIdx.x] += data[jj] * x[indices[jj]];

  // parallel reduction in shared memory
  if (lane<16) vals[threadIdx.x]+=vals[threadIdx.x+16];
  if (lane<8) vals[threadIdx.x]+=vals[threadIdx.x+8];
  if (lane<4) vals[threadIdx.x]+=vals[threadIdx.x+4];
  if (lane<2) vals[threadIdx.x]+=vals[threadIdx.x+2];
  if (lane<1) vals[threadIdx.x]+=vals[threadIdx.x+1];

  // first thread writes the result
  if (lane == 0)
     y[row] += vals[threadIdx.x];
  }
}
```

TABLE II. SPARSE MV BY USING THIRTY-TWO THREADS PER ROW. THE ROUTINE CAN BE FOUND IN [11].

```
__global__
void csr_v_k(int n, int *d_ia, int *d_ja,
double *d_a, double *d_y, double *d_x) {

  int nhw = gridDim.x*BLOCKDIM/16;
  int hwid = (blockIdx.x*BLOCKDIM+threadIdx.x)/16;
  int lane = threadIdx.x & (16-1);
  int hwlane = threadIdx.x/16;

  volatile __shared__ double r[BLOCKDIM+8];
  volatile __shared__ int startend[BLOCKDIM/16][2];

  for (int row = hwid; row < n; row += nhw) {
    if (lane < 2)
      startend[hwlane][lane] = d_ia[row+lane];
    int p = startend[hwlane][0];
    int q = startend[hwlane][1];
    double sum = 0.0;
    for (int i=p+lane; i<q; i+=16) {
       sum += d_ia[i] * d_x[d_ja[i]]
    }

    r[threadIdx.x] = sum;
    r[threadIdx.x] = sum = sum + r[threadIdx.x+8];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+4];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+2];
    r[threadIdx.x] = sum = sum + r[threadIdx.x+1];
    if (lane == 0)
        d_y[row] = r[threadIdx.x];
  }
}
```

TABLE III. SPARSE MV BY USING SIXTEEN THREADS PER ROW.

### *B. Performing the inner products, vector updates and re-orthogonalization steps*

Apart the MV product, filtered Lanczos algorithm also needs to perform two DOT products (for determining $\alpha_i,\ \beta_i$) as well as one DAXPY operation; all of length $n$. Furthermore, additional cost is introduced by the full reorthogonalization routine. The cost of full reorthogonalization eventually runs at $O(nm^2)$ and far dominates that of the sparse MV product one in the latter stages of the algorithm. Thus, when using full reorthogonalization, it is crucial to select a polynomial filter $\rho(A)$ such that, relatively, only a few filtered Lanczos steps have to be performed.

In our implementation we use the CUBLAS library to perform the above operations. The CUBLAS library [8] is an implementation of BLAS (Basic Linear Algebra Subprograms) which is built on top of the NVIDIA CUDA runtime. It essentially performs basic linear algebra subroutines in the GPU unit and can potentially achieve significant speed-ups over the standard CPU-based MKL BLAS [7].

### *C. Solution of the tridiagonal eigenvalue problem*

The final step of filtered Lanczos algorithm is to solve the tridiagonal eigenvalue problem in (3) where the non-zero entries of $T$ have been filled by filtered Lanczos. Because of the tridiagonal nature of $T$, complexity of this step can be well below $O(m^3)$ where $m$ is the number of filtered Lanczos steps. In our implementation, the tridiagonal eigenvalue problem is solved on host by calling the DSTEVD routine of LAPACK library [9] (if eigenvectors are also wanted).

### *D. Computing the polynomial coefficients*

In our implementation, the coefficients of the polynomial are computed on host, before the main filtered Lanczos loop starts iterating. The coefficients are generated as described in Section II. Ideally, we would generate each single coefficient of the polynomial (of degree $\delta$) in parallel, by launching a single block with with $\delta + 1$ threads. However, we found that the gain by computing the coefficients on a GPU were minusculus, since we need only a few FLOP per coefficient and the degree of the polynomial is too low (up to $\delta = 300$ in our project) to observe any difference.

## V. NUMERICAL EXPERIMENTS

In this section we present numerical experiments with our GPU implementation of the filtered Lanczos algorithm. The experiments were conducted at the Minnesota Supercomputing Institute (MSI) on a mixed GPU and PHI HPC cluster named Cascade. For the experiments we used a Fermi M2070 GPU. Our code was written in C and compiled under the NVIDIA CUDA compiler (NVCC).

Our experimental framework focuses on a few matrices taken from University of Florida sparse matrix collection [10]. We mainly focus on benchmarking the filtered MV product since this part typically dominates the running time (in our experiments this part accounted for more than $90\%$ of the wall-clock time). All benchmarks below use a fixed filter degree $\delta = 99$.

TABLE IV. TEST MATRICES FROM THE COLLECTION IN [10].

| Matrix | $n$ | nnz | nnz/row | Application |
|---|---|---|---|---|
| ex9 | 3363 | 99471 | 29.5 | CFD |
| bcsstk11 | 3600 | 26600 | 7.4 | Structural |
| Kuu | 7102 | 340200 | 47.9 | Structural |
| benzene | 8219 | 242669 | 29.5 | Quant. Chem. |
| Press_Poisson | 14822 | 715804 | 48.1 | CFD |
| raefsky4 | 19779 | 1328611 | 67.1 | Structural |

TABLE V. TIME (IN MS) TO PERFORM 100 MV PRODUCTS FOR DIFFERENT NUMBER OF THREADS PER ROW.

| Matrix/Threads per row | 1 | 8 | 16 | 32 |
|---|---|---|---|---|
| ex9 | 46 | **3.3** | 3.7 | 4.9 |
| bcsstk11 | 5.6 | **1.4** | 2.5 | 3.8 |
| Kuu | 97.4 | 9.0 | **8.7** | 10.9 |
| benzene | 76.7 | **8.7** | 9.5 | 14.4 |
| Press_Poisson | 210.2 | 20.2 | **17.7** | 26.6 |
| raefsky4 | 416.2 | 32.7 | **27.6** | 38.1 |

Table IV shows the matrices used along their size $n$, number of non-zero entries (nnz), average number of non-zero entries per row (nnz/row) and application domain. Figures 7 and 8 sketch the non-zero pattern for two of the matrices used in our experiments.

Table V shows the elapsed time to perform 100 MV products using one, eight, sixteen and thirty-two threads per row. We can verify that using one thread per sparse inner product is significantly more worse than assigning eight or more threads per row. This difference should mostly be attributed to the waste of memory bandwidth because memory accesses are not coalesced. On the other hand, exploiting eight and sixteen threads per sparse inner-product typically performs the best since these sizes tend to minimize the number of idle threads. Exploiting eight threads per sparse inner-product should naturally lead to near-optimal results for matrix bcsstk11 since this matrix has a smooth non-zero pattern with around eight non-zero elements per row. On the other hand, matrix raefsky4 has an irregular non-zero pattern with some of its rows have many more than thirty-two elements, while most of them have almost sixteen elements. Thus, as expected, using sixteen threads per row should lead to the best performance.

Figure 5 shows the performance obtained for the GPU in terms of GFLOP/s. As was expected by the timings listed in table V, the best performance is obtained when using eight and sixteen threads. Note that the peak performance achieved is about 10 GFLOP/s which is way below the 515 GFLOP/s peak performance for double precision computations provided by NVIDIA.

Figure 6 compares the performance of the GPU and CPU architectures on the same test matrices, showing the speedup obtained when shifting from the CPU to the GPU architecture. We can clearly see that in most cases the GPU-based filtered MV outerperforms the CPU-based one by a factor of 5 to 10. Given that the filtered Lanczos method can take a number of steps $m$ in the order of hundreds or even thousands, the GPU implementation will be much faster than the CPU version; especially for polynomials of high degree $\delta$.

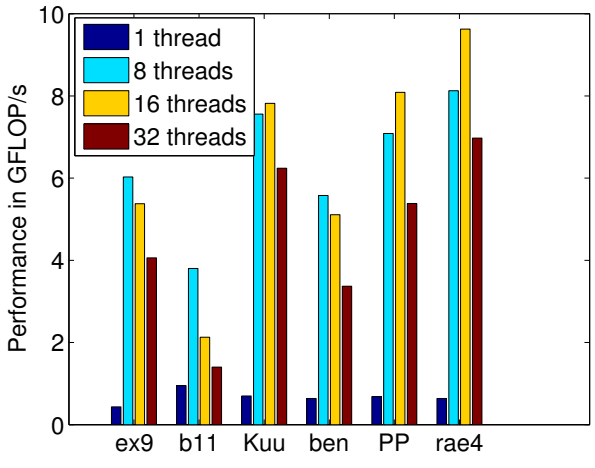


Fig. 5. Performance in terms of GFLOP/s for the results in table V.

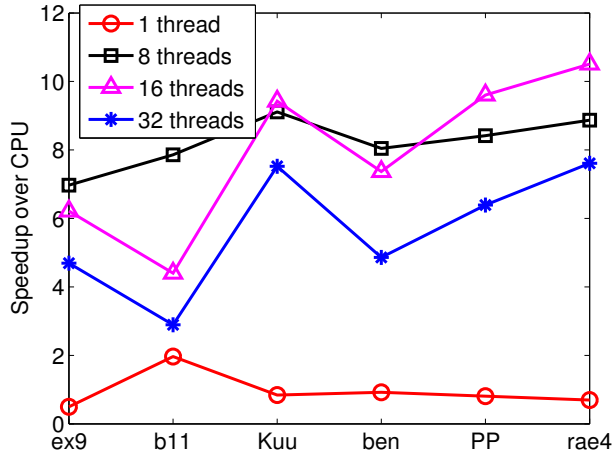


Fig. 6. Speedup over CPU performance (1,8,16 and 32 threads per row).

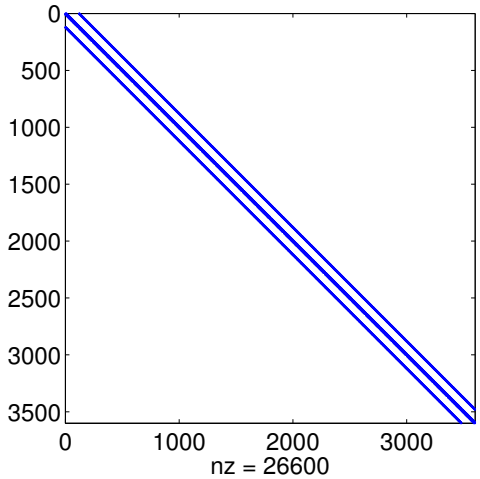


Fig. 7. The non-zero pattern of bcsstk11.

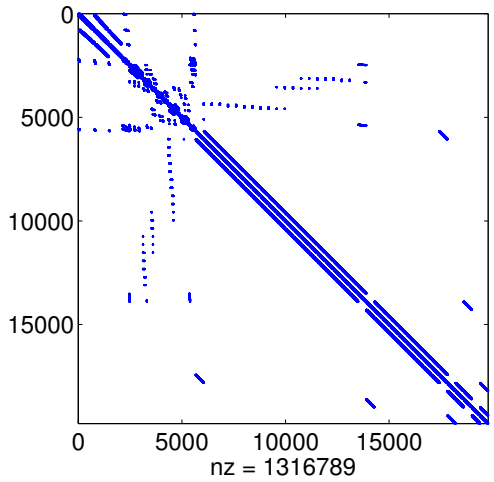


Fig. 8. The non-zero pattern of raefsky4.

## VI. Conclusion

In this report we presented a CUDA implementation of the filtered Lanczos algorithm for the solution of symmetric interior eigenvalue problems. We provided a description of our implementation and focused on the filtered MV product which dominates the overall runtime. Numerical experiments demonstrated the superiority of the GPU implementation versus the standard CPU one. In the future we hope to develop a block Lanczos filtered method while also considering other optimization aspects.

## References

[1] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. J. Res. Natl Bur. Std. 45, pp:225-282, 1950.

[2] L. Lin, S. Shao and W. E, *Efficient iterative method for solving the Dirac-Kohn-Sham density functional theory*. J. Comput. Phys 245, pp:205-217, 2013.

[3] D. Lee, T. Miyata, T. Sogabe, T. Hoshi and Shao-Liang Zhang. *An interior eigenvalue problem from electronic structure calculations*. Japan Journal of Industrial and Applied Mathematics, 30 **(3)**, pp:625-633, 2013.

[4] Haw-ren Fang and Y. Saad. *A Filtered Lanczos Procedure for Extreme and Interior Eigenvalue Problems*. SIAM J. Sci. Comput., 34 **(4)**, pp:2220-2246, 2012.

[5] Y. Saad. *On the Rates of Convergence of the Lanczos and the Block-Lanczos Methods*. SIAM Journal on Numer. Analysis, 17 **(5)**, pp:687-706, 1980.

[6] E. Di Napoli, E. Polizzi, and Y. Saad. *Efficient estimation of eigenvalue counts in an interval*. Preprint ys-2013-2, Dept. Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 2013.

[7] Intel Math Kernel Library Reference Manual, Intel MKL 11.0.

[8] CUDA CUBLAS Library, PG-05326-032_V02, August 2010.

[9] Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D. *LAPACK Users' Guide*. Third edition, SIAM, Philadelphia PA, 1999.

[10] T. Davis and Y. Hu. *The University of Florida Sparse Matrix Collection*. NA Digest, Vol. 92, No. 42, October 16, 1994.

[11] N. Bell and M. Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004, 2008.