Eigenvalue Computation with CUDA

Christian Lessig

October 2007

Document Change History

Version	Date	Responsible	Reason for Change
	August, 2007	Christian Lessig	Initial release

Abstract

The computation of all or a subset of all eigenvalues is an important problem in linear algebra, statistics, physics, and many other fields. This report describes the implementation of a bisection algorithm for the computation of all eigenvalues of a tridiagonal symmetric matrix of arbitrary size with CUDA.

Background

In this section we will establish our notation and provide the mathematical background for the remainder of the report. On a first read some of the presented material might prove difficult for the mathematically less inclined reader. Most theorems can, however, be employed as black box results and their value becomes apparent when they are used in the next sections. Readers interested in a more thorough discussion of eigen analysis algorithms are referred, for example, to the book by Parlett [4] or the thesis by Dhillon [13].

Notation

In this report we will use Householder's notation. Scalars are denoted by roman or greek lowercase letters such as a, b, and c and α, β, and γ. Lowercase bold letters such as $\mathbf{a} = \{a_i\}_{i=1}^n$, $\mathbf{b} = \{b_i\}_{i=1}^n$, and $\mathbf{c} = \{c_i\}_{i=1}^n$ are used to represent vectors. If not mentioned otherwise, all vectors are assumed to be column vectors. Matrices are denoted by uppercase bold roman letters such as $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{T}$. Matrix elements are scalars a_{ij} indexed with two indices i and j, where i denotes the row index and j the column index. A matrix $\mathbf{A} \in \Re^{n \times n}$ can thus be written as

$$\mathbf{A} = \begin{pmatrix} a_{11} & .. & .. & a_{1n} \\ .. & & & .. \\ & & & \\ .. & & & .. \\ a_{n1} & .. & .. & a_{nn} \end{pmatrix}.$$

The (main) diagonal of a matrix $\mathbf{A} \in \Re^{m \times n}$ is formed by the elements a_{ii} with $i = 1...n$. For brevity we will usually omit the second index and consider the diagonal as a vector $\mathbf{a}$ of length n whose elements are $a_i \equiv a_{ii}$. The elements a_{ij} of $\mathbf{A}$ with $i = j - k$ and $k \in \{1,...,n-1\}$ form the k-th upper diagonal of $\mathbf{A}$, and the element a_{ij} with $i - k = j$ the k-th lower diagonal. Analogous to the main diagonal, the k-th upper and lower diagonal can be represented by vectors of length $n-k$. A matrix that has non-zero elements only on the main diagonal is called a diagonal matrix. An important instance is the identity matrix where all diagonal elements are 1.

A square matrix $\mathbf{A} \in \Re^{n \times n}$ is symmetric if $a_{ij} = a_{ji}$ for all i and j. We will usually use a symmetric letter such as $\mathbf{A}$ or $\mathbf{T}$ to denote such a matrix. The transpose of a matrix $\mathbf{A} \in \Re^{m \times n}$ will be denoted with $\mathbf{A}^T$. We will sometimes also transpose vectors. In this case we treat a vector of length n as $n \times 1$ matrix.

Of particular interest for this report are tridiagonal symmetric matrices where only the main dsiagonal and the first lower and upper diagonal are non-zero

$$\mathbf{T} = \begin{pmatrix} a_{11} & b_{12} & 0 & .. & 0 \\ b_{21} & a_{22} & b_{23} & 0 & .. \\ 0 & b_{32} & .. & .. & 0 \\ .. & 0 & .. & .. & b_{(n-1)n} \\ 0 & .. & 0 & b_{n(n-1)} & a_{nn} \end{pmatrix}$$

with $b_{ij} = b_{ji}$. If not mentioned otherwise, we will use $\mathbf{T}$ to denote a tridiagonal symmetric matrix.

Eigenvalues and Eigenvectors

Definition 1: Let $\mathbf{A} \in \Re^{n \times n}$ be a square matrix. An *eigenvalue* λ of $\mathbf{A}$ is a scalar satisfying

$$\mathbf{Au} = \lambda \mathbf{u} .$$

The vector $\mathbf{u} \neq \mathbf{0}$ is a (right) *eigenvector* of $\mathbf{A}$. The *spectrum* $\lambda(\mathbf{A})$ of $\mathbf{A}$ is formed by the set of all eigenvalues of $\mathbf{A}$

$$\lambda(\mathbf{A}) = \{\lambda_i\}.$$

If $\mathbf{A}$ is non-degenerated then the cardinality of $\lambda(\mathbf{A})$ is n.

Corollary 1: The spectrum of a diagonal matrix $\mathbf{D}$ with non-zero elements d_i is

$$\lambda(\mathbf{D}) = \{d_i\}_{i=1}^{n} .$$

If we arrange the spectrum of a general square matrix $\mathbf{A}$ in a diagonal matrix $\mathbf{\Lambda}$, and form a square matrix $\mathbf{U}$ so that their columns are the eigenvectors, then the eigen decomposition of $\mathbf{A}$ can be written as

$$\mathbf{AU} = \mathbf{U\Lambda} .$$

The eigenvalues λ_i are also the roots of the characteristic polynomial

$$\det(\mathbf{A} - \lambda_i \mathbf{I}) = 0 .$$

Eigenvalues can therefore in general be real or complex. For a symmetric matrix the eigenvalues are guaranteed to be real [1, pp. 393].

Gerschgorin Circle Theorem

Theorem 1. (Gerschgorin Circle Theorem) Let $\mathbf{A} \in \Re^{n \times n}$ be symmetric and $\mathbf{Q} \in \Re^{n \times n}$ be orthogonal. If $\mathbf{Q}^T \mathbf{AQ} = \mathbf{D} + \mathbf{F}$, where $\mathbf{D}$ is diagonal and $\mathbf{F}$ has zero diagonal entries, then

$$\lambda(\mathbf{A}) \subseteq \bigcup_{i=1}^{n} [d_i - r_i, d_i + r_i]$$

with $r_i = \sum_{j=1}^{n} \left| f_{ij} \right|$ for $i = 1 \ldots n$.

Proof: See [1, pp. 395]. □

If we let $\mathbf{Q}$ be the (trivially orthogonal) identity matrix then $\mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{D} + \mathbf{F}$ is always satisfied. In practice one wants to employ a matrix $\mathbf{Q}$ such that $\mathbf{Q}^T \mathbf{A} \mathbf{Q}$ is (approximately) diagonally dominant. This improves the quality of the bounds provided by the Gerschgorin interval $\bigcup_{i=1}^{n} \left[d_i - r_i, d_i + r_i \right]$ which can otherwise be rather pessimistic.

Corollary 2. (Gerschgorin Circle Theorem for Symmetric, Tridiagonal Matrices) Let $\mathbf{T} \in \Re^{n \times n}$ be tridiagonal and symmetric, and let $\mathbf{a}$ and $\mathbf{b}$ be the vectors containing the diagonal and off-diagonal elements of $\mathbf{T}$, respectively. The spectrum $\lambda(\mathbf{T})$ is then bound:

$$\lambda(\mathbf{T}) \subseteq \bigcup_{i=1}^{n} \left[a_i - r_i, a_i + r_i \right]$$

with $r_i = b_i + b_{i-1}$ for $i = 2 : n-1$, $r_1 = b_1$, and $r_n = b_{n-1}$.

Eigenvalue Shifts and Sylvester's Law of Inertia

Theorem 2. Let $\mathbf{A} \in \Re^{n \times n}$ be a square matrix with eigenvalues λ_i, and let $\mu \in \Re$ be a *shift index*. The matrix $\overline{\mathbf{A}}_\mu = \mathbf{A} - \mu \mathbf{I}$ has then eigenvalues $\overline{\lambda}_i = \lambda_i - \mu$. The eigenvectors of $\overline{\mathbf{A}}_\mu$ and $\mathbf{A}$ are identical.

Proof: Consider the characteristic polynomial

$$\det(\mathbf{A} - \lambda_i \mathbf{I}) = 0 .$$

By replacing $\mathbf{A}$ with $\mathbf{A} = \overline{\mathbf{A}}_\mu + \mu \mathbf{I}$ we obtain

$$\det\left(\left(\overline{\mathbf{A}}_\mu + \mu \mathbf{I} \right) - \lambda_i \mathbf{I} \right) = 0 .$$

Rearranging terms yields

$$\det\left(\overline{\mathbf{A}}_\mu + (\mu \mathbf{I} - \lambda_i \mathbf{I}) \right) = 0$$
$$\det\left(\overline{\mathbf{A}}_\mu - (\lambda_i - \mu) \mathbf{I} \right) = 0.$$

The second claim follows by simplifying

$$(\mathbf{A} - \mu \mathbf{I}) \mathbf{U} = (\mathbf{\Lambda} - \mu \mathbf{I}) \mathbf{U}$$

which is the eigenvalues equation $\mathbf{A}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}$ with $\overline{\mathbf{A}}_\mu = \mathbf{A} - \mu \mathbf{I}$ and $\overline{\mathbf{\Lambda}}_\mu = \mathbf{\Lambda} - \mu \mathbf{I}$. □

Theorem 2 establishes a relationship between the eigenvalues of $\mathbf{A}$ and those of a shifted version $\overline{\mathbf{A}}_\mu$ of $\mathbf{A}$. It is the foundation for many eigenvalue algorithms because the shift affects the distance between eigenvalues, and therefore the conditioning of the problem, but does not change the eigenvectors. For eigenvalues of $\mathbf{A}$ that are close together it is often more stable and more efficient to first shift $\mathbf{A}$ and then determine the eigenvalues of the shifted matrix. Theorem 2 can then be used to relate the eigenvalues of the shifted matrix back to those of $\mathbf{A}$. See, for example, the technical report by Willems et al. [2] for a more detailed discussion.

Theorem 3. (Sylvester's Law of Inertia) Let the *inertia* of a square matrix $\mathbf{A} \in \Re^{n \times n}$ be a triple (n, z, p) of integer numbers, with n, z, and p being the numbers of negative, zero, and positive elements in the spectrum of $\mathbf{A}$, respectively. If $\mathbf{A}$ is symmetric and $\mathbf{X} \in \Re^{n \times n}$ is nonsingular, then the matrices $\mathbf{A}$ and $\mathbf{X}^T \mathbf{A} \mathbf{X}$ have the same inertia.

Proof: See [1, pp. 403] □

Theorem 4. Let $\mathbf{T} \in \Re^{n \times n}$ be tridiagonal and symmetric, and positive definite, and let $a_i b_i \neq 0$ for $i = 1:n$, that is $\mathbf{T}$ is non degenerated, then there exists a factorization

$$\mathbf{T} = \mathbf{L}\mathbf{D}\mathbf{L}^T$$

where $\mathbf{D}$ is diagonal and $\mathbf{L}$ is lower bidiagonal with all diagonal elements being 1.

Proof: Consider the Cholesky factorization $\mathbf{T} = \overline{\mathbf{L}}\overline{\mathbf{L}}^T$ of $\mathbf{T}$. The existence of an $\mathbf{L}\mathbf{D}\mathbf{L}^T$ factorization of $\mathbf{T}$ then follows immediately with $\overline{\mathbf{L}} = \mathbf{L}\mathbf{D}^{1/2}$ [13]. □

Positive definite factorizations such as the Cholesky factorization or the $\mathbf{L}\mathbf{D}\mathbf{L}^T$ factorization exists only if a matrix is positive definite. If this is not the case for a given tridiagonal symmetric matrix then we can choose an initial shift so that $\mathbf{T}$ becomes positive definite and use Theorem 2 to relate the computed eigenvalues to those of the original matrix, see [14] for a more detailed discussion. For matrices where any $a_i b_i = 0$ it is always possible to split the input matrix and compute a factorization for the two resulting sub-matrices; refer to the technical report by Willems et al. [2] for more details.

Corollary 3. The inertia of a tridiagonal symmetric matrix $\mathbf{T} \in \Re^{n \times n}$ can be computed by first determining the $\mathbf{L}\mathbf{D}\mathbf{L}^T$ factorization of $\mathbf{T}$, and then counting positive and negative eigenvalues of the diagonal matrix $\mathbf{D}$.

Eigenvalue Count

Definition 2. Let $\mathbf{T} \in \Re^{n \times n}$ be symmetric and tridiagonal, and let $x \in \Re$ be a shift. The *eigenvalue count* $C(x)$ of $\mathbf{T}$ is then the number of eigenvalues of $\mathbf{T}$ smaller than x

$$C(x) = |\lambda_x(\mathbf{T})| \text{ with } \lambda_x(\mathbf{T}) \equiv \{\lambda(\mathbf{T}) \mid \lambda_i < x\}.$$

The eigenvalue count $C(x_1, x_2)$ of an interval $(x_1, x_2]$ is

$$C(x_1, x_2) \equiv C(x_2) - C(x_1).$$

An interval $(x_1, x_2]$ is *empty* if $C(x_1, x_2) = 0$.

With Corollary 3 the eigenvalue count $C(x)$ of $\mathbf{T}$ can be found with the following algorithm:

1. Shift $\mathbf{T}$ by x to obtain $\overline{\mathbf{T}}_x$.
2. Compute the $\mathbf{LDL}^T$ factorization of $\overline{\mathbf{T}}_x$.
3. Count the negative elements of the diagonal of $\mathbf{D}$.

Computing the full $\mathbf{LDL}^T$ factorization is however wasteful because we are only interested in the signs of the non-zero elements of $\mathbf{D}$. Algorithm 1 finds the eigenvalue count $C(x)$ efficiently without computing a full $\mathbf{LDL}^T$ factorization and without storing the full vector of diagonal elements of $\mathbf{D}$ [6][1]:

Algorithm 1:

Input: Diagonal and off-diagonal elements a_i and b_i of the tridiagonal symmetric matrix $\mathbf{T} \in \Re^{n \times n}$, and a shift $x \in \Re$. The algorithm requires b_0 which is defined as $b_i = 0$.

Output: count $\equiv C(x)$

```
count = 0
d = 1
for i = 1 : n
    d = $a_i$ – x – ($b_i$ * $b_i$) / d
    if (d < 0)
        count = count + 1
    end
end
```

[1] Let $\mathbf{B}$ be an upper bidiagonal matrix and consider the tridiagonal symmetric matrix $\mathbf{B}^T\mathbf{B}$. Using Algorithm 1 to determine an eigenvalue count of $\mathbf{B}^T\mathbf{B}$ is unwise because computing $\mathbf{B}^T\mathbf{B}$ explicitly is numerically unstable [1, p. 453]. Fernando therefore developed a modified version of Algorithm 1 that works directly with the non-zero elements of the $\mathbf{B}$ [7].

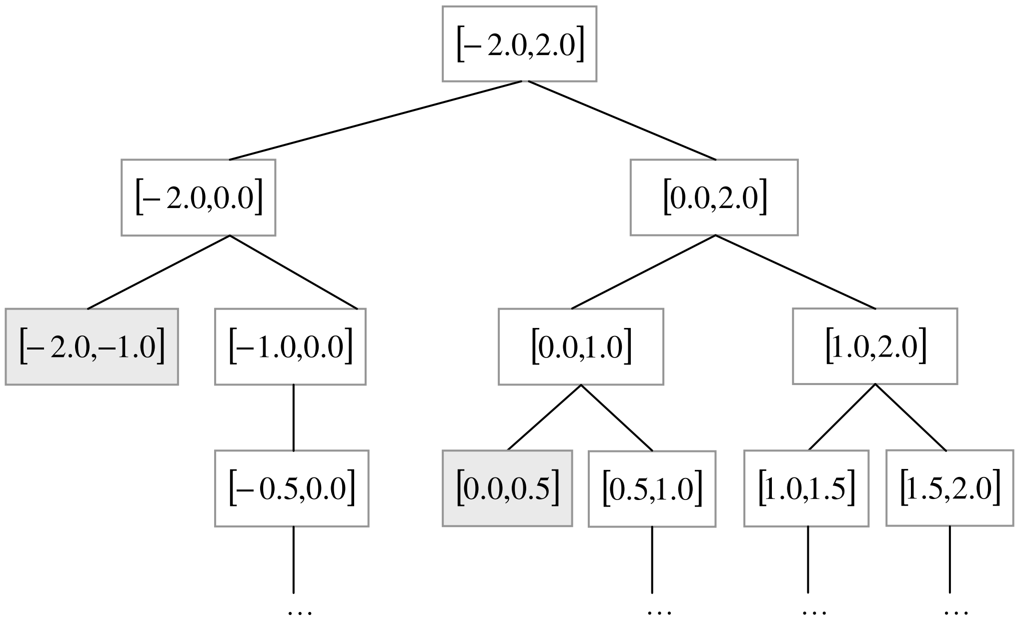

Figure 1. Example interval tree for a matrix $\mathbf{T} \in \Re^{4\times4}$ with Gerschgorin interval $[-1.0,2.0]$ and $\lambda(\mathbf{T}) = \{-0.25, 0.2, 1.2, 1.7\}$. The grayed out intervals are empty leaf nodes. All other branches terminate at non-empty intervals. Depending on the desired accuracy, approximations of the eigenvalues can be obtained by further refining non-empty intervals and always keeping the child interval whose eigenvalue count is strictly positive.

A Bisection-Like Algorithm to Compute Eigenvalues

Armed with the results of the last section we can now discuss a bisection-like algorithm for computing all eigenvalues of a tridiagonal symmetric matrix.

Let $\mathbf{T} \in \Re^{n\,x\,n}$ be tridiagonal and symmetric, and let $(l_{0,0}, u_{0,0}]$ be the Gerschgorin interval of $\mathbf{T}$, computed as described in Corollary 2. We can then subdivide $(l_{0,0}, u_{0,0}]$ into smaller *child* intervals $(l_{1,0}, u_{1,0}]$ and $(l_{1,1}, u_{1,1}]$, and obtain the eigenvalues counts $C(l_{1,0}, u_{1,0}) = k$ and $C(l_{1,1}, u_{1,1}) = n - k$ using Algorithm 1. Obviously, the child intervals provide better bounds on the eigenvalues than the initial Gerschgorin interval did. If we subdivide each of the child intervals then we can further improve these bounds, and by recursively continuing this process we can obtain approximations to the eigenvalues to any desired accuracy. Many of the generated intervals will be empty and further subdividing these intervals will only create more empty – and for us irrelevant – intervals. We therefore do not subdivide empty intervals.

As shown in Figure 1, the set of intervals generated by the subdivision process resembles an unbalanced binary tree with the Gerschgorin interval as root. A node of the tree is denoted by $(l_{j,k}, u_{j,k}]$ with $j \in J$ being the level of the tree and $k \in K(j)$ being an index set defined over all nodes of the tree at level j. Leaf nodes are either empty intervals or converged intervals for which the (absolute or relative) size is smaller than a desired accuracy Δ.

Algorithm 2 summarizes the bisection algorithm for computing the eigenvalues of a tridiagonal symmetric matrix $\mathbf{T}$.

Algorithm 2:

```
Input: Tridiagonal symmetric matrix $\mathbf{T}$, desired accuracy $\Delta$.
Output: Spectrum $\lambda(\mathbf{T})$ of $\mathbf{T}$ in a list $\boldsymbol{O}$.
2.1. Determine the Gerschgorin interval $(l_{0,0}, u_{0,0}]$ of $\mathbf{T}$ and store the interval on a
     stack $\boldsymbol{S}$.
2.2. Repeat until $\boldsymbol{S}$ is empty:
     2.2.a.  Pop one interval $(l_{j,k}, u_{j,k}]$ from the stack.
     2.2.b.  Compute $m_{j,k} \in (l_{j,k}, u_{j,k}]$.
     2.2.c.  Compute $C(m_{j,k})$ using Algorithm 1.
     2.2.d.  Determine if non-empty child intervals converge.
     2.2.e.  For non-empty, converged child intervals compute an approximation of the
             eigenvalue $\lambda_i$ and store it in $\boldsymbol{O}$ at index $C(\lambda_i)$.
     2.2.f.  Push non-empty, unconverged child intervals on $\boldsymbol{S}$.
```

CUDA Implementation

In this section we will describe how Algorithm 2 has been adopted for a data parallel programming model and implemented with CUDA [10]. Our implementation employs the scan algorithm[2] for various purposes. Readers not familiar with this algorithm should refer, for example, to the paper by Blelloch [8] or the technical report by Harris et al. [9] for more details. Throughout this report we will employ addition as scan operator.

An Efficient Parallel Algorithm

To implement Algorithm 2 efficiently on a data parallel architecture we first have to identify computations that are *independent* – and can thus be performed in parallel – and are *similar or identical* for many data items, for example all elements in an array.

Design of a Parallel Algorithm

When the eigenvalue count is computed with Algorithm 1 similar computations are performed for all diagonal and off-diagonal elements of the input matrix. The variable d, however, is iteratively updated in every iteration of the algorithm making every step of the loop dependent on the previous one. Algorithm 1 has therefore to be computed serially and we cannot parallelize the eigenvalue count computation for one shift.

The interval splits in Step 2.2 of Algorithm 2 require the Gerschgorin interval as input and Step 2.1 therefore has to be computed before Step 2.2. Likewise, if we consider two child nodes $\left(l^k_{j+1,k'}, u^k_{j+1,k'}\right]$ and $\left(l^k_{j+1,k'+1}, u^k_{j+1,k'+1}\right]$, then we obviously first have to perform Steps 2.2.a to 2.2f for the parent interval $\left(l_{j,k}, u_{j,k}\right]$, and generate $\left(l^k_{j+1,k'}, u^k_{j+1,k'}\right]$ and $\left(l^k_{j+1,k'+1}, u^k_{j+1,k'+1}\right]$, before we can process the child intervals. Taking this a step further and also considering dependencies across multiple levels, we can see that the interval tree can be regarded as a dependency graph for the computations. Nodes that have a (generalized) parent-child relationship depend on each other and have to be processed serially whereas the computations for other nodes are independent and can be performed in parallel.

We will gain a better understanding of what this means in practice if we follow the program flow of the computations in Step 2.2. Initially, the stack $\boldsymbol{S}$ contains only the Gerschgorin interval $\left(l_{0,0}, u_{0,0}\right]$ which is the root of the interval tree. All other nodes are children of $\left(l_{0,0}, u_{0,0}\right]$ and we therefore have to perform Steps 2.2.a to 2.2f for this interval before we can process any other interval. Splitting the Gerschgorin interval creates the two nodes $\left(l_{1,0}, u_{1,0}\right]$ and $\left(l_{1,1}, u_{1,1}\right]$ at the first level of the interval tree. These nodes are siblings and can thus be processed independently. Performing the interval splitting for $\left(l_{1,0}, u_{1,0}\right]$ and $\left(l_{1,1}, u_{1,1}\right]$ in parallel – computing Steps 2.2a to 2.2f with different input data – yields the four intervals on level two of the tree. If we continue this process then the stack will always

[2] In the literature the scan algorithm is also known as prefix sum.

contain only intervals on one level of the interval tree – not having a parent-child relationship – which can be processed in parallel.

A data parallel implementation of Step 2.2 of Algorithm 2 is provided in the following algorithm.

Algorithm 3.

```
3.1. Perform in parallel for all intervals $(l_{j,k}, u_{j,k}]$ (all at level $j$) in the list $\boldsymbol{L}$.
        3.1.a.  Compute $m_{j,k} \in (l_{j,k}, u_{j,k}]$.
        3.1.b.  Compute $C(m_{j,k})$.
        3.1.c.  Determine if non-empty child intervals converge.
        3.1.d.  For non-empty, converged child intervals compute an approximation of
                the eigenvalue $\lambda_i$ and store it in $\boldsymbol{O}$ at index $C(\lambda_i)$.
        3.1.e.  Store non-empty, unconverged child intervals in $\boldsymbol{L}$.
```

Note that Algorithm 3 does no longer use a stack to store tree nodes but only a simple list $\boldsymbol{L}$. The Algorithm terminates successfully when all non-empty intervals converged. The reported eigenvalues are then, for example, the midpoints of the final intervals. Care is required when the distance between eigenvalues is less than the desired accuracy Δ and converged intervals contain multiple eigenvalues. In this case we store k instances of the eigenvalue approximation at the appropriate indices $C(\lambda_i)$, where k is the multiplicity of the eigenvalue. Situations where Algorithm 3, as described above, does not terminate successfully are described in the following sections.

Thread Allocation

Although it is possible to process multiple intervals with one thread, we decided to use the the "natural" approach on a data parallel architecture and employ one thread to process one interval.

CUDA does not permit dynamic creation of new threads on the device. The number of threads specified at kernel launch time has therefore to match the maximum number of intervals that will be processed in parallel. Assuming that Δ is smaller than the minimum distance between any two eigenvalues, then the number of threads has to equal the number of eigenvalues of the input matrix. Note that this implies that for the first tree levels there are many more threads than intervals to process and many threads have to be inactive until a higher tree level has been reached. The available parallelism is therefore fully exploited only after some iterations of Step 3.1.

Interval Subdivision

Until now we did not specify how $m_{j,k}$ should be chosen but merely required that $m_{j,k} \in (l_{j,k}, u_{j,k}]$. In the literature different approaches such as midpoint subdivision or Newton-like methods have been used to determine $m_{j,k}$ [6,7,15]. We employ midpoint subdivision which ensures that all child intervals on the same tree level have the same size. All converged intervals are therefore on the same tree level and reached at the same processing step. This reduces divergence of the computations and therefore improves efficiency (cf. [10, Chapter 5.1.1.2]).

Eigenvalue Count Computation

At the beginning of this report we introduced Algorithm 1 to compute the eigenvalue count $C(x)$. Although the algorithm is correct in a mathematical sense, it is not monotonic when computed numerically. It is therefore possible for the algorithm to return an eigenvalue count $C(x_1)$ so that $C(x_1) > C(x_2)$ although $x_1 < x_2$. Clearly, this will lead to problems in practice. In our implementation we therefore employ Algorithm FLCnt_IEEE from the paper by Demmel et al. [6] which is guaranteed to be monotonic.

Data Representation

The data that has to be stored for Algorithm 3 are the active intervals and the non-zero elements of the input matrix. Each interval is represented by its left and right interval bounds and the eigenvalue counts for the bounds. The input matrix can be represented by two vectors containing the main and the first upper (or lower) diagonal, respectively.

Summarizing the guidelines from the CUDA programming guide [10], to obtain optimal performance on an NVIDIA compute device it is important to represent data so that

- (high-latency) data transfers to global memory are minimized,
- uncoalesced (non-aligned) data transfers to global memory are avoided, and
- shared memory is employed as much as possible.

We therefore would like to perform all computations entirely in shared memory and registers. Slow global memory access would then only be necessary at the beginning of the computations to load the data and at the end to store the result. The limited size of shared memory – devices with compute capabilities 1.x have 16 KB which corresponds 4096 32-bit variables – makes this unfortunately impossible. For matrices with more than 2048×2048 elements shared memory would not even be sufficient to store the matrix representation. We therefore store only the active intervals in shared memory and the two vectors representing the input matrix are loaded from global memory whenever the eigenvalue count has to be computed in Step 3.1.b.

Shared memory is not only limited in its size but also restrictive in that it is shared among the threads of a single thread block. With a maximum of 512 threads per block and by using one thread to perform Step 3.1 for each interval, we are thus limited to matrices with at most 512×512 elements. Although such an implementation is very efficient, it defeats our goal of an algorithm that can process arbitrary size matrices. An extension of Algorithm 3 for arbitrary size matrices will therefore be discussed in the next section. For the remainder of the section, however, we will restrict ourselves to the simpler case of matrices with at most 512 eigenvalues.

For simplicity, in practice the list $\boldsymbol{L}$, containing the information about the active intervals, is represented by four separate arrays:

```
__shared__ float s_left[MAX_INTERVALS_BLOCK]

__shared__ float s_right[MAX_INTERVALS_BLOCK]
__shared__ unsigned int s_left_count[MAX_INTERVALS_BLOCK]
__shared__ unsigned int s_right_count[MAX_INTERVALS_BLOCK],
```

where the left and right interval bounds are stored in `s_left` and `s_right`, respectively, and `s_left_count` and `s_right_count` are used to represent the eigenvalue counts.

A problem which is specific to a parallel implementation of the bisection algorithm results from the data-dependent number of child intervals. In a serial implementation we could write the data of the actual child intervals, either one or two, at the end of the four arrays and use an index pointer to track where the next data has to be written. However, in a data parallel programming model all threads write at the same time. To make sure that no data is overwritten, we therefore have to assume that every interval has two children when we write the data; the addresses `idx_0` and `idx_1` have thus to be computed as

```
idx_0 = threadIdx.x;
idx_1 = threadIdx.x + blockDim.x;
```

where `blockDim.x` is the total number of threads in the thread block. This generates a sparse list of interval data with unused elements in between. One possible approach would be to continue with this sparse list. However, because we do not know which child intervals are actually non-empty and were written to the four arrays, we have to use in the next iteration of Step 3.1 as many threads as there are *possible* child intervals, and we would also again have to compute the indices for the result data as listed above which would generate a full, balanced binary tree. Obviously, even for small matrices with at most 512×512 elements we would easily exceed the available 512 threads before all intervals converged. In the implementation, we therefore *compact* the list of child intervals after Step 3.1 before processing nodes on the next level of the interval tree. In the literature this problem is known as stream compaction, see for example [9]. The compaction takes as input a binary array which is one for all intervals that are non-empty, and zero otherwise. Performing a scan of this array yields the indices for the non-empty intervals so that these form a compact list, and the intervals are then stored directly to their final location.

In our implementation, however, we do not scan the whole array of possible child nodes. Every interval is guaranteed to have at least one non-empty child interval. If we store at the address `idx_0` always this non-empty child then no compaction is needed for this part of the interval list. In practice, we therefore store the first child interval directly into `s_left`, `s_right`, `s_left_count`, and `s_right_count` using `idx_0` and perform the compaction only for the second child intervals using a binary list `s_compaction_helper`. After the final addresses have been computed the second child intervals are stored after the first ones into the four shared memory arrays. Performing the scan only for the second child intervals thereby reduces the costs by half. With the compaction step Algorithm 3 takes the form:

Algorithm 3.

```
3.1. Perform in parallel for all intervals $(l_{j,k}, u_{j,k}]$ (all at level $j$) in the list $\boldsymbol{L}$.
    3.1.a.  Compute $m_{j,k} \in (l_{j,k}, u_{j,k}]$.
    3.1.b.  Compute $C(m_{j,k})$.
    3.1.c.  For non-empty child intervals determine if they converged.
```

3.1.d. Store non-empty, unconverged child intervals in ***L***.

3.1.e. Write the binary list `s_compaction_helper`.

3.2. Perform scan on `s_compaction_helper`.

3.3. Write all non-empty second child intervals to ***L*** using the addresses from `s_compaction_helper`.

Note that we no longer store non-empty, converged child intervals directly in the output list but repeat them on subsequent tree levels. In the implementation we therefore keep the data of converged intervals in ***L*** until Algorithm 3 terminated. This enables coalesced writes when the data is stored to global memory.

The elements of the two vectors representing the input matrix are loaded from global memory only during the eigenvalue count computation in Step 3.1.b. Given Algorithm 1 it would be possible to load only a single element of each vector at a time, compute one iteration of the loop, then load the next element, and so on. This strategy is inefficient because the loads are uncoalesced and because for low levels of the interval tree only a small number of threads would perform a load. In these cases the memory access latency could not be hidden by issuing load instructions for other threads. A better strategy is therefore to always load chunks of matrix data into shared memory and then compute some iterations of Algorithm 1 using this data. Coalescing the reads from global memory is then straightforward and memory latency can be hidden by using not only the active threads to load the data from global memory but all threads in the current thread block and using a sufficiently large number of threads per block. Loading chunks of data requires temporary storage of the information in shared memory. Two additional large arrays would, however, exceed the available 16 KB of shared memory. We therefore employ `s_left` and `s_right` to store the matrix elements and "cache" the interval bounds during the computation of Step 3.1.b in registers.

Computation of Eigenvalues for Large Matrices

So far we have described how Algorithm 3 can be implemented efficiently for small matrices with up to 512×512 elements by using one thread to process each interval and storing all active intervals in shared memory. For large matrices, however, there are not enough threads in a thread block to always be able to process all intervals simultaneously. It would be possible to process blocks of intervals serially. However, NVIDIA compute devices can process multiple thread blocks in parallel and we would therefore not exploit the full available computing power. Additionally, with significantly more than 512 eigenvalues it would no longer be possible to store all active intervals in shared memory. One possible alternative would be to *not* use shared memory but perform all computations directly from global memory. Then only minor modifications to Algorithm 3 would be necessary to process arbitrary size matrices with multiple thread blocks. However, the high latency of global memory and the lack of efficient synchronization between thread blocks make this approach inefficient. We therefore followed a different strategy to extend Algorithm 3 for arbitrary size matrices.

Let us consider for a moment what happens if Algorithm 3 is executed with a matrix with more than 512×512 elements as input. If we assume that no eigenvalues are clustered (that is the distance between the eigenvalues is larger than Δ) then eventually the number of active intervals will exceed the number of available threads. The algorithm will then terminate and return a list ***U*** of, in general, unconverged intervals that contain one or multiple eigenvalues. Although this is not yet the desired result, we can now determine the eigenvalues by continuing the bisection process on each of the intervals in $\boldsymbol{U}$. The discussion in this section will focus on how the list $\boldsymbol{U}$ of intervals is generated – in fact, we will see shortly that two lists are generated – and how the intervals can be processed efficiently.

A naïve approach to process $\boldsymbol{U}$ would be to employ Algorithm 3, as described in the last section, but always use as input always one interval from $\boldsymbol{U}$ instead of the Gerschgorin interval. This is however inefficient. Most intervals that result from the first step contain far less than 512 eigenvalues. The generated interval tree would therefore be very shallow and we would never reach high tree levels where the available parallelism could be fully exploited. We therefore employ a different approach to process the intervals in $\boldsymbol{U}$. First, we sort the intervals in two lists $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ containing only one and multiple eigenvalues, respectively. After $\boldsymbol{U_m}$ has been obtained, the intervals are grouped into blocks or batches so that none of them contains more than `MAX_THREADS_BLOCK` eigenvalues. The sorting into $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ and the batching are all performed as post-processing step to Algorithm 3. We then launch two additional kernels to process the elements in $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$, respectively. Each batch of intervals in $\boldsymbol{U_m}$ is thereby processed by one thread block guaranteeing that one additional kernel launch is sufficient to determine all eigenvalues. Our motivation to perform the rather expensive sorting to obtain $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ is two-fold. Firstly, it allows us to employ specialized kernels when we process each interval types which greatly improves the efficiency, and secondly, it reduces the amount of intermediate data that has to be read back to the host to two values; the number of elements in $\boldsymbol{U_o}$ and the number of batches of elements of $\boldsymbol{U_m}$.

Algorithm 4 summarizes the computations for matrices with more than 512×512 elements.

Algorithm 4.

```
4.1.  Coarse-grain processing.
      4.1.a.  Perform Algorithm 3 and obtain a list $U$ of intervals containing one or
              more eigenvalues.
      4.1.b.  Sort $U$ to obtain two lists $U_o$ and $U_m$ so that each one contains only
              intervals with one and multiple eigenvalues, respectively.
      4.1.c.  Batch $U_m$ into blocks so that none of the blocks contains more than
              MAX_THREADS_BLOCK eigenvalues
4.2.  Process the intervals in $U_o$ to obtain the contained eigenvalues.
4.3.  Process the intervals in $U_m$ to obtain the contained eigenvalues by using one
      thread block for each batch.
```

Step 4.1, 4.2, and 4.3 each correspond to a separate kernel launch. Step 1.b and Step 1.c are performed as a post-processing step of Algorithm 3 in Step 1.a.

On a data parallel architecture the sorting and blocking (or batching) in Step 4.1.b and Step 4.1.c are complex operations. These additional computations are nonetheless beneficial because they significantly simplify the computations in Step 4.2 and Step 4.3. In particular the sorting and batching enable the use of customized – and therefore more efficient – variants of Algorithm 3 to process $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$, respectively.

Algorithm 5 computes the eigenvalues contained in the intervals in $\boldsymbol{U_o}$. Every interval processed in this algorithm, either as input or generated during the computation, contains exactly one eigenvalue and has therefore exactly one child interval. This allows us to omit the compaction step employed in Algorithm 3 and makes Algorithm 5 significantly more efficient.

Algorithm 5.

```
5.1.   Read a set of intervals $(l_{j,k}, u_{j,k}]$ of $\boldsymbol{U_o}$ into shared memory.
5.2.   Repeat until all intervals converge.
    5.2.a.   Compute $m_{j,k} \in (l_{j,k}, u_{j,k}]$.
    5.2.b.   Determine $C(m_{j,k})$.
    5.2.c.   Determine if non-empty child intervals converge.
    5.2.d.   Store non-empty, unconverged child intervals in $\boldsymbol{L}$ or the original interval if
             it was already converged.
5.3.   Write eigenvalue approximations to global memory.
```

To efficiently compute Algorithm 5 the length of ***U***$\boldsymbol{U_o}$ is read back to the host and the number of thread blocks for Step 4.2 is chosen so that all intervals in $\boldsymbol{U_o}$ can be processed. The block id is used by each thread block to read the correct set of intervals from $\boldsymbol{U_o}$ into shared memory. Note that each thread block is launched with a large, prescribed number of threads even if the length of $\boldsymbol{U_o}$ is small. As in Algorithm 3 this enables to load the input matrix data more efficiently into shared memory when the eigenvalue count has to be computed.

Algorithm 6 is employed to find the eigenvalues for the intervals in $\boldsymbol{U_m}$.

Algorithm 6.

```
6.1.   Read one block of intervals from $\boldsymbol{U_m}$ into shared memory.
6.2.   Perform in parallel for all $n_j$ active intervals at level $j$:
    6.2.a.   Compute $m_{j,k} \in (l_{j,k}, u_{j,k}]$.
    6.2.b.   Compute $C(m_{j,k})$.
    6.2.c.   Determine if non-empty child intervals converge.
    6.2.d.   Store converged child intervals in an output list $\boldsymbol{O}$.
    6.2.e.   Store non-empty, unconverged child intervals in $\boldsymbol{L}$. or the original interval
             if it was already converged.
    6.2.f.   Perform scan on compaction.
    6.2.g.   Write all non-empty second child intervals to $\boldsymbol{L}$ using the addresses from
             compaction.
6.3.   Store eigenvalue approximations in global memory.
```

Algorithm 6 resembles Algorithm 3 very closely but takes as input not the Gerschgorin interval but a block of intervals. Thus, during processing not a tree but a forest of intervals is generated. In practice this is only of secondary interest as Algorithm 3 can be used without modifications to process a forest of trees instead of a single tree. The interval blocks that form the input to Algorithm 6 are generated so that the maximum number of active intervals in Step 6.2 never exceeds the available number of threads or the available shared memory space. All eigenvalues in $\boldsymbol{U_m}$ can therefore be found with one kernel launch by having as many thread blocks as interval batches. The number of batches is read from the device after Step 4.1.c.

Generating Lists and Blocks of Intervals

In the following we will describe in more detail how the two lists $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ are generated in Step 4.1.b of Algorithm 4, and how $\boldsymbol{U_m}$ is subdivided into blocks in Step 4.1.c.

The input to Step 4.1.b is $\boldsymbol{U}$ which contains the unconverged intervals resulting from Step 4.1.a. The lists $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$, formed by intervals containing one and multiple eigenvalues, respectively, are computed by two compaction operations. The input to both operations is $\boldsymbol{U}$ but when $\boldsymbol{U_o}$ is computed all intervals containing multiple eigenvalues are ignored, and when $\boldsymbol{U_m}$ is computed intervals containing only one eigenvalues are ignored. The difference between the two compaction operations is therefore only the subsets of intervals from $\boldsymbol{U}$ that are considered and both operations can therefore be performed simultaneously significantly reducing the overall costs. The result is two index lists that determine the positions of the elements of $\boldsymbol{U}$ in either $\boldsymbol{U_o}$ or $\boldsymbol{U_m}$. To save shared memory resources, we use `s_left_count` and `s_right_count` as buffers for the compaction operations and "cache" the eigenvalue counts in registers. Note that each thread has to cache the information for up to two intervals because Step 1.a in Algorithm 5 terminates when the number of active intervals *exceeds* the number of threads in the thread block. After the addresses have been computed it would be possible to directly write out the elements of $\boldsymbol{U}$ to representations of $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ in global memory. The writes would, however, in general be uncoalesced and therefore slow. Shared memory has no coalescing constraints and we can thus efficiently generate $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ in shared memory, and then copy the final lists to global memory with coalesced writes. Note that separate shared memory arrays for $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$ would exceed the available 16 KB. We therefore use `s_left`, `s_right`, `s_left_count`, and `s_right_count` to store the two lists sequentially one after the other by offsetting the elements of $\boldsymbol{U_m}$ by the length of $\boldsymbol{U_o}$.

Compaction, as used for generating $\boldsymbol{U_o}$ and $\boldsymbol{U_m}$, is a standard technique in parallel programming and our implementation differs from a textbook example only in that we perform two compaction operations in parallel. More interesting is the generation of blocks, or batches, of intervals so that the number of eigenvalues in each block never exceeds a prescribed threshold, in our case `MAX_THREADS_BLOCK`.

In general, the batch computation is a special instance of the knapsack problem which is known to be NP hard [12]. Finding the optimal solution is thus not practical. We therefore seek a greedy algorithm that provides batches of "good" quality so that the average number of intervals per block is close to the threshold but which can also be computed efficiently on a data parallel architecture. The solution proposed in this report is based on the scan algorithm as described by Blelloch [8] and Harris et al. [9]. We employ in particular the up-sweep of scan that generates a tree to determine the batches. Instead of building one complete binary tree containing all intervals, however, we generate a forest of trees so that for each tree the total numbers of eigenvalues in the leaf nodes does not exceed the prescribed maximum. In contrast to the original scan algorithm, we therefore combine two tree nodes $\nabla_{j,k}$ and $\nabla_{j,k+1}$ only if the new parent node does not contain more than `MAX_THREADS_BLOCK` eigenvalues. If the new node would exceed the threshold, we use an auxiliary buffer to mark $\nabla_{j,k}$ and $\nabla_{j,k+1}$ as root nodes of blocks. We build the trees by always discarding the right child node and replacing the left child with the parent (see Figure

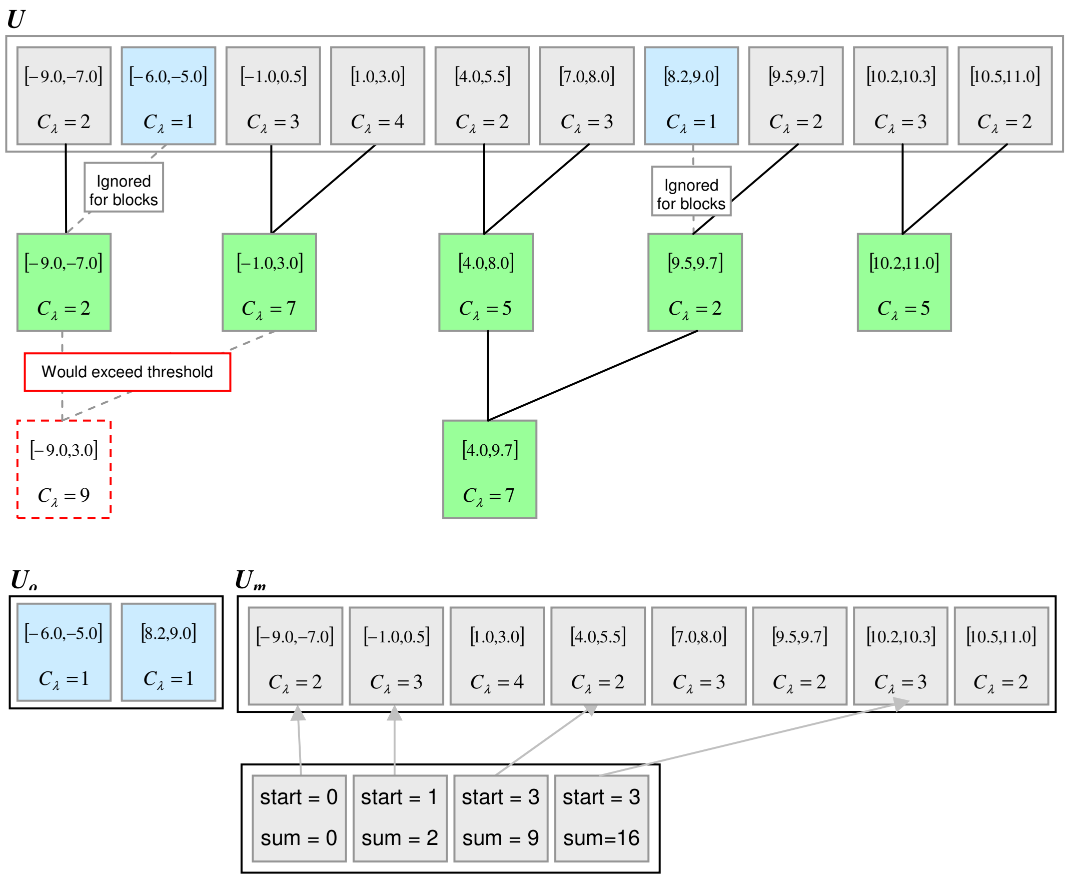
U
[−9.0,−7.0] $C_\lambda = 2$
[−6.0,−5.0] $C_\lambda = 1$
[−1.0,0.5] $C_\lambda = 3$
[1.0,3.0] $C_\lambda = 4$
[4.0,5.5] $C_\lambda = 2$
[7.0,8.0] $C_\lambda = 3$
[8.2,9.0] $C_\lambda = 1$
[9.5,9.7] $C_\lambda = 2$
[10.2,10.3] $C_\lambda = 3$
[10.5,11.0] $C_\lambda = 2$
Ignored for blocks
Ignored for blocks
[−9.0,−7.0] $C_\lambda = 2$
[−1.0,3.0] $C_\lambda = 7$
[4.0,8.0] $C_\lambda = 5$
[9.5,9.7] $C_\lambda = 2$
[10.2,11.0] $C_\lambda = 5$
Would exceed threshold
[−9.0,3.0] $C_\lambda = 9$
[4.0,9.7] $C_\lambda = 7$
U_o
U_m
start = 0 sum = 0
start = 1 sum = 2
start = 3 sum = 9
start = 3 sum=16

Figure 2. Example lists generated in Step 4.2 and 4.3 of Algorithm 4. The maximum number of eigenvalues allowed per batch is 8. Note that the interval containing one eigenvalue is ignored for building $\boldsymbol{U_m}$ and the block list.

2 for an example) and by storing root nodes in the auxiliary buffer at the same location as in the data buffer. In this way, the location of the first interval of a block in $\boldsymbol{U_m}$, which forms the start address of the block, can be looked up directly after building the forest. Finally, compacting the auxiliary buffer containing the start addresses yields a dense list of blocks ***B*** that is stored in global memory. In Step 4.3 each thread block reads the block start and end address from ***B*** and uses these to read the data from $\boldsymbol{U_m}$ that has to be processed by the block into shared memory (cf. Algorithm 6). Next to the start address we also compute for each block the accumulated number of eigenvalues in previous blocks which is needed to compactly store all eigenvalues obtained in Step 4.3 in global memory.

Computing the accumulated sum of eigenvalues requires an additional array in shared memory. Together with the auxiliary array that is used to compute the block start addresses the required memory would exceed the available shared memory resources. We therefore use `unsigned short` as data type for `s_left_count` and `s_right_count` and the two auxiliary buffers used to compute the blocks in Step 4.1. This limits us to matrices with not less than 65536×65536 elements but is sufficient for most applications. If input matrices can have more than 65536 eigenvalues then the size of the shared memory arrays can be restricted and again `unsigned int` can be used as data type.

As mentioned previously, the proposed algorithm to compute the blocks of $\boldsymbol{U_m}$ is not optimal and it is easy to construct pathological cases where it provides very poor results. In practice, however, we observed that on average each block contained about 75% of the maximum number of eigenvalues. This is sufficient to efficiently perform Step 3.4 of Algorithm 4.

Future Work

The implementation proposed in this report can be improved in various ways. For low levels of the interval tree it would be beneficial to employ multi-section instead of bisection. This would increase the parallelism of the computations more rapidly.

For small matrices with less than 256 eigenvalues it is possible to store both the active intervals and the input matrix in shared memory. The eigenvalue counts in the main loop of the algorithm can thus be computed without accessing global memory which will significantly improve performance.

For matrices that have between 512 and 1024 eigenvalues the current implementation can be improved by processing multiple intervals with each thread. This permits to directly employ Algorithm 3 to compute all eigenvalues and avoids multiple kernel launches.

There are also several possible improvements for large matrices with more than 1024 eigenvalues. Currently we require two additional kernel launches after Step 1 of Algorithm 4 to determine all eigenvalues. It would be possible to subdivide all intervals that contain only one eigenvalue to convergence in Step 1 of Algorithm 4. This would allow us to skip Step 2 and might be beneficial in particular for very large matrices where there are only a feew intervals containing one eigenvalue after Step 1.a. At the moment only one thread block is used in Step 1 of Algorithm 5. Thus, in this step only a small fraction of the available compute power is employed. One approach to better utilize the device during the initial step of Algorithm 5 would be to heuristically subdivide the initial Gerschgorin interval – ideally the number of intervals should be chosen to match the number of parallel processors on the device – and process all these intervals in the first processing step. For real-world problems where eigenvalues are typically not well distributed this might result in an unequal load balancing. We believe, however, that it will in most cases still outperform the current implementation. A more sophisticated approach would find initial intervals on the device so that the workload for all multi-processors is approximately equal in Step 1 (cf. [6]).

Conclusion

In this report we presented an efficient algorithm to determine the eigenvalues of a symmetric and tridiagonal real matrix using CUDA. A bisection strategy and an efficient technique to determine the number of eigenvalues contained in an interval form the foundation of the algorithm. The Gerschgorin interval that contains all eigenvalues of a matrix is the starting point of the computations and recursive bisection yields a tree of intervals whose leaf nodes contain an approximation of the eigenvalues. Two different versions of the algorithm have been implemented, one for small matrices with up to 512×512 elements, and one for arbitrary large matrices. Particularly important for the efficiency of the implementation is the memory organization which has been discussed in detail. Memory consumption and code divergence are minimized by frequent compaction operations. For large matrices the implementation requires two steps. Post-processing the result of the first step minimizes data transfer to the host and improves the efficiency of the second step.

Bibliography

1. G. H. Golub and C. F. van Loan, *Matrix Computations*, 2nd Edition, The John Hopkins University Press, Baltimore, MD, USA, 1989.
2. P. R. Willems, B. Lang, C. Voemel, *LAPACK Working Note 166: Computing the Bidiagonal SVD Using Multiple Relatively Robust Representations*, Technical Report, EECS Department, University of California, Berkeley, 2005.
3. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY, USA, 1992.
4. B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Upper Saddle River, NJ, USA, 1998.
5. I. S. Dhillon and B. S. Dhillon, *Orthogonal Eigenvectors and Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, 25 (3), 2003.
6. J. Demmel, I. Dhillon, and H. Ren, *On The Correctness Of Some Bisection-Like Parallel Eigenvalue Algorithms In Floating Point Arithmetic*, Trans. Num. Anal. (ETNA), 3, 1996.
7. K. V. Fernando, *Accurately Counting Singular Values of Bidiagonal Matrices and Eigenvalues of Skew-Symmetric Tridiagonal Matrices*, SIAM Journal on Matrix Analysis and Applications, 20 (2), 1999.
8. G. E. Blelloch, *Prefix Sums and Their Applications*, In Synthesis of Parallel Algorithms, Morgan Kaufmann, 1990.
9. M. Harris, S. Sengupta, and J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*, GPU Gems 3, Ed. H. Nguyen, Pearson Education, Boston, MA, 2007.
10. NVIDIA Corporation. *NVIDIA CUDA Programming Guide*. 2007.
11. T. H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, Second Edition, MIT Press, 2001.
12. I. S. Dhillon, *A New O(N^2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD. Thesis, University of California, Berkeley, Spring 1997.
13. I. S. Dhillon and B. N. Parlett, *Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices*, Linear Algebra and its Applications , Volume 387, Pages 1-28, 2004.
14. M. A. Forjaz and R. Ralha, *An Efficient Parallel Algorithm for the Symmetric Tridiagonal Eigenvalue Problem*, Selected Papers and Invited Talks of Vector and Parallel Processing - VECPAR 2000, Lecture Notes in Computer Science, Volume 1981, pp. 369, Springer Berlin / Heidelberg, 2001

Appendix

Source Code Files

Host source code files are shown black, device source code files in blue.

`main.cu`	Main file of the program.
`matlab.cpp`	Store result of the computation in Matlab file format.
`gerschgorin.cpp`	Computation of the Gerschgorin interval.
`bisect_small.cu`	512×512 elements (allocation of device code, kernel invocation, deallocation of resource, …).
`bisect_large.cu`	512×512 elements (allocation of device code, kernel invocation, deallocation of resource, …).
inc/bisect_large.cuh	Header file for `bisect_large.cu`.
inc/bisect_small.cuh	Header file for `bisect_small.cu`.
inc/config.h	General parameters (e.g. max. number of threads in one thread block).
inc/gerschgorin.h	Header for `gerschgorin.cpp`.
inc/matlab.h	Header for `matlab.cpp`.
inc/structs.h	Definition of structures which bundle kernel input and output parameter.
inc/util.h	Miscellaneous utility functionality.
kernel/bisect_kernel_large.cu	Kernel implementing Step 4.1.a to 4.1.c.
kernel/bisect_kernel_large_multi.cu	Kernel implementing Algorithm 6.
kernel/bisect_kernel_large_onei.cu	Kernel implementing Algorithm 5.
kernel/bisect_kernel_small.cu	Kernel implementing Algorithm 3.
kernel/bisect_util.cu	Functionality that is required by multiple kernels, for example implementation of Algorithm 1.

Default Behavior

If the program is run without command line arguments then the diagonal and off-diagonal elements of a matrix with 2048×2048 elements are loaded from the files `diagonal.dat` and `superdiagonal.dat` in the `data` directory, respectively. The output of the program is

```
Matrix size: 2048 x 2048
Precision: 0.000010
Iterations timing: 1
Result filename: 'eigenvalues.dat'
Gerschgorin interval: -2.894310 / 2.923304
Average time step 1: 5.263797 ms
Average time step 2, one intervals: 6.178160 ms
Average time step 2, mult intervals: 0.061460 ms
Average time TOTAL: 11.679418 ms

PASSED.
```

`PASSED` states that the correct result has been computed. For comparison a reference solution was loaded from `reference.dat` from the `data` directory.

Command Line Flags

`filename-result`	Name of the file where the computed eigenvalues are stored (only used when also `matrix-size` is user defined).	`eigenvalues.dat`
`iters-timing`	Number of iterations used to time the different kernels.	100
`precision`	Desired precision of the eigenvalues.	`0.00001`
`matrix-size`	Size of random matrix.	`2048`

Example: `%>eigenvalues --precision=0.001 --matrix-size=4096`

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com