Procedia Computer Science
Volume 29, 2014, Pages 800–810
ICCS 2014. 14th International Conference on Computational Science

# Computation on GPU of Eigenvalues and Eigenvectors of a Large Number of Small Hermitian Matrices

Alain Cosnuau[1]

ONERA
DTIM - Modeling and Information Processing 91123, Palaiseau Cedex,France
cosnuau@onera.fr

**Abstract**
This paper presents an implementation on Graphics Processing Units of QR-Householder algorithm used to find all the eigenvalues and eigenvectors of many small hermitian matrices ( double precision) in a very short time to address time constraints for Radar issues.

*Keywords:* Eigenvalues, Eigenvectors, Many Small Hermitian Matrices, QR, Householder, Parallel, GPU

## 1 Introduction

This paper presents an implementation on Graphics Processing Units of the QR-Householder method used for finding all the eigenvalues and eigenvectors of many small hermitian matrices in double precision in a very short time to deal with time constraints of Radar issues. This computation was previously done on a parallel Mercury system or on powerful PCs (multicores + OpenMP ) with times lying respectively between 1.15 and 0.50 secondes for 180 hermitian matrices (128 x 128).
The GPU libraries ( CULA, MAGMA....) for computing the QR decomposition are efficient and competitive with large matrices, at least over $1000 \times 1000$ , [12], [7], [6] but for smaller sizes, CPUs are faster.
Computing hundreds or more small matrices [2] by using libraries with hundreds of GPU calls would be inefficient. A solution is to load all the small matrices and perform extraction of all eigenvalues and eigenvectors simultaneously. The idea is both proposing a good algorithm and optimal parallelization. In this paper, algorithms and implementation are described for one matrix A, but it stands for all the loaded matrices. Presentation of GPU processors (hardware and CUDA) can be found on the Web.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014
© The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2014.05.072

## 2 The QR-Householder Algorithm for the Eigenvalue Problem

The QR algorithm consists in the factorization of a square or rectangular matrix by writing it as a product of an orthogonal matrix Q and an upper triangular matrix R. More precisely a real or complex matrix A may be written as $A = QR$ , with Q an orthogonal matrix (columns are orthogonal unit vectors) i.e $Q^*Q = I$ for the complex case , and R is an upper triangular matrix. If A is invertible, then the factorization is unique and the diagonal elements of R are non zero. This decomposition can be used to solve linear systems or solving least-squares problems. Another major use of QR is to extract eigenvalues and eigenvectors of square real or complex matrices.
There are several ways for computing the QR factorization : the Gram-Schmidt process, Householder reflections, Givens rotations. The Householder algorithm has been chosen because it is more stable than Gram-Schmidt, less expensive than Givens and good for parallelization on a GPU.
Finding eigenvalues $\lambda$ and eigenvectors $v$ of a matrix A is to solve : $Av = \lambda.v$
If A has rank n , there are n eigenvalues (some may be equal) and n eigenvectors. Successive QR decompositions are then applied to find eigenvalues and eigenvectors at the same time. The method is summarized by the following process :

$$\begin{array}{l} A_0 = A \\ For(m = 1, 2, ...) \\ \quad A_{m-1} = Q_m R_m \\ \quad A_m = R_m Q_m \\ EndFor \end{array} \tag{1}$$

This loop is over when $A_m$ is nearly diagonal. The eigenvalues are approximated by the values of the diagonal and the related eigenvectors are the columns of $Q_m$ . At each step, the eigenvalues of $A$ and $\tilde{A} = Q^h A Q$ are the same.

### 2.1 Householder Transform in the Hermitian Case

Let A be a complex hermitian matrix of size n x n such that $A^h = A$, with $A^h$ the transposed and conjugate of matrix A . Finding eigenvalues of A consists in diagonalizing A by using successive QR decompositions. To perform a QR decomposition, we apply Householder transforms. Each transform $H_j$ replaces with zeros all below diagonal elements of column $j$ . A Householder transform is an orthonormal transformation that can be written

$$H = (I - \sigma w w^h)$$

where $\sigma = 2/||w||_2^2$. These transformations, also called reflectors, have the following properties: $H^h = H$ , $H^h H = H H^h = HH = I$, and $||H||_2 = 1$.
Let

$$H = (I - \frac{2ww^h}{||w||_2^2})$$

H is hermitian and unitary. We would like a vector $w$ such that :

$$H^h x = Hx = x - \frac{2w^h x}{||w||_2^2}.w = \gamma ||x|| \vec{e_1} \tag{2}$$

with $|\gamma| = 1$ , $\vec{e_1}$ is the first column canonic vector .
Now let $w' = \frac{2w^h x}{||w||_2^2} w = \alpha w$, from equation (2) :

$$x \pm e^{i\theta_1}||x||\vec{e_1} = \alpha w = w'$$

, then use of $w'$ instead of $w$ in the Householder reflection gives :

$$H'x = x - \frac{2{w'}^h x}{||w'||_2^2}.w' = Hx$$

So we can take $w = x \pm e^{i\theta_1}||x||\vec{e_1}$ and to prevent from accuracy problems, we choose the plus sign, finally with vector notation :

$$\vec{w} = \vec{x} + e^{i\theta_1}||x||\vec{e_1} \tag{3}$$

It is clear that $||w||^2 = 2||x||.(||x|| + |x_1|)$
The vector $w$ is called the Householder vector in the following. Multiplying $H$ by a part of a column vector $x$ gives :
$Hx = -e^{i\theta_1}||x||e_1 = \gamma||x||\vec{e_1}$ so we have $\gamma \ = \ - e^{i\theta_1} \ = \ - x_1/|x_1|$

$$Hx = x - \frac{2w^h x}{||w||_2^2}.w = \gamma||x||\vec{e_1} \tag{4}$$

This choice of Householder matrix $H = (I - \sigma w w^h)$ with $\sigma = 2/||w||_2^2$ has two advantages : $H$ is Hermitian and $\sigma$ is real. The second point is important because it means a smaller number of operations than with a complex value. Other choices are possible with a complex $\sigma$ for instance [8], [1]. So applying Householder transforms successively on columns of matrix A by zeroing under diagonal values of each column, we get :

$H_{n-1}.H_{n-2} \ ..... \ H_0.A \ = \ Q^h.A \ = \ R$

Now we diagonalize $A$ by the algorithm previously described with successive QR transforms in loop 1 .
The real eigenvalues of matrix A are approximated by the diagonal matrix D:

${Q_{m-1}}^h ... {Q_1}^h {Q_0}^h \ A \ Q_0 Q_1 ... Q_{m-1} = D$

This strategy of diagonalization has some drawbacks. Many operations are needed at each column-step $= 0 \ to \ n-1$ and the convergence is generally slow which means that the number of global iterations $m$ can be large and is not known in advance. The usual strategy is to transform the hermitian matrix A(n,n) into a hermitian tridiagonal matrix T in one step. Then $T$ is transformed into real symmetric matrix and QR iterations are applied on it for the final diagonalization. This way, each iteration deals with small columns with 2 real terms instead of $n-k$ (at step k) and convergence to diagonal is fast.

## 2.2 The Householder QR and Tridiagonalization

In the previous section the Householder transforms were applied to each column zeroing the values under the diagonal. Now the Householder transform on each column is used to transform the values to zero under the first subdiagonal. With this trick, orthogonal similarity on a

general matrix A gives a Hessenberg matrix ie an upper triangular matrix plus a subdiagonal. In the hermitian case for symmetric reasons, it gives a tridiagonal hermitian matrix.
Let $u = A_{k+1:n-1,\,k}$ a vector extracted from column $k$ of $A$, for $k = 0 \;\; to \;\; n-3$ and now form and apply successively on each of the first $n-2$ columns householder reflections of type $H = (I - \sigma ww^h)$ then $A_k = H_k^h.A_{k-1}.H_k$ , for each column $k$ .
Let $Q = H_0.H_1.H_2 \cdots H_{n-3}$ , so $T = Q^hAQ$ is hessenberg and tridiagonal if $A$ is hermitian.

$$\mathrm{T} = \begin{pmatrix} a_0 & \bar{b}_0 & & & \\ b_0 & a_1 & \bar{b}_1 & & \\ & b_1 & \ddots & \ddots & \\ & & \ddots & \ddots & \bar{b}_{n-2} \\ & & & b_{n-2} & a_{n-1} \end{pmatrix}$$

This tridiagonal matrix is computed through $k = 0, n-3$ iterations :
$A_k = H_k^h.A_{k-1}.H_k = (I - \sigma w_k {w_k}^h)A(I - \sigma w_k {w_k}^h)$
with the following tools :

$$Hx = x - \frac{2w^hx}{||w||_2^2}.w = \gamma ||x|| \vec{e_1} \tag{5}$$

$\gamma = -e^{i\theta_1}$ and Householder vector :

$$\vec{w} = \vec{x} + e^{i\theta_1}||x||\vec{e_1} \tag{6}$$

such that : $||w||^2 = 2||x||.(||x|| + |x_1|)$ and then with $\sigma$ for computing $H$ :

$\sigma = 2/||w||^2 = 1/(||x||.(||x|| + |x_1|))$

Notation : vector $\vec{e_1}$ stands for the canonical vector of the column $k$ beginning on row $k+1$. In practice at each step $k$ the $A_k = (I - \sigma w_k {w_k}^h)A(I - \sigma w_k {w_k}^h)$ are not computed in this form but in a reduced formula with which only one matrix vector computation is done instead of two. For sake of generality $\sigma$ is assumed to be complex and the Householder vector is noted $u$ in the following. By developing and changing variables we can write :
$\tilde{A} = H^h.A.H = (I - \bar{\sigma}uu^h).A.(I - \sigma uu^h)$ in a reduced form at step $k = 1, n-2$ :
1. $v = -\sigma Au$
2. $\alpha = -(1/2)\sigma v^hu$
3. $w = v + \alpha u$
4. $\tilde{A} = A + wu^h + uw^h$

At each step $k$, $A_{kk} = a_k$ and $A_{k,k+1} = \bar{b}_k$ are stored into global memory of the GPU for building the tridiagonal hermitian matrix $T$. The former left and right matrix vector products are now replaced by only one matrix vector product for each column k (point 1.). It can be proposed some remarks to optimize computations for the implementation on GPU. The threads are organized into a 1D grid of 1D blocks. One block of threads is in charge of one square matrix, so the number of blocks is equal to the number of matrices. In a block, the number of threads is the matrix size (number of lines or columns). So the same computations can be done in this way at step $k$ for each matrix in each independent block. The computation on $A_k$ became smaller and smaller $(n-k, n-k)$ , it means that less and less threads are really working, memory accesses (not perfectly coalesced ) are also impacted.

Point 1. : the ordinary way of computing $\vec{v} = A_k.\vec{u}_k$ for each row $i$ of $A_k$ is of the form : $v(i) \ = \ \sum_{j=k}^{n} a_{ij}.u_j$

The consecutive GPU threads access to consecutive matrix $A$ row data but in a non-coalesced way (first address of the transaction) and sommation is a sum-reduction, which is fast but not optimal on GPU. Another approch is possible to improve performance. The matrix at each step $k$ is hermitian and assumed to be fully stored . So to perform matrix multiplication a Saxpy strategy is prefered : $A_k.\vec{u}_k \ = \ \sum_{j=k}^{n} \vec{A}_j.u_j$ with $\vec{A}_j$ a column vector of hermitian matrix $A_k$ . Then for each column $j$ , $\vec{A}_{colj} = \vec{A}^h_{rowj}$ , $h$ for transposed and conjugate, and consequently $A_k u_k = \sum_{j=k}^{n} \vec{A}^h_{rowj}.u_j$ .
This means that the threads read successive rows of $A_k$ and in each thread the associated conjugate term of the row $j$ is multiplied by the same $u_j$ and added to the next conjugate row $j+1$ of $A$ multiplied by one $u_{j+1}$ completely in parallel in each thread and so on.

Point 2. is done by a sum-reduction with data in shared memory [5].
The last point $\tilde{A} \ = \ A \ + \ wu^h \ + \ uw^h$ can be also efficiently computed row after row at step $k$ :
$\tilde{A} \ = \ A \ + \ \vec{w}\vec{u}^h \ + \ \vec{u}\vec{w}^h$ becomes $\tilde{A}_{rowi} \ = \ A_{rowi} \ + \ w_i\vec{u}^h \ + \ u_i\vec{w}^h$. Each $A_{rowi}$ is read in global memory and $\vec{u}$ ,$\vec{w}$ are assumed to be stored in shared memory so we can expect a good performance for this point.
The access for each $k$ step to a row of $A$ at $A(k,k)$ is not perfectly coalesced .

Now matrix $Q$ must be updated. Matrix $Q^h$ is the result of $H_{n-3}$ ..... $H_1 \ H_0 \ = \ Q^h$ which is performed by $Q_{k+1} \ = \ H_k {Q_k}^h$ of the form $(I - \sigma u u^h)Q^h \ = \ (Q^h - \sigma u u^h.Q^h)$ at step $k$ with $Q_0 \ = \ H_0.I$ .
The main computation is $(\sigma u)u^h.Q^h$ . At each step $k$ the matrix $Q^h$ is assumed to be in the global memory of the GPU and Householder vector $u$ in the shared memory.

$$u^h.Q^h = \begin{pmatrix} 0 & \dots & 0 & \overline{u}_{k+1} & \overline{u}_{k+2} & \dots & \overline{u}_{n-1} \end{pmatrix} \times \begin{pmatrix} q_{01} & q_{02} & q_{03} & \cdots & q_{0,n-1} \\ q_{11} & q_{12} & q_{13} & \cdots & q_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ q_{k1} & q_{k2} & q_{k3} & \cdots & q_{k,n-1} \\ q_{k+1,1} & q_{k+1,2} & q_{k+1,3} & \cdots & q_{k+1,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ q_{n-1,1} & q_{n-1,2} & q_{n-1,3} & \cdots & q_{n-1,n-1} \end{pmatrix} \tag{7}$$

In a block of threads, each thread makes successive (vertical) additions :

$$\begin{array}{ccccc} thread0 & thread1 & thread2 & \ldots\ldots & thread n-1 \\ \overline{u}_{k+1}\cdot q_{k+1,0} & \overline{u}_{k+1}\cdot q_{k+1,1} & \overline{u}_{k+1}\cdot q_{k+1,2} & \ldots\ldots & \overline{u}_{k+1}\cdot q_{k+1,n-1} \\ + & + & + & \ldots\ldots & + \\ \overline{u}_{k+2}\cdot q_{k+2,0} & \overline{u}_{k+2}\cdot q_{k+2,1} & \overline{u}_{k+2}\cdot q_{k+2,2} & \ldots\ldots & \overline{u}_{k+2}\cdot q_{k+2,n-1} \\ \ldots\ldots\ldots & & \ldots\ldots\ldots & & \ldots\ldots\ldots \\ + & + & + & \ldots\ldots & + \\ \overline{u}_{n-1}\cdot q_{n-1,0} & \overline{u}_{n-1}\cdot q_{n-1,1} & \overline{u}_{n-1}\cdot q_{n-1,2} & \ldots\ldots & \overline{u}_{n-1}\cdot q_{n-1,n-1} \end{array}$$

The $n-1-k$ rows of $Q$ are read in a coalesced way and $u$ components are in shared memory. The result of this multiplication is $\vec{p}$, a row vector of size $n$ stored in shared memory. Then the last product is $(\sigma\vec{u}).\vec{p^t}$ which is a matrix $n \times n$ which rows $i$ are equal to : $(\sigma u_i).\vec{p}$.
This computation is fast, each thread $j = 0, n-1$ in a block just does $(\sigma u_i).p_j$ for $i = k+1, n-1$ , then each row $i$ is added to the corresponding $i = k+1, n-1$ row of the previous matrix $Q^h$.
After the reduction computation into $n-2$ steps, the resulting matrix $A$ is a tridiagonal and hermitian matrix $T$. Just the diagonal and the overdiagonal of $T$ are kept and stored into 2 vectors one real and one complex in shared memory , plus the whole complex matrix $Q$.
For diminishing number of computations and memory accesses in the diagonalization step, we would like to work with a real symmetric tridiagonal matrix instead of a hermitian one. This could have been done directly in the tridiagonalization part by using special $\sigma$ complex values but it would have produced much more computations. To address reduction of computation cost, we multiply hermitian $T$ left and right by a special diagonal matrix $S$ . This way $T$ is transformed into a real symmetric tridiagonal matrix.
Let the diagonal matrix $S^h$ written on a row :

$$S^h = (1. , \frac{\bar{b}_1}{|b_1|} , \frac{\bar{b}_1\bar{b}_2}{|b_1||b_2|} , \frac{\bar{b}_1\bar{b}_2\bar{b}_3}{|b_1||b_2||b_3|} , \ldots, \frac{\bar{b}_1\bar{b}_2 \ldots \bar{b}_{n-2}\bar{b}_{n-1}}{|b_1||b_2| \ldots |b_{n-2}||b_{n-1}|})$$

$$\text{Then } S^h.T.S = \begin{pmatrix} a_1 & |b_1| & & & \\ |b_1| & a_2 & |b_2| & & \\ & |b_2| & \ddots & \ddots & \\ & & \ddots & \ddots & |b_{n-1}| \\ & & & |b_{n-1}| & a_n \end{pmatrix}$$

The computation of $S^h.T.S$ is not done explicitly in the GPU since the result is known in advance. It is just necessary to replace in $T$ all the terms $\bar{b}_i$ stored in the upper diagonal by their modules $|b_i|$ . Then the $Q^h$ matrix is to be updated and becomes $S^h.Q^h$ which is done row after row in a coalesced way in parallel. Then the simultaneous diagonalization of all the real symmetric $T$ matrices is carried out by a separate kernel (a GPU routine).

## 3 Diagonalization of the Real Tridiagonal Symmetric Matrix

This part consists in computing the eigenvalues and eigenvectors of the real symmetric tridiagonal matrices $T$ . We are also using QR-Householder method to diagonalize all $T$ matrices iteratively. The Householder vectors have now only 2 components and data are real. In addition, a new choice for Householder vector is done to reduce computation of the diagonalization process, [4].
In this process, a Wilkinson-type shift, adapted for each $T$ matrix, is applied to accelerate the convergence to diagonal form .
Like in the previous tridiagonalization kernel, a matrix of size $n$ is handled by a 1D block of $n$ threads and the grid contains as many blocks as matrices.

## 3.1 Diagonalization Scheme

This algorithm is very sequential. So the QR-Housholder method is computed by only one thread of each block for zeroing symmetric diagonals. Updates of $Q^h$ containing the eigenvectors are much more time consuming than diagonalization itself but involve all the threads of the block in parallel.
The process is similar to the tridiagonalization, we apply successively on each of the first $n-1$ two terms columns Householder reflections of type $U = (I - \sigma w w^t)$ then $T_k \ = \ U_k^t.T_{k-1}.U_k$ , for each column $k = 0, \ n-2$ with the C language notation .
Let $Q \ = \ U_0.U_1.\cdots U_{n-2}$ . Now successive $Q$ global transforms are applied on the whole matrix $T$ , we get a nearly diagonal matrix $D$ :

$T_m \ = \ {Q_m}^t T_{m-1} Q_m$ then $D \ = \ \lim\limits_{m \to \infty} T_m$
The iterations $m$ are stopped when the desired precision is reached, $m$ depends on the size and the nature of $T$ .
Let $T$ the matrix we get after the tridiagonalization step. At every global $m$ iteration this tridiagonal matrix is computed through $k = 0, n-2$ steps on columns :

$T_k \ = \ U_k^t.T_{k-1}.U_k = (I - \sigma w_k {w_k}^t) T_{k-1} (I - \sigma w_k {w_k}^t)$ , all $T_k$ are symmetric and tridiagonal.
Exactly in the same way as in the tridiagonalization, this computation is carried out by the same reduced process in 4 points, already described, but with different $\sigma$ and Householder vector:

$u \ = \ (1.0 \ , \ \frac{t_{k,k+1}}{\nu + x_1}, 0, ...0)$ , $\sigma = \frac{\nu + x_1}{\nu}$ and $\nu = sign(x_1)||x||$ .

At every step $k$ the complex matrix $Q^h$ is updated , this operation is done by all the threads of the block.

### 3.1.1 Updating $Q^h$

In this section the current update at any step $k$ is considered with simplified notations :

$$Q_{k+1} \ = \ H_k {Q_k}^h \ = \ (I - \sigma u u^h) Q^h \tag{8}$$

$\vec{u^t} \ = \ (0, 0, 1.0, u_1, 0, 0, 0)$ is the real Householder vector at step $k$ .
Matrix $Q_k$ is complex and stored in global memory , vector $\vec{u}$ is defined by just one value $u_1$, stored in shared memory. The first part of updating operation is :

$$u^t.Q^h = \begin{pmatrix} 0 & \dots & 0 & 1.0 & u_1 & \dots & 0 \end{pmatrix} \times \begin{pmatrix} q_{00} & q_{01} & q_{03} & \cdots & q_{0,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ q_{k0} & q_{k1} & q_{k2} & \cdots & q_{k,n-1} \\ q_{k+1,0} & q_{k+1,1} & q_{k+1,2} & \cdots & q_{k+1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ q_{n-1,0} & q_{n-1,1} & q_{n-1,2} & \cdots & q_{n-1,n-1} \end{pmatrix} \tag{9}$$

$$u^t.Q^h = (\sigma(q_0 + u_1.q_1), \dots, \sigma(q_k + u_1.q_{k+1}), \dots, \sigma(q_{n-2} + u_1.q_{n-1})) \tag{10}$$

This is done in parallel in the GPU, each thread fetches 2 elements of rows $q_k$ and $q_{k+1}$ in $Q^h$ and compute $S$ in the local thread $j$, $S = \sigma(q_{k,j} + u_1.q_{k+1,j})$ . $\sigma$ is in the shared memory of the block and $S$ in a register. Next,the 2 rows $q_k$ and $q_{k+1}$ are updated with $(Q^h - \sigma u.(u^t Q^h))$ : $q_k = q_k - S$ and $q_{k+1} = q_{k+1} - u_1.S$ , with a $S$ related to each thread.
Updating of $Q^h$ uses a prefetching technique to access rows, one at a time, in the global memory and also to take advantage of some overlapping of computations and global memory accesses.

## 3.2 Shifted algorithm

Some eigenvalues can be near from each other, then diagonalization can take a long time. To reduce the number of iterations, we apply a Wilkinson's shift [13] .
At each global iteration $m$, a shift is used and if upper diagonal term of $T$, $t_{n-2,n-1}$ is sufficiently small, $t_{n-1,n-1}$ is stored as eigenvalue $\lambda_{n-1}$. The process goes on with a reduced matrix $T$ without its last row/column. Dimension reduction of $T$ also contributes to lower the computation load of the diagonalization. For updating $Q^h$ there is no change, all the threads are involved.
The Wilkinson's shift consists in approximating $\lambda_{n-1}$ by the last square matrix eigenvalue which is the closest to $t_{n-1,n-1}$ at iteration $m$ .
Let this last square $\begin{bmatrix} t_{n-2,n-2} & t_{n-2,n-1} \\ t_{n-2,n-1} & t_{n-1,n-1} \end{bmatrix}$ the shift $\mu$ is :

$$\mu \; = \; t_{n-1,n-1} \; - \; \frac{sign(\delta).t^2{}_{n-2,n-1}}{|\delta| + \sqrt{\delta^2 + t^2{}_{n-2,n-1}}}$$

with $\delta \; = \; (t_{n-2,n-2} \; - \; t_{n-1,n-1})/2$ , if $\delta \; = \; 0$ then $\mu \; = \; t_{n-1,n-1} - |t_{n-2,n-1}|$
Let $\mu$ denote the eigenvalue estimate chosen at step $m$ of the QR algorithm. We built a new sequence with the recurrence :

```
For   m=0,1,2, ......
 $T_m - \mu I = Q_m.R_m$          (11)
$T_{m+1} \; = \; R_m.Q_m + \mu I$
Endfor
```

All matrices $T_{m+1}$ have same the eigenvalues and eigenvectors as the original tridiagonal matrix $T$ . The acceleration impact of the shift takes place in equation 11 .

# 4 Results

This code is implemented in C , double precision, with two CUDA/C kernels running on Fermi M2070, ECC=1. Here the relative precision obtained is between $10^{-6}$ and $10^{-9}$ for eigenvalues and for each components of eigenvectors between $10^{-11}$ and $10^{-13}$ but it can be more if needed. The measures are done by comparing $A.Q$ and $Q.D$ , $D$ is the diagonal matrix containing eigenvalues and $Q$ columns are the eigenvectors. Orthogonality of eigenvectors is extremely good. All times are in milliseconds .

| Matrix Size | 64 | 128 | 256 | 512 |
|---|---|---|---|---|
| Iterations | 88 | 177 | 356 | 707 |
| Tridiag. | 3.4 | 13.2 | 66.0 | 385.7 |
| GFlops | 1.76 | 3.6 | 5.9 | 8.1 |
| Diag. | 9.8 | 38.3 | 162.0 | 727.0 |
| GFlops | 0.46 | 0.4 | 1.15 | 2.9 |
| Final Time | 13.2 | 51.5 | 228.0 | 1113.0 |
| GFlops | 0.56 | 1.26 | 2.5 | 4.7 |
| Time Ratios | 1.0 | 3.9 | 17.2 | 84.3 |

Table 1. One test matrix

From column 64 to 512, the Tridiagonalization computations are theoretically multiplied a factor of $8^3$, in fact we get 385.7/3.4 =113.4. As to Diagonalization, the number of iterations increases from 88 to 707, it means an expected factor $8^2 \times (707/88) = 514$ , in fact we get 727/9.8 =74.2 . Globally the ratios show that the execution times increase by just 84.3 for a 512-matrix compared to a 64-matrix.
Table 2 shows results with 180 identical test matrices :

| Matrix Size | 64 | 128 | 256 | 512 |
|---|---|---|---|---|
| Iterations | 88 | 177 | 356 | 707 |
| Tridiag. | 18.1 | 144.6 | 1143.0 | 9093.0 |
| GFlops | 59.3 | 60.2 | 61.2 | 61.8 |
| Diag. | 26.9 | 199.3 | 836.0 | 6090.0 |
| GFlops | 9.9 | 25.2 | 40.4 | 62.7 |
| Final Time | 45.1 | 263.9 | 1979.8 | 15183.5 |
| GFlops | 29.8 | 44.3 | 52.4 | 62.2 |
| Time Ratios | 1.0 | 5.85 | 43.9 | 336.6 |

Table 2. 180 test matrices

Table 3, gathering Table 1 and 2, reports global time comparison between 180 matrices and one matrix. It shows that the simultaneous parallelization on the 180 matrices compared to one matrix is efficient. Computation of 180 matrices compared to one gives a ratio 45.1/13.2 = 3.4 with matrices (64x64) and 15183.5/1113.0 = 13.6 with matrices (512x512 ) instead of 180.

| Matrix Size | 64 | 128 | 256 | 512 |
|---|---|---|---|---|
| Iterations | 88 | 177 | 356 | 707 |
| 1 Matr. Time | 13.2 | 51.5 | 228.0 | 1113.0 |
| 180 Matr. Time | 45.1 | 263.9 | 1979.8 | 15183.5 |
| Time Ratios | 3.4 | 5.1 | 8.6 | 13.6 |

Table 3. Comparisons between 1 and 180 matrices

Now the application matrix is 128x128 coming from an actual physics experiment. On Fermi (ECC=0) for 180 matrices, the times and average performances are :

```
Tridiagonalization                : 146.66 (ms) ,  59.35 GFlops
Diagonalization ( 147 iterations ) :  93.01 (ms) ,  57.24 GFlops
Global result                     : 239.67 (ms) ,  58.53 GFlops
```

On K20 (SMX=13) (ECC=1) for the same set of matrices :

```
Tridiagonalization                : 116.35 (ms) ,  74.81 GFlops
Diagonalization ( 147 iterations ) :  77.19 (ms) ,  68.96 GFlops
Global result                     : 193.54 (ms) ,  72.48 GFlops
```

No other paper was found on this topic, so "comparisons" are made with results in [11] (Fig.4) for one 1024x1024 matrix and just one QR decomposition double precision which reaches less than 50 GFlops on Fermi C2050. The paper [2] gives results but just for QR decomposition of hundreds or thousands of small matrices in simple precision.
Comparisons with bisection method for finding eigenvalues can be found in [10] which gives 1.08 ms for computing eigenvalues on a 120-sized tridiagonal matrix. In our case it is 38.3ms for one 128-sized tridiagonal matrix for finding eigenvalues AND eigenvectors, and 199.3 ms for 180 matrices (Table 2.) which is equivalent to 1.107 ms per matrix. An interesting paper due to Ch. Lessig is in [9] for bisection with larger matrices.
Some other results using Lapack and ScaLapack can be found in [3]. The performances on one small matrix are far less efficient than on GPU, less than 0.15 GFlops for a 128-size matrix. The average performance per matrix of our GPU implementation is 58.53 GFlops/180 = 0.325 GFlops on Fermi.

## 5 Conclusion

The goal of finding all the eigenvalues and eigenvectors of many small hermitian matrices in double precision in less than one second is achieved. The code can handle matrices of sizes 128, 256 or 512 and 1024 (with small modifications). This code uses most available optimizations possibilities of the GPUs : massive use of registers, shared memory , prefetching accesses to global memory. Special attention was paid to theoretical aspects of the algorithm before parallelization. The QR-Householder algorithm, among several others has been chosen. Strong effort at all steps is carried out to decrease the number of operations. In other words we prefered to analyse the algorithm very carefully by computing just needed values instead of using global style routines like matrix-vector or matrix-matrix which would have computed values to be known as zero for instance.
Future work will be to reduce the computing time by distributing matrices on several GPUs and improve diagonalization step with parallel Givens method.

## References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users Guide*. SIAM,Philadelphia, 3. edition, 1999.

[2] Michael J. Anderson, David Sheffield, and Kurt Keutzer. A predictive model for solving small linear algebra problems in gpu registers.

[3] Grey Ballard and Mehrzad Tartibi. Symmetric eigenvalue problem: Tridiagonal reduction, May 18, 2009. CS 267 Parallel Algorithms.

[4] Dr. George W. Benthien. Notes on numerical linear algebra, 2006.

[5] Mark Harris. Sc07 high performance computing with cuda, optimizing cuda. Technical report, NVIDIA Developer Technology.

[6] Andrew Kerr, Dan Campbell, and Mark Richards. Qr decomposition on gpus.

[7] Jakub Kurzak, Rajib Nath, Peng Du, and Jack Dongarra. An implementation of the tile qr factorization for a gpu and multiple cpus. Technical report, University of Tennessee.

[8] R.B. Lehoucq. The computation of elementary unitary matrices, 1995.

[9] Christian Lessig. Eigenvalue computation with cuda.

[10] Ion LUNGU, Alexandru PRJAN, and Dana-Mihaela PETROANU. Optimizing the computation of eigenvalues using graphics processing units. *U.P.B. Sci. Bull., Series A*, 74(3), 2012.

[11] Stanimire Tomov Rajib Nath and Jack Dongarra. An improved magma gemm for fermi gpus. Technical report, University of Tennessee (USA), Oak Ridge National Laboratory (USA), University of Manchester (UK), July 20, 2010.

[12] Raffaele Solc, Thomas C. Schulthess, Azzam Haidar, Stanimire Tomov, Ichitaro Yamazaki, and Jack Dongarra. A hybrid hermitian general eigenvalue solver.

[13] J.H. Wilkinson. The algebraic eigenvalue problem, 1965.