A Scalable Parallel Bisection Algorithm for Symmetric Tridiagonal Eigenvalue Problem

Barok Imana  
Department of Engineering  
Trinity College  
Hartford, CT 06106 USA  
Email: barok.imana@trincoll.edu

Peter Yoon  
Department of Computer Science  
Trinity College  
Hartford, CT 06106 USA  
Email: peter.yoon@trincoll.edu

Abstract — Bisection method is a numerically stable algorithm used to find the eigenvalues of symmetric tridiagonal matrices. It is distinct from other methods in that it can be used to compute a subset of eigenvalues with high accuracy. However, the algorithm is significantly slow compared to other methods when a large number of eigenvalues are desired. Fortunately, the algorithm exhibits a high level of parallelism when it is implemented on various types of multiprocessors, including a single-GPU system. In this paper, we describe a highly scalable implementation using multi-GPU systems to accommodate large matrices. Our approach exploits the latest memory management features available on Nvidia Tesla K20c GPUs, including unified memory architecture, peer-to-peer data transfer, and dynamic parallelism. Our implementation was at least 60 times faster than a multi-core CPU system and exhibits a linear speedup with respect to the number of GPUs in the system.

Keywords: Symmetric eigenvalue problem, bisection method, parallel algorithm, general-purpose GPU computing

1. Introduction

Consider an \( n \times n \) symmetric tridiagonal matrix \( T \),

\[
T = \begin{pmatrix}
a_1 & b_1 & & \\
b_1 & a_2 & b_2 & \\
& a_3 & \ddots & \ddots \\
& & \ddots & \ddots \\
& & & a_{n-1} & b_{n-1} \\
& & & b_{n-1} & a_n
\end{pmatrix}
\]

Suppose that the eigenvalues of \( T \) are ordered so that

\[ \lambda_1 \leq \lambda \leq \cdots \leq \lambda_n \]

where each scalar \( \lambda_i, i = 1, 2, \ldots, n \), satisfies

\[ Tx_i = \lambda_i x_i \]

for non-zero vector \( x_i, i = 1, 2, \ldots, n \). Here, the vector \( x_i \) is called \( i \)-th eigenvector of \( T \) associated with the eigenvalue \( \lambda_i \).

The solution to symmetric tridiagonal eigenvalue problem is a critical part of computing eigenvalues and eigenvectors of general symmetric matrices, where the matrix is first reduced to symmetric tridiagonal form. In addition, the computation of eigenvalues and eigenvectors of symmetric tridiagonal matrices arises in many important applications areas including structural dynamics, quantum chemistry, oceanography, economics theory and control process.

Several approaches to computing eigenvalues of symmetric tridiagonal matrices have been proposed, including the QR iteration [1], the divide and conquer method [5], [6], and the bisection method [8], [9], [11]. The QR iteration has been considered the most efficient method to compute all eigenvalues, only requiring \( O(n) \) operations for a tridiagonal matrix, but, unfortunately, there are not efficient algorithms which are amenable to various parallel architectures.

The divide and conquer method is the fastest now available if all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired. The method, implemented in LAPACK [12], begins with dividing the original matrix into two smaller symmetric tridiagonal matrices, computing the eigenvalues of the submatrices, and combining the computed eigenvalues using rank-one modifications [5]. Unlike the QR iteration, the method has been successfully implemented on several multiprocessors [6].

The bisection method may be used to find all eigenvalues or a subset of the eigenvalues, requiring only \( O(nk) \) operations, where \( k \) is the number of eigenvalues desired. Since each eigenvalue can be computed independent of one another, the whole procedure can be made highly parallel. Several parallel implementations of the method have been proposed [9], [14]. In particular, as the use of general-purpose GPUs for scientific computing has been dramatically increasing in recent years, it has been noted that the GPUs have an enormous potential in computing eigenvalues using the bisection method.

The initial release of a GPU-accelerated bisection method has been included in Nvidia CUDA SDK [16]. This implementation utilizes simple yet highly efficient data structures for the division steps, but it suffers some issues such as overflow problems and its inability to deal with special structure.
of the matrix. To that end, Volkov and Demmel [15] proposed an improved version which overcomes architectural limitations and delivers high performance and high accuracy of the eigenvalues. However, neither implementations are scalable to multiple GPUs in order to accommodate large matrices.

In this paper, we focus on a highly scalable implementation of the bisection method which will be amenable to multi-GPU systems. Our approach exploits the latest memory management features available on the Nvidia Tesla K20c GPU, including unified memory architecture, peer-to-peer data transfer, and dynamic parallelism. Our implementation of the bisection method, which is entirely done on multiple GPUs, runs at least 60x faster for large matrices, compared to multi-core CPU versions and exhibits a linear speedup with the number of GPUs used.

The remainder of the paper is organized as follows: Section 2 briefly describes the bisection algorithm, Section 3 presents a detailed parallel implementation, Section 4 presents the experimental results followed by discussions and future directions.

2. The Bisection Method

The main idea of the bisection method is based on Sylvester’s Inertia Theorem [17], which states that the number of positive eigenvalues greater than \( \lambda \) is the same as the number of positive eigenvalues of \( T - \lambda I \), that is, the matrix of the form,

\[
\begin{pmatrix}
  a_1 - \lambda & b_1 \\
  b_1 & a_2 - \lambda & b_2 \\
  & b_2 & a_3 - \lambda & b_3 \\
  & & \ddots & \ddots & \ddots \\
  & & & b_{n-2} & a_{n-1} - \lambda & b_{n-1} \\
  & & & & b_{n-1} & a_n - \lambda 
\end{pmatrix}
\]

Let Inertia(\( T \)) be the triplet \( (\mu, \xi, \pi) \), where \( \mu \) is the number of negative eigenvalues of \( T \), \( \xi \) is the number of zero eigenvalues of \( T \), and \( \pi \) is the number of positive eigenvalues of \( T \). Then, the theorem states that

\[
\text{Inertia}(T) = \text{Inertia}(U^T U),
\]

where \( U \) is nonsingular. Furthermore, if \( T - \lambda I \) can be factorized to \( LDL^T \), where \( L \) is nonsingular and \( D \) diagonal, then

\[
\text{Inertia}(T - \lambda I) = \text{Inertia}(D).
\]

which leads to Algorithm 1 which computes the number of negative elements of \( D \), which is the same as the eigenvalues of \( T \) that are less than \( \lambda \).

In Algorithm 1, the function Count(\( T, \lambda \)), given a real shift value \( \lambda \), shifts matrix \( T \) by \( \lambda \), and then performs \( LDL^T \) decomposition on the resulting matrix. By a careful examination of the function, one can show that the number of negative entries on the diagonal of \( D \) is the number of eigenvalues of \( T \) that are less than \( \lambda \). Note that the function does not explicitly compute \( LDL^T \) decomposition. The bisection method uses Count(\( T, \lambda \)) function to calculate the total number of eigenvalues in interval \((a, b)\), which is equivalent to Count(\( T, b \) – Count(\( T, a \)).

The initial interval which contains all eigenvalues of \( T \) is given by Gerschgorin Circle Theorem [1] for symmetric tridiagonal matrix. The theorem states that all eigenvalues of \( T \) is bounded by the spectrum \( \lambda(T) \), such that

\[
\lambda(T) \in \bigcup [a_i - r_i, a_i + r_i]
\]

where

\[
r_i = b_i + b_{i-1}, i = 2, \ldots n - 1
\]

\[
r_1 = b_1, r_n = b_{n-1}
\]

Starting with a Gerschgorin interval, the bisection algorithm iteratively divides the interval into smaller subintervals. These subintervals are either discarded if they contain no eigenvalues of \( T \), or continued being subdivided until they are sufficiently small, which is determined by a given tolerance. Finally, the eigenvalues of \( T \) are approximated by taking either bounds or midpoints of converged intervals.

3. Parallel Implementation

In this section, we describe a scalable parallel bisection algorithm and its implementation on a multi-GPU system. The use of GPUs in scientific computing applications has risen dramatically in recent years mainly because of their highly parallel architecture, energy efficiency and cost-effectiveness. Unlike a CPU, a GPU consists of thousands of small computing cores designed to run tens of thousands of threads in parallel. Although each core may not be as powerful as a single CPU core, together, the thousands of cores produce higher throughput in applications that demand a high-level of data parallelism.

3.1 CUDA

One of the most important aspects in designing algorithms on GPUs is the memory structure. GPUs reside on an external hardware and have no direct access to the main memory of the CPU. Data residing on the main memory of the CPU has to be copied to the GPUs via system bus...
such as PCI Express. However, the PCI Express has limited bandwidth and poses a bottleneck in many applications that require frequent data transfer between CPUs and GPUs.

This is not the case for our implementation because most of our data is small enough to fit on the dedicated memory of individual GPUs. The only times we have to use the PCI Express channel is when loading input data on the GPUs and when transferring data between multiple GPUs. In the latter case, where data has to be copied to another GPU, the source GPU has to first copy the data to the main memory of the CPU and only then can the CPU can copy that data to another GPU. With newer GPUs such as Nvidia Tesla K20c, however, a direct data transfer between the GPUs, called peer-to-peer, is allowed to facilitate memory transfer more efficiently.

For our multi-GPU implementation of the bisection algorithm, we will be using CUDA, a parallel computing platform developed by Nvidia for general-purpose computing on GPUs [18]. In CUDA programming architecture, a CPU is referred to as a host as it launches a kernel on the GPU which is called the device. Once the kernel is completed, the device copies back the results back to the host. The kernel is run by means of threads organized into blocks. Each block retains a copy of the kernel, and the threads in the same block run the same code in parallel. Threads have their own local memory and have access to a shared memory space dedicated for the threads of the same block.

### 3.2 Single-GPU Implementations

Lesig [16] presented a single GPU implementation for the bisection algorithm. In this implementation, at every division stage, two kernels are launched: one for intervals containing exactly 1 eigenvalue, another for intervals containing more than 1. Their work was based on Tesla C870 with the peak throughput of 500 Gflops. Tesla K20c, on which this work is based, achieves 1.17 Tflops for double precision, and 3.52 Tflops for single precision.

Thus, by terminating the first kernel and reusing the kernel for all non-empty intervals, we may achieve additional speedup. In addition, the number of CUDA cores was also increased to 2,496 on the Tesla K20c. These improvements lead to significant increase in throughput to a point where the overhead of running to separate kernels is no longer justifiable. Therefore, all implementations in this paper will not group non-empty intervals based on the number of eigenvalues they contain.

Note that the Count($T, \lambda$) function is called at each level of the bisection tree by each child interval. It has been shown in [15] that 90% of the computation time is spent on the Count($T, \lambda$) functions for $n > 100$. Hence, a hybrid approach was proposed where a part of the function calls for the Count($T, \lambda$) function run on CPUs and part of them on GPUs. However, in order to exploit the full capacity of multiple GPUs, we will consider a GPU-only approach.

### Algorithm 2 GPU-Based Bisection

```plaintext
1: procedure BISECT(S)
2: Let $S$ be the collection of the four arrays:
3: s_left, s_right, s_left_count, s_right_count
4: tid ← id of the current thread
5: for each $I$ in $S$ do
6: Divide $I$ into $I_1$ and $I_2$
7: if $I_1$ is not empty then
8: Store $I_1$ in $S[tid]$
9: Store $I_2$ in $S[tid + N]$
10: else
11: Store $I_2$ in $S[tid]$
12: end if
13: end for
14: Perform scan-compaction on $S$
15: return $S$
16: end procedure
```

### 3.3 Multi-GPU Implementation

A simple approach to a multi-GPU implementation is to take advantage of the inherent parallelism in the division stage. In this step, even though two kernels run one after another, they operate on different groups of intervals and, therefore, can be parallelized. Afterwards, we run these two kernels concurrently on two separate GPUs and gain additional performance. We start with the Gerschgorin interval which can be determined on a GPU, similar to the single-GPU implementation.
The biggest overhead in this approach compared to the single-GPU implementation is data transfer. Unlike single-GPU versions, we must transfer data between the GPUs during the transition at the division stage. To minimize this communication overhead, we can take advantage of the way multi-GPU systems are designed. Data transfer can be done via a peer-to-peer manner, which is much faster than moving the data from device to host and then back to device. Though this approach may achieve a significant speedup, this implementation is limited to only two GPUs. In addition, as mentioned earlier, two kernels launches are unnecessary in most cases.

We now present a scalable approach that will ensure an even workload among multiple GPUs. In the case of bisection, workload has a direct dependency on the number of eigenvalues a device has to compute. Therefore, ensuring a balanced workload means ensuring that each device has to compute about the same number of eigenvalues.

As in the case of single-GPU systems, the first step in the bisection algorithm for multi-GPUs is calculating Gerschgorin interval of a given matrix. An intuitive approach for multi-GPUs is evenly dividing the computed Gerschgorin interval among the available GPUs. But this does not ensure a balanced workload because the spectrum of most matrices does not follow a uniform distribution. Some of those child intervals will likely contain more eigenvalues than the others. Hence, this approach will introduce load imbalance among the GPUs.

So, the approach we take is one that builds upon our algorithm for a single GPU system. Our implementation for a single GPU case begins by launching a bisection kernel to further process the Gerschgorin interval. This pre-processing step takes a constant time which involves bisecting the Gerschgorin interval until the number of child intervals exceeds the maximum number of threads available per block. Our multi-GPU implementation also begins with the same step as described below in more detail.

3.3.1 Pre-processing

In this step, we start with the Gerschgorin interval and keep bisecting it until the number of eigenvalues exceeds the maximum number of threads per block of the CUDA device. This step involves launching a kernel with a grid size of only one block. The results of the bisection are smaller intervals with the known number of eigenvalues. We then distribute these intervals using four arrays that are all stored in the shared memory of the single block that we launch on this kernel.

The left and right bounds of the intervals are, respectively, stored on the s_left and s_right arrays while the result of the Count(T, λ) function at those bounds are, respectively, stored in the s_left_count and s_right_count arrays. Note that all four arrays must be sorted. See Algorithms 2 and 3 for details.

3.3.2 Blocking Intervals

The second step which also takes place within the same kernel can be understood as grouping intervals from the pre-processing step into block. This involves grouping intervals so that the number of eigenvalues in each block will not exceed the maximum number of threads available in each block. As observed in [16], the blocking of the intervals is a particular instance of the knapsack problem that is known to be NP-hard.
Algorithm 3 Pre-processing
1: Let $S$ be the collection of the four arrays:
2: s_left, s_right, s_left_count, s_right_count
3: maxThreads $\leftarrow$ MAX_THREADS_BLOCK
4: $[G_l, G_h]$ $\leftarrow$ computed Gerschgorin interval
5: $N$ $\leftarrow$ num of intervals
6: Store $[G_l, G_h]$ in $S$
7: $N = 1$
8: while $N \leq$ maxThreads do
9: $S = \text{BISECT}(S)$
10: $N = \text{number of intervals in } S$
11: end while

To this end, we use a greedy algorithm that involves scanning the four arrays. To illustrate this, we assume that there are a maximum of eight threads per block. Let $C_\Lambda$ represent the number of eigenvalues in an interval or a block. As can be seen in Figure 1, our algorithm does not give the optimal solution for interval blocking, but the blocks are still constructed so that the computing power of the GPU is utilized efficiently. The result of the blocking is stored in another shared array called $s\_blocks$. For this example, $s\_blocks = [0, 3, 5, 6, 9]$. Algorithm 4 gives a detailed description of this step.

Algorithm 4 Blocking intervals
Let $S$ be the collection of the four arrays:
1: s_left, s_right, s_left_count, s_right_count
2: size $\leftarrow$ size of current block
3: $M \leftarrow$ number of blocks
4: for each $I$ in $S$ do
5: if $size + \text{count}(I) \leq$ maxThreads then
6: $size = size + \text{count}(I)$
7: else
8: $M = M + 1$
9: $s\_block[M] = \text{index of } I \text{ in } S$
10: $size = \text{count}(I)$
11: end if
12: end for

3.3.3 Distribution of Intervals
The results from the previous step are then written to global memory of the GPU and then copied to the host memory. To this end, CPU threads are created for each GPU. Each thread will determine the blocks to be assigned to the GPU based on the number of available GPUs.

It is important to note that the number of blocks assigned to each GPU is roughly the same. If the number of blocked intervals is not evenly divisible by the number of GPUs, the first GPUs will be assigned more blocks. Once the number of block assigned to the GPU is determined, all the results from the interval blocking step are copied to each GPU.

A kernel is then launched on each GPU with a grid size corresponding to the number of blocked intervals assigned to each GPU. For example, if there are only two GPUs, GPU 0 will be assigned block 0 and block 1, and GPU 1 will be assigned block 2 and block 3. Each of these kernels will run the bisection algorithm, described in Section 2, until the desired precision level is reached. See Figure 2 for details.

3.3.4 Collecting Computed Eigenvalues
At this stage, all the converged eigenvalues are copied back to the host CPU. All local lists of the eigenvalue from each GPU are combined to generate a global list containing all the eigenvalues. The whole reduction is done in parallel and takes time that is proportional to the logarithm of the number of blocks per grid.

Finally, the whole procedure on multi-GPU is given in the following algorithm:

Algorithm 5 Bisection on multi-GPUs
Let $S$ be the collection of the four arrays:
1: s_left, s_right, s_left_count, s_right_count
2: $S_{\text{multi}}$ be each GPU’s version of $S$
3: $K \leftarrow$ number of GPUs
4: $M \leftarrow$ number of blocks
5: for $i$ to $K - 1$ do
6: $\text{start} = s\_blocks[(i + 1) * \frac{M}{K} - 1]$
7: $\text{end} = s\_blocks[i * \frac{M}{K}]
8: $S_{\text{multi}} = S[\text{start:end}]
9: \text{repeat}
10: $S_{\text{multi}} = \text{BISECT}(S_{\text{multi}})$
11: until all intervals converge
12: end for

4. Experimental Results and Discussion
Our CUDA implementation of the bisection algorithm was tested on a machine which comprises a dual Intel Xeon E5-2620 CPUs with 64 GB main memory, and four NVIDIA Tesla K20c GPU, each with 5 GB global memory.

Table 1 shows the actual running time and the speedup of our multi-GPU implementation with random matrices of various size. Clearly, our multi-GPU implementation outperformed the full configuration of the CPU, which comprises 16 cores. When $n$ is large, our implementation runs up to 57x faster than that of CPU with 16 cores. We observe a more dramatic speedup when $n > 16,384$. This is because with large $n$ the global GPU memory becomes saturated with the entries of $T$ and temporary data required by the division steps.

It also shows how much faster the implementation is on multiple GPUs compared to one GPU. Our implementation achieves almost linear speedup with respect to the number of
Table 1: Performance Results on Random Matrices

<table>
<thead>
<tr>
<th>( n )</th>
<th>16 CPUs</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
<th>16 CPUs / 4 GPUs</th>
<th>1 GPU / 2 GPUs</th>
<th>1 GPU / 4 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.1</td>
<td>0.045</td>
<td>0.049</td>
<td>0.052</td>
<td>2.08</td>
<td>0.92</td>
<td>0.86</td>
</tr>
<tr>
<td>2048</td>
<td>0.6</td>
<td>0.088</td>
<td>0.094</td>
<td>0.098</td>
<td>5.75</td>
<td>0.93</td>
<td>0.90</td>
</tr>
<tr>
<td>4096</td>
<td>2.1</td>
<td>0.181</td>
<td>0.194</td>
<td>0.198</td>
<td>10.83</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>8192</td>
<td>8.7</td>
<td>0.575</td>
<td>0.398</td>
<td>0.416</td>
<td>20.81</td>
<td>1.44</td>
<td>1.38</td>
</tr>
<tr>
<td>16384</td>
<td>35.1</td>
<td>1.65</td>
<td>1.193</td>
<td>0.782</td>
<td>44.77</td>
<td>1.38</td>
<td>2.11</td>
</tr>
<tr>
<td>32768</td>
<td>133.6</td>
<td>6.23</td>
<td>3.382</td>
<td>2.338</td>
<td>57.16</td>
<td>1.84</td>
<td>2.66</td>
</tr>
</tbody>
</table>

Table 2: Performance Results on Uniform Matrices

<table>
<thead>
<tr>
<th>( n )</th>
<th>16 CPUs</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
<th>16 CPUs / 4 GPUs</th>
<th>1 GPU / 2 GPUs</th>
<th>1 GPU / 4 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.12</td>
<td>0.037</td>
<td>0.037</td>
<td>0.041</td>
<td>2.97</td>
<td>1.00</td>
<td>0.92</td>
</tr>
<tr>
<td>2048</td>
<td>0.38</td>
<td>0.074</td>
<td>0.077</td>
<td>0.080</td>
<td>4.73</td>
<td>0.95</td>
<td>0.92</td>
</tr>
<tr>
<td>4096</td>
<td>1.33</td>
<td>0.146</td>
<td>0.152</td>
<td>0.156</td>
<td>8.52</td>
<td>0.96</td>
<td>0.94</td>
</tr>
<tr>
<td>8192</td>
<td>5.15</td>
<td>0.523</td>
<td>0.302</td>
<td>0.306</td>
<td>16.84</td>
<td>1.73</td>
<td>1.71</td>
</tr>
<tr>
<td>16384</td>
<td>20.32</td>
<td>1.936</td>
<td>1.569</td>
<td>0.607</td>
<td>33.48</td>
<td>3.40</td>
<td>3.19</td>
</tr>
<tr>
<td>32768</td>
<td>77.54</td>
<td>6.380</td>
<td>2.114</td>
<td>1.143</td>
<td>67.84</td>
<td>3.02</td>
<td>5.58</td>
</tr>
</tbody>
</table>

GPUs, demonstrating the scalability of our implementation on multi-GPU systems.

In addition, when the eigenvalues are uniformly distributed, we achieve a superlinear speedup as shown Table 2. This can be explained by the relationship between matrix size and the number of multiprocessors available on the K20c GPUs. We see that for matrix size up to 4,096, the relationship between matrix size and run time is linear. However, for matrices that are larger than that, the relationship becomes almost quadratic. A possible explanation for this is that when the matrix size is 4,096, the GPU’s computing capability is fully exhausted. Hence, for matrix sizes larger than this threshold, not all eigenvalues can be computed in parallel. A similar phenomenon exists in the case of two GPUs. For the matrices smaller than 16,384, the relationship is linear, but for larger matrices, the speedup becomes almost quadratic.

Our implementation of the bisection algorithm in this paper, however, has not taken full advantage of hardware and software features that GPUs have to offer. We are considering a hybrid approach that uses both CPU and GPU to speed up the bisection algorithm even further. For example, the Gerschgorin interval can be preprocessed on the CPU. Furthermore, CPU can employ multi-section algorithm instead of bisection, which means dividing a big interval into multiple smaller intervals instead of only two ones. The optimal number of subintervals in a multi-section algorithm remains to be found.

Future work should also be able to accommodate out-of-core cases. These situations may occur with very large matrices because symmetric tridiagonal matrices are represented by only two vectors for diagonal and off-diagonal elements. Parallel out-of-core version of the bisection method should be an interesting set of problems for creative solutions.

5. Acknowledgements

The authors would like to thank CUDA Teaching Center Program, Nvidia Research and Student Research Program, Trinity College for supporting this research. The authors would also like to thank Nam Thai of Trinity College for his contribution to a preliminary version of our implementation.

References