U.P.B. Sci. Bull., Series A, Vol. 74, Iss. 3, 2012 ISSN 1223-7027

OPTIMIZING THE COMPUTATION OF EIGENVALUES USING GRAPHICS PROCESSING UNITS

Ion LUNGU[1], Alexandru PÎRJAN[2], Dana-Mihaela PETROŞANU[3]

În această lucrare descriem mai întâi, pe scurt, câteva aspecte matematice referitoare la calcularea valorilor proprii, urmate de o abordare originală: un algoritm de bisecție util în calculul valorilor proprii pentru matrici tridiagonale simetrice de dimensiuni arbitrare, folosind capabilitățile computaționale ale celor mai noi unități de procesare grafică ce înglobează arhitectura de calcul paralel Compute Unified Device Architecture. Originalitatea abordării din această lucrare constă în optimizarea implementării algoritmului, bazată pe îmbunătățirea gestionării memoriei partajate şi pe eficientizarea algoritmului de scanare. Astfel, dezvoltatorii de aplicații pot folosi puterea imensă de calcul paralel oferită de unitățile de procesare grafică de ultimă generație.

In this paper, we first briefly describe some mathematical aspects regarding the computation of eigenvalues, followed by an original approach: a bisection algorithm useful in computing eigenvalues for a tridiagonal symmetric matrix of arbitrary size, using the computing capabilities of the latest graphics processing units that incorporate the Compute Unified Device Architecture. The novel approach developed in this paper relates to an optimized algorithm's implementation, based on the improvement of the shared memory management and on the increased efficiency of the scan algorithm. Thus, developers can use the huge parallel computational power offered by the Compute Unified Device Architecture.

Keywords: eigenvalues, tridiagonal symmetric matrix, graphics processing units, Compute Unified Device Architecture.

1. Introduction

The determination of all the eigenvalues and eigenvectors of a matrix is an extremely important issue in linear algebra, statistics, physics, engineering and many other fields. This is often used in common applications such as stability analysis, physics of rotating bodies and small oscillations of vibrating systems. This paper describes the implementation of a bisection algorithm useful in

[1]Professor, Economic Informatics Department, Academy of Economic Studies, Bucharest, Romania, e-mail: ion.lungu@ie.ase.ro
[2]Teaching Assistant, IT, Statistics and Mathematics Department, Romanian-American University, Bucharest, Romania, e-mail: alex@pirjan.com
[3]Lecturer, Department of Mathematics-Informatics I, University POLITEHNICA of Bucharest, Romania, e-mail: danap@mathem.pub.ro

computing all the eigenvalues for a tridiagonal symmetric matrix of arbitrary size using the capabilities of graphics processing units (GPUs) that are based on the parallel Compute Unified Device Architecture (CUDA) developed by NVidia.

In the following, we first intend to mention a few notations and describe some mathematical aspects regarding the computation of eigenvalues [1], [2]. In the following we use Householder's notations: scalars are denoted by lowercase letters (a, b, c or α, β, γ); vectors are considered by default as being column vectors and are denoted with lowercase bold letters $\boldsymbol{a} = (a_i)_{i=\overline{1,n}}$; square matrices with real elements are denoted by bold uppercase roman letters $\boldsymbol{A} \in \mathcal{M}(n,n), \boldsymbol{A} = (a_{ij})_{i,j=\overline{1,n}}$; the transpose of the $\boldsymbol{A}$ matrix is denoted $\boldsymbol{A^T}$. This paper refers especially to tridiagonal symmetric matrices with real elements, denoted by $\boldsymbol{T}$ and characterized by the fact that only its first, lower and upper diagonals are non-zero: $\boldsymbol{T} = (a_{ij})_{i,j=\overline{1,n}}, a_{ij} = a_{ji}, a_{ij} = 0\ if\ |i-j| \geq 2, \forall i,j = \overline{1,n}$.

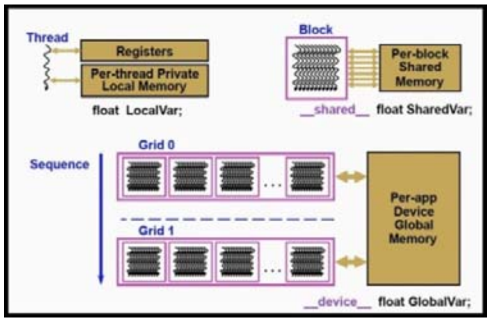
For a square matrix with real elements $\boldsymbol{A} = (a_{ij})_{i,j=\overline{1,n}}$, we denote an eigenvalue by λ**,** an eigenvector by $\boldsymbol{u}$**,** the spectrum of $\boldsymbol{A}$ by $\lambda(\boldsymbol{A})$. An interesting result for this study is the fact that all the eigenvalues for a symmetric matrix are real [3].

In contrast with traditional data processing performed by central processing units (CPUs), a general-purpose graphics processing unit (GPGPU) represents a new concept. By increasing the clock speed and the number of processor cores, central processing units (CPUs) and graphics processing units (GPUs) have evolved over time. CUDA is a hardware architecture that introduces a new parallel programming model that uses the computational power of GPUs to solve complex computational problems more efficiently than through traditional central processing units. The CUDA architecture provides a software development environment that allows developers to use high-level language C.

The CUDA parallel programming model was designed to offer to the developers the necessary tools for designing scalable parallel applications in a standard programming language like C [4]. In the future, CUDA will provide support for other programming languages and interfaces such as C++, OpenCL and DirectX Compute.

2. The compute unified device architecture (CUDA)

In the following, we describe the main features of the Compute Unified Device Architecture (CUDA). The CUDA programming environment provides three levels of abstraction [5]: a hierarchy of thread groups, the shared memory and the barrier synchronization (Fig. 1). These abstractions, available to the developer as a minimal set of extensions in the C language, provide fine-grained parallelism for data and threads, associated with large grained parallelism for data and tasks. Using these abstractions, the developer partitions the problem into subproblems of small sizes that can be solved in parallel. Such an approach allows threads to cooperate when solving each sub-problem and also provides scalability since each sub-problem can be solved by any of the available processing cores. Consequently, a CUDA C compiled program can be executed on any number of processing cores.


Thread
Registers
Per-thread Private Local Memory
float LocalVar;
Block
Per-block Shared Memory
__shared__ float SharedVar;
Sequence
Grid 0
Grid 1
Per-app Device Global Memory
__device__ float GlobalVar;
Fig. 1. NVidia Compute Unified Device Architecture (CUDA)

Using CUDA, the latest NVidia GPUs effectively become open architectures like CPUs and thus, the latest graphics processing units become accessible for general-purpose computations that have been previously possible only on central processing units (CPUs). However, unlike a CPU, a GPU has parallel "many-core" architecture, each core being capable of running thousands of threads simultaneously - if an application is suited to this kind of an architecture, the GPU can offer large performance benefits. This approach of solving general-purpose problems on GPUs is known as General-Purpose computation on Graphics Processing Units (GPGPU).

A CUDA program calls parallel program kernels. A set of parallel threads is executed in parallel by the kernel. The programmer or compiler organizes these threads into thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid containing parallel thread blocks. Each thread from the block executes an instance of the kernel, has a unique ID associated to its registers and its private memory within the thread block [5].

In the CUDA programming model, when algorithms are being developed, the developers' most important concern is to divide the required work in fragments that can be processed in parallel by a number of thread blocks, each containing more threads. In order to avoid the execution of threads within a block by multiple cores within a streaming multiprocessor, it is recommended that the number of thread blocks match the number of processors. The most important factor in achieving performance is the repartition of tasks that have to be performed between the thread blocks.

The newest NVidia's architecture is called Fermi and became commercially available on March 26, 2010. This architecture is implemented in the GeForce GTX400 series and it features 16 SMs (streaming multiprocessors), each of them having 32 SPs (streaming processors, also called CUDA cores in the Fermi architecture) and 64 KB shared memory that is configurable as larger shared memory or larger L1 cache (48/16 KB or 16/48 KB). The total amount of SPs is 512 and the whole GPU shares a L2 cache of 768 KB. Fermi offers eight times faster double precision performance, IEEE 754-2008 FP precision and error correcting code (ECC) memory, necessary for consistency requirements of scientific computing [5]. A comparison between different CUDA architectures is depicted in [6]. This architecture offers a high degree of flexibility when it comes to allocate local resources like registers or local memory in threads. The programmer divides local resources among threads and every CUDA core can process a variable number of threads. Although this flexibility offers a high degree of control over an application's performance, it has also a great impact on optimizing the performance of computations. Another important aspect is related to how the GeForce GTX480 can execute applications and what are the elements that improve or limit its performance. Numerous software applications have been ported and evaluated on the CUDA platform as a result of its huge data processing power [7].

3. The Gerschgorin circle theorem and the computation of the inertia for a tridiagonal symmetric matrix

In the following, we depict some mathematical results that are used later, in order to establish a bisection algorithm for computing all eigenvalues in the case of a tridiagonal symmetric matrix with real elements.

The first important result for our study is the Gerschgorin Circle Theorem. If $\boldsymbol{A} \in \mathcal{M}(n,n)$ is a symmetric matrix with real elements, $\boldsymbol{Q} \in \mathcal{M}(n,n)$ an orthogonal matrix with real elements, $\boldsymbol{Q^T A Q}=\boldsymbol{D}+\boldsymbol{F}$ with $\boldsymbol{D} \in \mathcal{M}(n,n), \boldsymbol{D} = (d_{ij})_{i,j=\overline{1,n}}, d_{ii} = d_i, d_{ij} = 0 \; if \; |i-j| \geq 1, \forall i,j = \overline{1,n}$ a diagonal matrix and $\boldsymbol{F} \in \mathcal{M}(n,n), \boldsymbol{F} = (f_{ij})_{i,j=\overline{1,n}}, f_{ii} = 0 \; \forall i = \overline{1,n}$ a matrix with zero diagonal entries, then the spectrum of $\boldsymbol{A}$,

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

where $r_i = \sum_{j=1}^{n} |f_{ij}|, \forall i = \overline{1,n}$ [3].

If we consider a tridiagonal symmetric matrix with real elements:

$$\boldsymbol{T} = (a_{ij})_{i,j=\overline{1,n}}, a_{ii} = a_i, \forall i = \overline{1,n}, a_{ii+1} = b_i, \forall i = \overline{1,n-1},$$
$$a_{ij} = a_{ji}, a_{ij} = 0 \text{ if } |i-j| \geq 2, \forall i,j = \overline{1,n}$$

we obtain a corollary of the above mentioned theorem. The spectrum of $\boldsymbol{A}$ verifies:

$$\lambda(\boldsymbol{A}) \subseteq \bigcup_{i=1}^{n}[a_i - r_i, a_i + r_i] \tag{2}$$

where $r_1 = b_1, r_i = b_i + b_{i-1}, \forall i = \overline{2, n-1}, r_n = b_{n-1}$ [7].

Another interesting result establishes a relationship between the eigenvalues of a square matrix $\boldsymbol{A}$ and those of a shifted version of $\boldsymbol{A}$

$$\boldsymbol{A}_\mu = \boldsymbol{A} - \mu \boldsymbol{I} \tag{3}$$

where $\mu \in \mathbb{R}$ is a parameter called shift index. If $\lambda_i, i = \overline{1, n}$ are the eigenvalues of the matrix $\boldsymbol{A}$, then the matrix $\boldsymbol{A}_\mu$ has the eigenvalues $\mu_i = \lambda_i - \mu$, $i = \overline{1, n}$ and the eigenvectors of $\boldsymbol{A}$ and $\boldsymbol{A}_\mu$ are identical. The proof of this theorem is based on the characteristic polynomial

$$det(\boldsymbol{A} - \lambda_i \boldsymbol{I}) = 0, \forall i = \overline{1, n} \tag{4}$$

Using (3) and (4) we obtain:

$$det\big((\boldsymbol{A}_\mu + \mu \boldsymbol{I}) - \lambda_i \boldsymbol{I}\big) = 0, \forall i = \overline{1, n} \tag{5}$$

and this relation is equivalent with

$$det\big(\boldsymbol{A}_\mu + (\mu \boldsymbol{I} - \lambda_i)\boldsymbol{I}\big) = 0, \forall i = \overline{1, n} \tag{6}$$

Consequently, denoting by $\mu_i = \lambda_i - \mu$, $i = \overline{1, n}$, we obtain that μ_i are the eigenvalues of the matrix $\boldsymbol{A}_\mu$. If we arrange the spectrum of a the square matrix A in a diagonal matrix $\boldsymbol{\Lambda}$, we obtain

$$(\boldsymbol{A} - \mu \boldsymbol{I})\boldsymbol{U} = (\boldsymbol{\Lambda} - \mu \boldsymbol{I})\boldsymbol{U} \tag{7}$$

and therefore, the eigenvectors of $\boldsymbol{A}$ and $\boldsymbol{A}_\mu$ are identical. This result is the starting point for a wide class of eigenvalue algorithms as the shift does not change the eigenvectors but translates the eigenvalues by the shift index μ. In many situations, it is more efficient to compute first the eigenvalues of the shifted matrix and then, using the above obtained result, to compute the eigenvalues of the original matrix.

For a square matrix $\boldsymbol{A}$, after computing its spectrum $\lambda(\boldsymbol{A})$, one can establish the number of negative elements in this spectrum, denoted by $n(\boldsymbol{A})$, the number of zeros contained in the spectrum of $\boldsymbol{A}$, denoted by $z(\boldsymbol{A})$ and the number of positive elements in the spectrum $\lambda(\boldsymbol{A})$, denoted by $p(\boldsymbol{A})$. The inertia of the square Hermitian matrix $\boldsymbol{A}$ is defined as the triple of positive integers $(n(\boldsymbol{A}), z(\boldsymbol{A}), p(\boldsymbol{A}))$. An interesting result for our study is the Sylvester's Law of Inertia [3] which states that if $\boldsymbol{A}$ is a tridiagonal symmetric, positive definite matrix and $\boldsymbol{X} \in \mathcal{M}(n, n)$ is a nonsingular matrix, then the inertia is the same for the matrices $\boldsymbol{A}$ and $\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{A}\boldsymbol{X}$.

In the following we introduce the notion of Cholesky decomposition (or Cholesky triangle), used in linear algebra. This is a decomposition of a positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. If $\boldsymbol{A}$ is a tridiagonal symmetric, positive definite matrix, then $\boldsymbol{A}$ can be decomposed as:

$$\boldsymbol{A} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^{\boldsymbol{T}} \tag{8}$$

where $\bar{\boldsymbol{L}}$ is a lower triangular matrix with strictly positive diagonal entries, and $\bar{\boldsymbol{L}}^{\boldsymbol{T}}$ denotes the transpose of $\bar{\boldsymbol{L}}$. This is the Cholesky decomposition, which is unique [2].

Based on this notion, an interesting theorem can be obtained. If $\boldsymbol{T}$ is a non-degenerated tridiagonal symmetric, positive definite matrix, then there are two matrices, $\boldsymbol{D}$ diagonal and $\boldsymbol{L}$ lower bidiagonal with all diagonal elements being 1, and there exists a factorization of $\boldsymbol{T}$,

$$\boldsymbol{T} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\boldsymbol{T}} \tag{9}$$

The proof of this statement is based on the Cholesky factorization $\boldsymbol{T} = \bar{\boldsymbol{L}}\bar{\boldsymbol{L}}^{\boldsymbol{T}}$, [2]. Using $\bar{\boldsymbol{L}} = \boldsymbol{L}\boldsymbol{D}^{\boldsymbol{1/2}}$, the existence of the factorization of T as in (9) follows immediately.

As a corollary of this theorem, for a tridiagonal symmetric matrix $\boldsymbol{T} \in \mathcal{M}(n,n)$ with real components, one can compute the inertia first by determining the factorization (9) of $\boldsymbol{T}$ and then counting the positive and negative eigenvalues of $\boldsymbol{D}$, the diagonal matrix which appears in the decomposition (9).

4. The eigenvalue count for a tridiagonal symmetric matrix

In the following, we introduce the notion of eigenvalue count. Considering a tridiagonal symmetric matrix $\boldsymbol{T} \in \mathcal{M}(n,n)$ with real components, a real number x, the eigenvalue count of $\boldsymbol{T}$ is denoted $C(x)$ and represents the number of eigenvalues of $\boldsymbol{T}$, satisfying the condition:

$$C(x) = card\, \lambda_x(\boldsymbol{T}), \lambda_x(\boldsymbol{T}) = \{\lambda \in \lambda(\boldsymbol{T}) | \lambda < x\} \tag{10}$$

Similarly, the eigenvalue count of an interval $(x_1, x_2]$, denoted by $C(x_1, x_2)$ is defined by:

$$C(x_1, x_2) = C(x_2) - C(x_1) \tag{11}$$

If $C(x_1, x_2) = 0$, the interval $(x_1, x_2]$ is called empty.

Using the above-mentioned corollary, the eigenvalue count $C(x)$ of $\boldsymbol{T}$ can be obtained applying the following steps:

1. Using $\boldsymbol{T}$ and x, we first compute the shifted version of $\boldsymbol{T}$, $\boldsymbol{T}_x = \boldsymbol{T} - x\boldsymbol{I}$.
2. For the shifted version $\boldsymbol{T}_x$ we compute the $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\boldsymbol{T}}$ factorization, as in (9).
3. Once obtained the diagonal matrix $\boldsymbol{D}$, we can count its negative elements.

Actually, the computation of the full $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\boldsymbol{T}}$ factorization is not necessary, but only the signs of those elements of $\boldsymbol{D}$ that are non-zero.

In [8] an algorithm is proposed which is useful for obtaining the eigenvalue count in an efficient way, without the computation of the full $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\boldsymbol{T}}$ factorization and avoiding the storage of the full vector containing diagonal elements of the matrix $\boldsymbol{D}$. We will refer later at this algorithm as A1. As input there are considered the elements of the tridiagonal symmetric matrix $\boldsymbol{T}$ with real elements: the diagonal and upper diagonal elements $a_{ii} = a_i, \forall i = \overline{1,n}, a_{ii+1} = b_i, \forall i = \overline{1,n-1}$ and we define $b_n = 0$. Also the real number x, which is used as a shift, is considered as an input. The output is the eigenvalue count of $\boldsymbol{T}$ denoted by $C(x)$ and is obtained by using the following steps:

1. The initialization: $count = 0$
2. The initialization: $d_0 = 1$
3. For every index $i = 1{:}n$ we compute $d_i = a_i - x - \frac{b_i * b_i}{d_{i-1}}$. If the computed value d_i is negative, then the counter variable is incremented $count = count + 1$.

5. An algorithm for computing the eigenvalues of a tridiagonal symmetric matrix

Using the previous results, in the following we depict an algorithm [9] for computing the eigenvalues of a tridiagonal symmetric matrix with real components. Considering a tridiagonal symmetric matrix $\boldsymbol{T} \in \mathcal{M}(n,n)$ with real components and applying the Gerschgorin circle theorem for symmetric, tridiagonal matrices, as in the final part of section 2, we obtain the Gerschgorin interval of $\boldsymbol{T}$, which we note by $I_{00} = (\alpha_{00}, \beta_{00}]$. The eigenvalue count for this interval is computed using the A1 algorithm, depicted in the previous section.

This interval can be divided in two smaller intervals of the same length denoted by $I_{10} = (\alpha_{10}, \beta_{10}]$ and $I_{11} = (\alpha_{11}, \beta_{11}]$ where, obviously, $\alpha_{10} = \alpha_{00}$, $\beta_{10} = \alpha_{11}$, $\beta_{11} = \beta_{00}$, $I_{00} = I_{10} \cup I_{11}$ and $I_{10} \cap I_{11} = \phi$. Applying the A1 algorithm depicted in the previous section we compute the eigenvalues count for these two intervals, $C(\alpha_{10}, \beta_{10}) = s$ and $C(\alpha_{11}, \beta_{11}) = n - s$. Using these two smaller intervals, we obtain for the eigenvalues better bounds than we had obtained before when using the first interval. The indices used to denote the intervals have the following meaning: the first index denotes the level of bisection and the second index is used for counting the obtained interval at that level. The process continues and each of the two previously built intervals is split in two intervals, determining another improvement of the eigenvalues bounds. The process continues recursively and through this method, we obtain approximations with a desired accuracy for the eigenvalues. If one of the generated intervals is empty, its subdivision is stopped as it will no longer be interesting for our study. Non-empty intervals are split until we obtain approximations of the eigenvalues according to the desired accuracy. After every split only the intervals that have a strictly positive eigenvalue count are kept.

As a result of the subdivision process we obtain a tree having the Gerschgorin interval of $\boldsymbol{T}$ as a root and in the nodes of this tree there are the subdivision intervals $I_{ij} = (\alpha_{ij}, \beta_{ij}]$ obtained through the above described method. The index i represents the level of the tree and the index j is used for counting the obtained intervals at level i. The external nodes could be empty intervals if $C(\alpha_{ij}, \beta_{ij}) = 0$ or converged intervals having a size smaller than a previous established accuracy.

The process of computing the eigenvalues of a tridiagonal symmetric matrix can be summarized through the following algorithm, denoted A2. The input consists in the elements of the tridiagonal symmetric matrix $\boldsymbol{T}$ with real elements and the desired accuracy ε. The output is the spectrum $\lambda(\boldsymbol{T})$ of the matrix $\boldsymbol{T}$ and is determined through the following steps:

1. Compute the Gerschgorin interval of $\boldsymbol{T}$, $I_{00} = (\alpha_{00}, \beta_{00}]$.

2. Divide the interval in two smaller equal intervals $I_{10} = (\alpha_{10}, \beta_{10}]$ and $I_{11} = (\alpha_{11}, \beta_{11}]$ where, obviously, $\alpha_{10} = \alpha_{00}$, $\beta_{10} = \alpha_{11}$, $\beta_{11} = \beta_{00}$, $I_{00} = I_{10} \cup I_{11}$ and $I_{10} \cap I_{11} = \phi$.

3. For each of the obtained intervals a number is chosen within the interval, noted by $\gamma_{ij} \in (\alpha_{ij}, \beta_{ij}]$. Using A1, the eigenvalue count of the considered interval, $C(\gamma_{ij})$ is computed.

4. If $C(\gamma_{ij}) = 0$, then the corresponding interval is empty and this interval will not be split anymore. Otherwise, if $C(\gamma_{ij}) \neq 0$, the steps 2-4 are repeated, keeping only non-empty intervals, until the approximations for the eigenvalues are obtained, according to the desired accuracy.

5. All the obtained eigenvalues are saved in a list.

6. The implementation of the algorithm for computing eigenvalues using CUDA

In the following, we will implement the A2 algorithm from section **5** for computing eigenvalues using the CUDA data parallel programming model [5]. The implementation is based on the scan algorithm (also known as prefix sum) [10].

Usually, when one intends to parallelize an algorithm in order to implement it efficiently on a parallel data architecture, the first necessary step is to identify the perfect candidates that can be processed in parallel: all the computations that are independent and similar for a large set of data. In our case, in the first algorithm, when the eigenvalue count is computed, similar computations are performed for all the elements of the input tridiagonal symmetric matrix. Every iteration of the algorithm updates the variable d and thus, every step of the algorithm is linked to the previous one. Therefore, this algorithm is computed serially and the eigenvalue count computation cannot be parallelized. In the A2 algorithm depicted above, a parallelization can be obtained if the steps 3-5 are performed in parallel for all the intervals $I_{ij} = (\alpha_{ij}, \beta_{ij}]$ at the level j.

When implementing a CUDA data parallel programming model, one of the most important issues is the threads allocation. From the available options, the programmer can choose one or another taking into account different criteria (efficiency, costs, the available devices and the set of data). As an example, in the studied problem it is possible to process multiple intervals from the set I_{ij} using the same thread or using for each interval a different thread, which is more suitable when dealing with a data parallel architecture. The Compute Unified Device Architecture (CUDA) programming model does not offer the possibility to dynamically allocate or create threads on the device and thus it needs a prior decision and specification referring to threads allocations. Therefore it is of paramount importance to choose a suitable number of threads when the kernel program is launched, corresponding to the maximum number of parallel processed intervals. If the desired accuracy is chosen smaller than the minimum distance between any two eigenvalues, then the number of allocated threads must be equal to the number of eigenvalues of the input tridiagonal symmetric matrix. For the first levels of the tree processed by the computation algorithm, many threads are inactive until a higher level of the tree has been reached. In that moment, an increased number of threads are exploited. This means that the parallelism is fully exploited after the first iterations of the algorithm [9].

In the following, we address several issues related to the intervals' subdivision. In the 3-rd step of the algorithm depicted in section 5, for each of the split intervals, a number within the interval has been chosen and denoted by $\gamma_{ij} \in (\alpha_{ij}, \beta_{ij}]$. In the literature, different approaches have been proposed in order to determine γ_{ij} [8], [11]. In this paper, we have preferred the midpoint subdivision, whose main advantage is that at a fixed level j of the tree, all the child intervals have the same size. Therefore, all the converged intervals are obtained at the same processing step and they lie on the same level of the tree. This approach improves the efficiency of the eigenvalues computation [5].

The above depicted algorithm for computing the eigenvalues of a tridiagonal symmetric matrix imposes the storage of a data set: the non-zero elements of the input matrix and the active intervals. For the input matrix, data is stored using two vectors that contain the diagonal and the first upper diagonal. For the intervals, the data stored consists in the left and right bound and the eigenvalue count for these bounds.

Taking into account the recommendations from the CUDA programming guide [5], in order to attain an optimal performance the data must be represented according to some conditions, the most important ones being: minimizing data transfers to global memory; avoiding non-aligned data transfers to global memory; using, whenever possible, the shared memory and registers for all computations. The global memory is much slower than the shared memory and so, we have used it only at the beginning of the computations (for loading the data) and at the end (for the storage of the results).

7. The parallel prefix sum algorithm

The implementation of the algorithm for computing the eigenvalues of a tridiagonal symmetric matrix using a data parallel programming model through CUDA is based on the scan algorithm (also known as prefix sum) [10]. In this paper, we propose several improvements of the scan algorithm based on the huge parallel computing power of the Fermi architecture.

The parallel prefix sum or parallel scan algorithm is a useful tool for sorting routines, for building data structures and for many parallel algorithms. Consequently, the efficient implementation of the parallel scan algorithm in CUDA improves the performance for all the algorithms that make use of the parallel prefix sum. By implementing the basic algorithm in CUDA, one can obtain many advanced techniques useful in parallel algorithms. A particular case is our algorithm for computing the eigenvalues using the Compute Unified Device Architecture.

The implementations of the prefix sum concept based on the modern parallel architectures are becoming increasingly important for developing efficient computation algorithms. In [12] it is mentioned that, even if the prefix sum seems to be an inherent situation of sequential computations, it can be efficiently parallelized. In the following, we introduce the definition of the prefix sum, its main properties and several improvements to the scan algorithm, based on the CUDA parallel computing architecture. These improvements are used in the above-mentioned algorithm for computing the eigenvalues using CUDA. There are two basic concepts regarding the prefix sum: the all-prefix-sums operation (inclusive scan) and the exclusive scan operation [12].

Let's consider $\ast$ a binary associative operator defined on the set of real numbers and $v = [a_0, a_1, \dots, a_{n-1}]$ an array of n real numbers. The inclusive scan operation associates to the array v the n-dimension array w_i, defined as follows:

$$w_i = [a_0, (a_0 \ast a_1), (a_0 \ast a_1 \ast a_2), \dots, (a_0 \ast a_1 \ast \dots \ast a_{n-1})] \qquad (12)$$

Some of the most common applications of the inclusive scan operation are linked to lexical analysis, sorting, stream compaction, polynomial evaluation, string comparison, building histograms, data structures (graphs, trees, etc.) in parallel [12]. The main advantage of using all-prefix-sums implementations is the fact that it converts a part of sequential computations in parallel equivalent ones.

The inclusive scan operation associates an array to another, for which each element k is the sum of all precedent elements including the k-element of the input array. This fact justifies the name of inclusive scan. In practice, another output array is often used: each element k is the sum of all precedent elements of the input array, except the k-element of the input array. This is the exclusive scan operation (or prescan) [12]. Let's consider $\otimes$ a binary associative operator defined on the set of real numbers, e the identity element for the binary operation, $v = [a_0, a_1, \dots, a_{n-1}]$ an array of n real numbers. The exclusive scan operation associates to the array v the array w_e of n elements, defined as follows:

$$w_e = [e, a_0, (a_0 \otimes a_1), (a_0 \otimes a_1 \otimes a_2), \dots, (a_0 \otimes a_1 \otimes \dots \otimes a_{n-2})] \qquad (13)$$

One can easily generate an exclusive scan output vector w_e starting from an inclusive scan, by inserting on the first position of the output vector the identity element for the binary operation and deleting the last component of the output vector w_i. Similarly, one can easily generate an inclusive scan output vector w_i starting from an exclusive scan. In the following we discuss the case of the exclusive scan.

In the sequential version of the exclusive scan, a single thread on a central processing unit (CPU) executes the computations. All the elements of the input array are processed in an ascendant order and the current element of the output array is computed as the sum of the previous element of the output array and the previous element of the input array. In this case, if the input array has the size n, the computation of the output array requires exactly n additions to be performed.

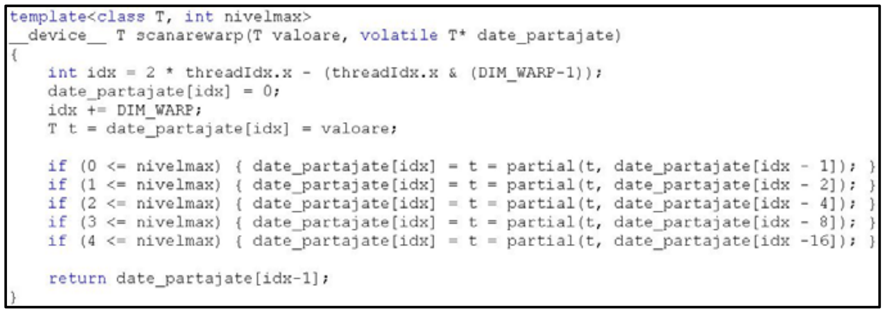
8. Managing shared memory bank conflicts, a solution for optimizing the parallel prefix sum algorithm in CUDA

In [10], some parallel versions of the exclusive scan computing method are depicted: a parallel scan implementation that is not work-efficient and an improved, work-efficient scan algorithm.

In the situation of the above-depicted algorithm for computing the eigenvalues using CUDA, we have used an efficient implementation of scan based on multiple blocks of threads, resulting in a significant speedup over a sequential implementation on a fast CPU. We have taken into account both the algorithm's and the hardware's efficiency. In order to make the parallel scan version efficient on NVidia GPU hardware, we have optimized the memory access patterns.

The overall performance has been significantly improved by managing the shared memory bank conflicts. In [5] the CUDA shared memory (used by the scan algorithm) is described as being composed of multiple banks (equally sized memory modules). Consecutive array accesses through consecutive threads are very fast as each memory bank holds a successive 32-bit value (e.g. a float variable). Multiple data requests from the same bank generate bank conflicts. The requests can originate from the same address or multiple addresses may map to the same bank. The hardware serializes the memory operations when the conflict occurs and this forces all the threads to wait until all memory requests are fulfilled [6]. Serialization is avoided if all threads read from the same-shared memory address, because a broadcast mechanism is automatically triggered. The broadcast mechanism is an excellent high-performance method to deliver data simultaneously to many threads.

In the Fermi architecture the streaming multiprocessor schedules threads in groups of 32 parallel threads called warps. We have used a warp serialize flag to determine if a shared memory bank conflict occurs in any of the kernels. The algorithm breaks the data block into warp-sized fragments and then scans these fragments independently by using a warp of threads for each of them. We do not have to synchronize in order to share data within the same warp, only across different warps, because the instructions are executed in a SIMD (Single Instruction Multiple Data) way. To complete the scan, the warps write their last elements to a temporary shared array where a single warp performs an exclusive scan on it. In the end, each thread adds the warp's sum result to the one from the first pass. In SIMD the most efficient algorithm is a step-efficient one, so we have used within each warp a Hillis-and-Steele-style scan that takes $log_2 n$ steps to scan the warp [13], rather than the work-efficient tree-based algorithm described by Guy Blelloch [12]. Because the warp size is 32, there are five steps per warp. Each thread scans a single array element, inputs a single value to the scan through the variable "valoare" and returns its own scanned result element. Each of the warps' threads cooperates through the shared memory array "date_partajate" to scan a number of "DIM_WARP" elements. The template parameter "nivelmax" allows the scan to be performed on partial warps. For example, if we need only the first 16 elements of each warp to be scanned, the scan performs only $log_2 16 = 4$ steps rather than $log_2 32 = 5$. In order to enable warps to offset beyond their input data and receive the identity element without having to use branch instructions, the computation uses 2*DIM_WARP elements of shared memory for each warp. The shared memory array "date_partajate" is declared volatile in order to prevent the compiler from optimizing away writes to shared memory and to ensure a correct communication within the same warp without the need of synchronization. In Fig. 2 it is presented the "scanarewarp" CUDA kernel.


```
template<class T, int nivelmax>
__device__ T scanarewarp(T valoare, volatile T* date_partajate)
{
    int idx = 2 * threadIdx.x - (threadIdx.x & (DIM_WARP-1));
    date_partajate[idx] = 0;
    idx += DIM_WARP;
    T t = date_partajate[idx] = valoare;

    if (0 <= nivelmax) { date_partajate[idx] = t = partial(t, date_partajate[idx - 1]); }
    if (1 <= nivelmax) { date_partajate[idx] = t = partial(t, date_partajate[idx - 2]); }
    if (2 <= nivelmax) { date_partajate[idx] = t = partial(t, date_partajate[idx - 4]); }
    if (3 <= nivelmax) { date_partajate[idx] = t = partial(t, date_partajate[idx - 8]); }
    if (4 <= nivelmax) { date_partajate[idx] = t = partial(t, date_partajate[idx -16]); }

    return date_partajate[idx-1];
}
```
Fig. 2. The "scanare_warp" CUDA kernel

Finally, taking into account all these technical aspects, we have obtained an efficient parallel scan routine that optimizes the algorithm for computing the eigenvalues of a tridiagonal symmetric matrix, avoiding the shared memory bank conflicts.

9. Examples and numerical results

In the following, we depict a series of experimental tests. In the benchmarking, we have used the following configuration: Intel i7-2600K at 3.4 GHz with 8 GB (2x4GB) of 1333 MHz DDR3, dual channel. The GPU used was GeForce GTX480 (from the FERMI architecture). Programming and access to the GPUs used the CUDA toolkit 4.0 with NVidia driver version 270.81. In addition, all processes related to graphical user interface have been disabled to reduce the external traffic to the GPU.

In order to compute the average execution time we could have used one of the CPU or operating system timers, but this would have included latency and variations from different sources like system thread scheduling, availability of high-precisions CPU timers, etc. In addition, we can asynchronously perform computations on the host while the GPU kernel runs and the only way to measure the necessary time for the host computations is to use the CPU or the operating system timing mechanism. Therefore, in order to measure the average execution time a GPU spends on a task we use the CUDA event API. A GPU time stamp recorded at a user specified point in time represents an event in CUDA. Because the time stamp is recorded directly by the GPU, we do not encounter the problems that could have appeared if we had tried to time the GPU execution using CPU timers. In order to time correctly the execution of the algorithm, we create both a start and a stop event. Some of the kernel calls we make in CUDA C are asynchronous, the GPU begins to execute the code but the CPU continues the execution of the next line of the program before the GPU has finished. In order to safely read the value of the stop event we instruct the CPU to synchronize on the event using the API function “cudaEventSynchronize()”, like depicted in Fig. 3.

```
cudaEvent_t inceput, sfarsit;
cudaEventCreate(&inceput);
cudaEventCreate(&sfarsit);
cudaEventRecord(inceput, 0);
calculValProprii(intrare, dim_matrice, precizie, Gerschgorin_inf, Gerschgorin_sup, rezultat);
cudaEventRecord(sfarsit, 0);
cudaEventSynchronize(sfarsit);
```

Fig. 3. Measuring the execution time using events

In this way we instruct the runtime to block further instructions until the GPU has reached the stop event so when calling the “cudaEventSynchronize()” we are sure that all the GPU work prior to the stop event has been completed and it is safe to read the recorded time stamp. In this way we get a reliable measurement of the execution time for computing the eigenvalues.

In *Table 1* we present the experimental results obtained by taking different sizes of the tridiagonal input matrix and choosing 10^{-6} as the desired accuracy, necessary in the bisection algorithm for computing the eigenvalues of the tridiagonal symmetric matrix with real components.

Table 1

The average execution time for different input matrix sizes

Test number	Matrix size	Average execution time (milliseconds)	Test number	Matrix size	Average execution time (milliseconds)
1	16	0.170977	7	1024	18.270479
2	30	0.287934	8	2040	34.498478
3	64	0.610396	9	4096	66.367801
4	120	1.080387	10	8190	140.005358
5	256	2.383497	11	16384	373.554061
6	510	5.254698	12	32760	913.889898

The average execution time measured in milliseconds is computed in the third column of the Table 1. For each average execution time, 5 different examples of matrices have been generated, their size being indicated in the second column of the table. For each of the matrices the eigenvalues and the necessary time for obtaining them have been computed. Finally the average time corresponding to these 5 values has been computed. The sizes of the input matrices can be any natural number $\boldsymbol{n \geq 2}$, sufficiently large for most applications, and have been chosen as to cover both cases of small and large dimensions. In order to fill quickly the tridiagonal matrices used in the numerical examples, the necessary real numbers have been automatically generated, using a random number generator. It is also possible to provide the input matrix through a file containing the desired values.

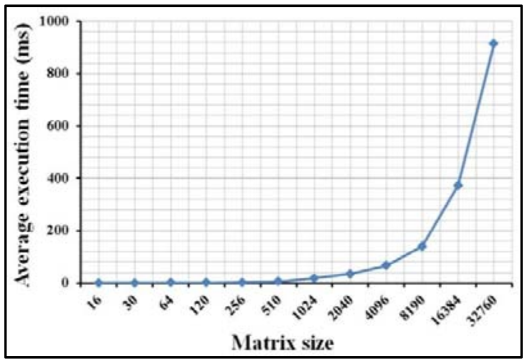


Fig. 4. The average execution time for different input matrix sizes

The effect of matrix dimension on the average execution time (based on the results listed in the Table 1) is represented in Fig. 4. As we had expected, the average execution time increases with the matrix dimension, even if there are more threads invoked when the input matrix has a greater size and the parallel computation level is higher. On a GeForce GTX480 (from the FERMI architecture), execution times in our experimental tests have varied between 0.170977 ms (corresponding to a symmetric matrix 16x16) and 913.889898 ms (corresponding to a symmetric matrix 32760x32760), which is 5345 times greater than the first measurement, but still under 1 second.

10. Conclusions and future work

In this paper, after we have depicted the main features of the Compute Unified Device Architecture, we have briefly described some mathematical aspects regarding the computation of eigenvalues and afterwards we have presented an original approach: a bisection algorithm useful in computing eigenvalues for a tridiagonal symmetric matrix of arbitrary size, using the computing capabilities of the latest graphics processing units that incorporate the parallel Compute Unified Device Architecture developed by NVidia. The implemented algorithm was based on an efficient parallel prefix sum algorithm designed and implemented in CUDA. The computation of the eigenvalues for a tridiagonal symmetric matrix of arbitrary size starts by obtaining the Gerschgorin interval that contains all the eigenvalues of the input matrix. After applying a recursive bisection process, we obtain a tree, which contains in each leaf an approximation of an eigenvalue, with a desired accuracy. The obtained algorithm can be successfully applied to arbitrary large matrices.

The novel approach developed in this paper is related to the efficiency of the algorithm's implementation, which is based on the improvement of the shared memory management and on the optimization of the scan algorithm. Therefore, the algorithm for computing the eigenvalues of a tridiagonal symmetric matrix is a powerful and useful tool, able to facilitate the work of specialists in both pure and applied mathematics. The eigenvalues computation is useful in many domains like: in the Schrödinger equation in quantum mechanics; in atomic and molecular physics; in geology and glaciology; in the principal component analysis and factor analysis in structural equation modeling; in vibration analysis; in image processing and sound processing; in mechanics; in spectral graph theory; in medical studies etc.

The implementation depicted in this paper can be further improved using different approaches. A first solution is to increase the level of parallelism in the computations. In perspective, the study in this article can be developed and enhanced by executing the computations on multiple GPUs connected in parallel, using the Scalable Link Interface (SLI), a multi-GPU solution developed to link two or more graphics processors together, so that they can process as a whole. A second possible method to improve significantly the performance obtained in our implementation is to store both the active intervals and the input matrix in shared memory, avoiding the access to the global memory in the step of counting the eigenvalues. Our implementation can be improved by processing multiple intervals per thread. Another possible improvement is to find an enhanced method for dividing the Gerschgorin interval, the best choice being if the number of intervals matches the number of parallel processors on the device.

A frequently encountered problem when computing eigenvalues is that they are not always well distributed and this might cause an unbalanced load of the processors. Therefore, an improvement to the above-described algorithm is first to find the initial intervals using the device memory, which would help obtain equilibrated workloads for all the multi-processing units. A series of studies in this field already exists, but problems still remain open to further research.

R E F E R E N C E S

[1]. *B. N. Parlett,* The Symmetric Eigenvalue Problem, Prentice-Hall, New Jersey, 1980.

[2]. *I. S. Dhillon, B. N. Parlett,* "Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices", in Linear Algebra and its Applications, **vol. 387**, no.1, August 2004, pp 1-28.

[3]. *G. H. Golub, C. F. van Loan,* Matrix Computations, 2nd Edition, The John Hopkins University Press, Baltimore, 1989.

[4]. *J. Sanders, E. Kandrot,* CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, New Jersey, 2010.

[5]. *** NVidia CUDA - Programming Guide, Version 3.1, NVidia Whitepaper, 2010.

[6]. *A. Pirjan*, "Improving software performance in the Compute Unified Device Architecture", in Informatica Economica, **vol. 14**, no. 4, December 2010, pg. 30-47.

[7]. *P. Bakkum, K. Skadron*, "Accelerating SQL Database Operations on a GPU with CUDA", in Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, **vol. 425**, no.1, March 2010, pp. 94-103.

[8]. *J. Demmel, I. Dhillon, H. Ren,* "On The Correctness Of Some Bisection-Like Parallel Eigenvalue Algorithms In Floating Point Arithmetic", in Trans. Num. Anal. (ETNA), **vol. 3**, no.1, December 1995, pp. 116-149.

[9]. *C. Lessig,* Eigenvalue Computation with CUDA, NVidia Corporation, California, 2007.

[10]. *M. Harris, S. Sengupta, J. D. Owens, "*Parallel Prefix Sum (Scan) with CUDA", in GPU Gems 3, Pearson Education, Boston, 2007.

[11]. *K. V. Fernando, "*Accurately Counting Singular Values of Bidiagonal Matrices and Eigenvalues of Skew-Symmetric Tridiagonal Matrices", in SIAM Journal on Matrix Analysis and Applications, **vol. 20**, no. 2, April 1999, pp. 373 - 399.

[12]. *G. E. Blelloch,* "Prefix Sums and Their Applications", in Synthesis of Parallel Algorithms, Morgan Kaufmann, Massachusetts, 1990.

[13]. *D. Hillis, G. Steele,* "Data Parallel Algorithms", in Communications of the ACM, **vol. 29**, no.12, December 1986, pp. 1170-1183.