Computing the singular value decomposition in parallel on graphics processing units using a one-sided Jacobi method

Michael V. Romer

Citation: Proc. Mtgs. Acoust. **19**, 070096 (2013); doi: 10.1121/1.4801392
View online: https://doi.org/10.1121/1.4801392
View Table of Contents: https://asa.scitation.org/toc/pma/19/1
Published by the Acoustical Society of America

Proceedings of Meetings on Acoustics

Volume 19, 2013 http://acousticalsociety.org/

ICA 2013 Montreal
Montreal, Canada
2 - 7 June 2013

Underwater Acoustics
Session 5aUW: Using Graphic Processing Units for Computationally Intensive Applications in Acoustic Modeling and Signal Processing

5aUW5. Computing the singular value decomposition in parallel on graphics processing units using a one-sided Jacobi method

Michael V. Romer*

***Corresponding author's address: Applied Research Laboratories at the University of Texas at Austin, Austin, TX 78758, romer@arlut.utexas.edu**

The singular value decomposition (SVD) provides a robust means of determining the dominant modes in a collection of signals that can then be used in adaptive beamforming to suppress loud interferers that would otherwise cover signals of interest. However, on serial architectures this decomposition can quickly become a bottleneck, hindering the real-time performance of the beamformer. Commercial off-the-shelf graphics processing units (GPUs) are an inexpensive means of adding parallel processing capabilities to a system and can reduce computation time by several orders of magnitude. The following work presents an algorithm that computes the full SVD of a dense matrix in parallel using a one-sided Jacobi method on a standard GPU. Both the runtime performance and speed of convergence for the algorithm are shown. Potential limitations to this approach due to restrictions imposed by the hardware are also discussed.

Published by the Acoustical Society of America through the American Institute of Physics

 [DOI: 10.1121/1.4801392]
Received 22 Jan 2013; published 2 Jun 2013

INTRODUCTION

Adaptive beamforming is an instrumental part of undersea surveillance systems that utilize passive sonar arrays. Over conventional beamforming methods, it can provide greater suppression of loud interferers thereby allowing for better detection of quieter signals. Dominant mode rejection (DMR) is an enhancement of the minimum variance distortionless response beamformer on top of which many other adaptive beamforming methods are built. [1] It estimates the largest eigenvalues and eigenvectors of the cross spectral density matrix (CSDM) and uses them to null contributions from the loudest sources in the sampled environment.

In practice, this eigendecomposition can be performed by computing the singular value decomposition (SVD) of a matrix containing several snapshots of the Fourier transformed output of the individual hydrophone elements. However, for a matrix A with dimensions $m \times n$ ($m \geq n$), the complexity of computing the SVD of A grows as a function of mn^2. [2] Consequently, calculating the SVD can become a predominant factor impacting the efficiency of a DMR-based beamformer, especially for arrays containing a large number of receivers. In order to achieve realtime throughput or better of the beamformer, it becomes desirable to find ways of reducing the time necessary to compute the SVD.

Recent trends in computing have resulted in a greater emphasis placed on exploiting parallelism rather than relying on faster hardware in order to achieve scalable improvements in performance. With the proliferation of multi- and many-core processors, it has become increasingly more important to find algorithms that partition data sets into independent sub-problems that can be solved concurrently. This allows software to scale to arbitrary levels of parallelism independent of the system's core count.

To that end, this paper will explore the feasibility and practicality of implementing a parallel SVD on a graphics processing unit (GPU). GPUs are massively parallel devices that excel at processing data using algorithms with a high density of arithmetic operations. The basic approach used to implement the SVD is based on a one-sided Jacobi method demonstrated by Hestenes in 1958 that has the desired property of being inherently parallel. The next section will provide an overview of this method so as to provide the context necessary for the discussion of the parallel implementation that follows.

SOLVING THE SVD WITH A ONE-SIDED JACOBI

Of interest is the decomposition of a matrix A into matrices U and V with orthonormal columns and diagonal matrix Σ such that $A = U\Sigma V^T$. Such a factorization is the SVD of A and always exists. If A is $m \times n$ ($m \geq n$), then the following properties hold for the SVD:

- Σ is an $n \times n$ diagonal matrix whose diagonal entries are the singular values of A.
- U is an $m \times n$ matrix whose n columns are the left singular vectors of A.
- V is an $n \times n$ matrix whose n columns are the right singular vectors of A.

Jacobi methods are iterative procedures originally developed to find the eigenvalues of real symmetric matrices. They have since been adapted to solve for the SVD of general matrices. Hestenes showed how a one-sided Jacobi method could be used to solve for the SVD via the following series of orthogonal transformations: [3]

$$\begin{aligned} A_0 &= A \\ A_k &= A_{k-1} V_k \end{aligned} \tag{1}$$

V_k is an orthogonal matrix chosen such that the columns of A_k become increasingly more orthogonal after each transformation. Such a procedure is known as a one-sided Jacobi method because it involves only a post-multiplication of V_k. Hestenes showed that if

$$W = \lim_{k \to \infty} A_k \tag{2}$$

then the columns $\boldsymbol{w_1},\ldots,\boldsymbol{w_n}$ are mutually orthogonal and $\|\boldsymbol{w_i}\| = \sigma_i$ are the singular values of A. Since the product of two orthogonal matrices is itself an orthogonal matrix, the following yields a singular value decomposition for A:

$$U\Sigma = W = A_k = A(V_1 \cdots V_k) = AV \tag{3}$$

The goal is then to find V as a product of orthogonal matrices that, when applied to A, results in a matrix $U\Sigma$ with orthogonal columns. One approach is to construct V from a series of plane rotations where each rotation V_k orthogonalizes the columns $\boldsymbol{a_p}$ and $\boldsymbol{a_q}$ $(p < q)$ of the current matrix A_{k-1}. Each rotation V_k is given by:

$$V_k = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & p & & q & & \end{array} \\ \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos\phi & \cdots & -\sin\phi & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin\phi & \cdots & \cos\phi & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \end{array} \tag{4}$$

Nash showed a method of determining $c = \cos\phi$ and $s = \sin\phi$ that had the effect of sorting the singular values in decreasing order down the diagonal of Σ. [4] Since each rotation updates only the columns $\boldsymbol{a_p}$ and $\boldsymbol{a_q}$, the corresponding columns in A_k can be written as

$$\begin{aligned} \boldsymbol{a_p}' &= c\boldsymbol{a_p} + s\boldsymbol{a_q} \\ \boldsymbol{a_q}' &= -s\boldsymbol{a_p} + c\boldsymbol{a_q} \end{aligned} \tag{5}$$

Rotations are grouped together in *sweeps* where each sweep orthogonalizes each of the $n(n-1)/2$ possible pairs of columns. At the end of each sweep, convergence is tested by counting the pairs of mutually orthogonal columns in A_k. If the count equals $n(n-1)/2$, convergence is assumed, and the algorithm is complete.

PARALLELIZING THE ONE-SIDED JACOBI

Parallelizing the one-sided Jacobi entails partitioning the $n(n-1)/2$ pairs of columns that must be orthogonalized each sweep into *sets* of independent column pairs. Each sweep is then processed one set at a time, orthogonalizing the column pairs within the current set in parallel.

Column pairs for each set are generated using a round-robin tournament scheduling algorithm. Conceptually, each round in the tournament represents a set, and the pairings of players within a round correspond to the pairings of columns within that set. As an example, the following is all possible sets containing their respective column pairs for $n = 6$:

$$\begin{aligned} \text{Set } 1 &= \{(1,2),(3,4),(5,6)\} \\ \text{Set } 2 &= \{(1,4),(2,6),(3,5)\} \\ \text{Set } 3 &= \{(1,6),(2,3),(4,5)\} \\ \text{Set } 4 &= \{(1,5),(2,4),(3,6)\} \\ \text{Set } 5 &= \{(1,3),(2,5),(4,6)\} \end{aligned}$$

In general, each set contains $\hat{n}/2$ pairs of columns that are orthogonalized in parallel, where $\hat{n}$ is the next even integer greater than or equal to n. If n is odd, then each set will have one column paired with a "ghost" column; pairs containing the ghost column are not orthogonalized. Under this scheme, it takes $\hat{n} - 1$ sets to complete a full sweep.

Column pairs for each set are generated using a variation of the **music** procedure originally provided in Golub and Van Loan. [5] The modified version does not store the column pairs as a pair of arrays. Instead,

the column pair (p', q') orthogonalized by the i^{th} rotation in a set is computed directly from the corresponding column pair (p, q) orthogonalized by the i^{th} rotation in the previous set. In practice, this generation scheme is better suited for execution on a GPU where compute bandwidth greatly exceeds memory bandwidth.

Using GPUs for Parallel Computation

Parallelizing an algorithm for a particular device requires understanding both the underlying hardware architecture and the programming model used to abstract it. This project targets the NVIDIA Geforce GTX 670 GPU and uses version 5.0 of NVIDIA's CUDA software development kit, which defines the model and programming interfaces used for development. The discussions that follow rely on concepts specific to CUDA and NVIDIA GPUs, an overview of which is given next. Further details can be found in the CUDA C Programming Guide. [6]

CUDA Programming Model

The CUDA programming model exposes the GPU as a separate *compute device* that works cooperatively with a *host*, which is typically the system CPU. Software for the device is written as a single *kernel function* that is *launched* by the host for execution on the device.

When the host launches a kernel, it defines a *compute grid* over which the kernel is executed. The compute grid consists of a number of *thread blocks*, where each block contains the same number of threads. Once a kernel has been launched, all threads in the grid's thread blocks begin running the kernel's code concurrently. Execution continues until all thread blocks have returned from the kernel.

Under CUDA, each thread block has access to a *global memory* store located on the device. However, reading from or writing to global memory is a high latency operation, and accessing it too frequently can easily diminish the overall performance of the kernel. To mitigate this, each thread block can allocate its own segment of *shared memory* to use for computation and communication between its threads.

Accessing shared memory can be up to two orders of magnitude faster than accessing global memory. It is common, then, to read data from global memory into shared memory, perform some compute-intensive function on it, and then write the results back to global memory. However, the amount of shared memory on the device is considerably lower than the amount available for global memory, thus limiting the amount of low-latency work that can be done per global memory transaction. Furthermore, unlike for global memory, the contents of shared memory do not persist across kernel launches.

NVIDIA GPU Hardware Architecture

An NVIDIA GPU comprises a set of *streaming multiprocessors* responsible for executing a grid of thread blocks concurrently. At kernel runtime, each thread block is assigned a multiprocessor on which it will run. The multiprocessor then divides the currently running threads within each assigned thread block into groups of 32 threads called *warps*.

When a warp is ready for execution, a warp scheduler issues its next instruction, and all threads within that warp execute it in parallel. If a branch causes threads within a warp to take different code paths, the warp *diverges*, and execution must be serialized such that one branch path is fully executed before the second path may be executed. After all branch paths have been executed, threads within the warp converge back together. To achieve maximum thread-level parallelism, it is important to minimize the number of diverging warps.

Each multiprocessor also has an area of on-chip memory that provides low-latency read-write access. This area is used, in part, to implement the shared memory space used by threads currently executing on the multiprocessor. For the Geforce GTX 670, there is a maximum of only 48KB of shared memory available per multiprocessor. Since this memory space is used by all thread blocks executing on the multiprocessor, shared memory quickly becomes a limiting resource that must be carefully managed in order to maximize the amount of work done per global memory transaction while still allowing for a high degree of parallelism.

IMPLEMENTING THE SVD USING CUDA

It is now possible to explore using CUDA to implement a parallel SVD based on the one-sided Jacobi method described above. The $m \times n$ input matrix A is taken to be real-valued with $m \geq n$. If $m < n$, then the SVD of A can be computed from the SVD of $A^T = V\Sigma U^T$. Entries are stored in column-major order as single-precision floating point numbers.

Defining the Kernels

The implementation fundamentally consists of two operations that must be done in order to find matrices U, Σ, and V. First is transforming A into W while simultaneously computing V via orthogonalization, and the second is column scaling W in order to obtain U and Σ. Due to limitations discussed below, it is not possible to combine both operations into a single kernel. Consequently, the SVD is implemented as a pair of kernels: an orthogonalization kernel K_{ortho} and a column-scaling kernel K_{scale}.

K_{ortho} Details

K_{ortho} concurrently orthogonalizes each pair of columns in a set, giving it a linear compute grid consisting of $\hat{n}/2$ thread blocks. For each set, the b^{th} thread block is responsible for orthogonalizing the columns $\boldsymbol{a_{p_b}}$ and $\boldsymbol{a_{q_b}}$, where (p_b, q_b) is the b^{th} column pair of the current set.

Each thread block consists of m threads and needs at most $16m$ bytes of shared memory to store the column vectors $\boldsymbol{a_{p_b}}$, $\boldsymbol{a_{q_b}}$, $\boldsymbol{v_{p_b}}$, and $\boldsymbol{v_{q_b}}$. Threads with consecutive thread identifiers read from global memory into shared memory consecutive elements of each of the previously listed column vectors. This has the benefit of coalescing global memory accesses into fewer transactions, thereby improving the memory bandwidth of the kernel.

With one thread allocated per element for one of the column vectors read into shared memory, it is possible to update each vector component in parallel, resulting in increased thread-level parallelism. Since each thread undergoes the same computation, divergent warps only occur when fewer than m threads need to be used to update the components of $\boldsymbol{v_{p_b}}$, and $\boldsymbol{v_{q_b}}$.

K_{scale} Details

K_{scale} normalizes the n columns of the matrix $W = U\Sigma$ obtained from orthogonalization. Since normalizations are independent of each other, the compute grid for K_{scale} is sized to n thread blocks. Similar in principle to K_{ortho}, the b^{th} thread block is responsible for normalizing $\boldsymbol{u_b}$, the b^{th} column of $U\Sigma$.

Each thread block contains m threads and requires $4m$ bytes of shared memory to store the column vector u_b. By allocating one thread per element in u_b, K_{scale} is also able to take advantage of coalescing global memory transactions and updating each component of u_b in parallel.

Kernel Implementation Restrictions

The CUDA programming model has an inherent design trait that complicates the implementation of K_{ortho}. In short, it is impossible to synchronize the execution and global memory transactions of threads in different thread blocks within a running kernel. Without loss of generality, for K_{ortho} this means that at the end of a set when thread block b_1 writes its orthogonalized columns back to global memory, there is no way to guarantee that thread block b_2 will see and read in either of the updated columns at the start of the next set.

In practice, the only way for threads in different thread blocks to synchronize and share data is to relaunch the kernel after each thread block has written its initial results to global memory. This restricts the scope of K_{ortho} and forces the host to launch the kernel once per set. This results in the following undesirable consequences:

- For every run of K_{ortho}, each thread block must read the columns $\boldsymbol{a_{p_b}}$, $\boldsymbol{a_{q_b}}$, $\boldsymbol{v_{p_b}}$, and $\boldsymbol{v_{q_b}}$ from global memory into shared memory. Since the contents of shared memory are transient, this guarantees that each thread block on every set pays the penalty for reading four column vectors from global memory.

  In contrast, if synchronization across thread blocks were possible, it is possible a more intelligent column pair partitioning scheme could be devised. Ideally, such a scheme would minimize the number of global memory reads a thread block requires at the start of a new set.

- Convergence testing must be done on the host. This requires a counter to be stored in global memory that is incremented for each thread block whose column pair is already sufficiently orthogonal. The host then needs to read this value at the end of each set to determine if it needs to continue calling K_{ortho}.

  Since the host must read memory from the device after every set, this has the effect of implicitly synchronizing the host with the device after every invocation of K_{ortho}. This effectively reduces, if not eliminates, possibilities for concurrent kernel execution, which in turn limits total throughput.

RESULTS

The runtime performance for an initial, unoptimized implementation of the approach given above was measured on a test workstation equipped with an Intel Xeon E31245 3.30 GHz CPU, 16GB of RAM, and the aforementioned Geforce GTX 670. Tests were conducted using square matrices ranging from 2×2 to 512×512 in size. The entries of each matrix were randomly selected integers from the interval $[-50, 50]$.

The author readily admits these matrices do not correlate with those that would be decomposed in a production system. At the time of writing, research is still ongoing to determine the efficacy of the presented implementation in a more realistic signal processing environment.

Despite these circumstances, performance testing revealed some unique runtime characteristics. Figure 1 plots on a logarithmic axis the combined time in seconds required for K_{ortho} and K_{scale} to compute the SVD as a function of matrix size. Of particular interest is the increased scattering that occurs around $n \geq 400$. Figure 2, which displays on a linear axis the same timing information as Figure 1, better illustrates this phenomenon.

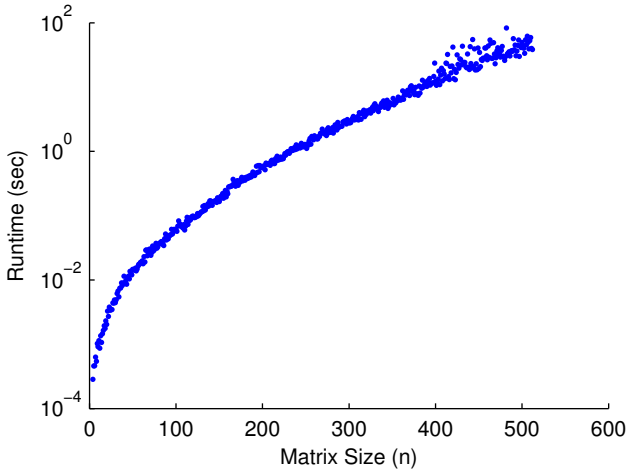

FIGURE 1: Combined runtime of K_{ortho} and K_{scale}, in seconds, for matrices of size $n \times n$. Logarithmic time axis.

It is unclear at this point in time why the variance of the kernel runtime increases for larger matrix sizes. One hypothesis that future work plans to test is that there is greater contention for hardware resources at larger matrix sizes, namely execution time on one of the seven multiprocessors available on the Geforce GTX 670.

It is interesting to note that a similar pattern is observed in Figure 3 when measuring the number of sweeps required for convergence as a function of matrix size. This suggests that at a certain point, namely

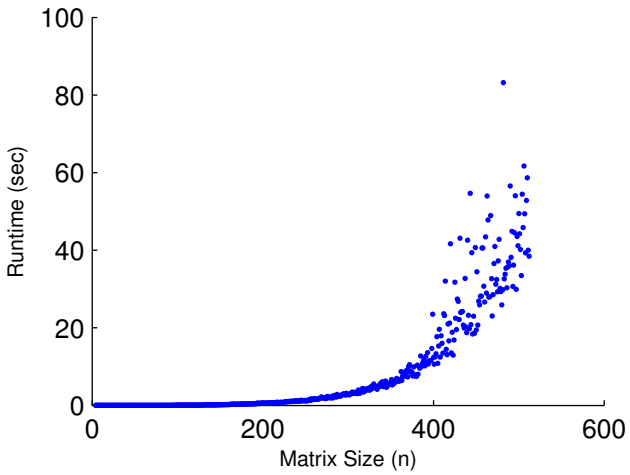

FIGURE 2: Combined runtime of K_{ortho} and K_{scale}, in seconds, for matrices of size $n \times n$. Linear time axis.

around $n \geq 400$ in this instance, it becomes significantly more difficult to orthogonalize large matrices. Whether this is due to an accumulation of round-off error, the intrinsic structure of the input matrices, or some other factor is something further work is also continuing to explore.

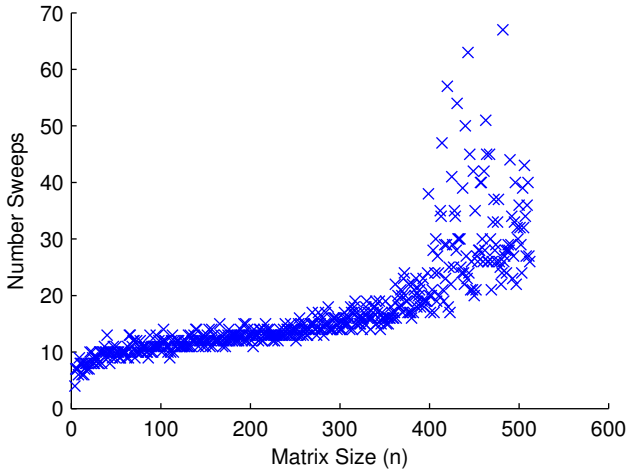

FIGURE 3: Number of sweeps required for convergence for matrices of size $n \times n$.

CONCLUSION

In principle, an SVD based on the one-sided Jacobi method can be adapted to exploit readily available devices with massively parallel capabilities. With it, it is possible to achieve a high degree of thread-level parallelism that is on the order of the number of elements in the matrix.

However, core characteristics of both the programming model and hardware architecture of NVIDIA GPUs make the above approach a bit of a mismatch for them in practice. Most notably a lack of grid-level thread synchronization forces the orthogonalization process to be spread out over multiple kernel invocations. Consequently, the host is forced to synchronize with the device after each set, and the device is unable reduce global memory transactions between sets through caching of data in shared memory. Both of these result in additional overhead that ultimately limits the ability to fully exploit the compute capabilities of the GPU.

The initial results presented above indicate that the potential performance may be limited even for ma-

trices of a relatively small order. However, SVDs used in signal processing, and in particular adaptive beamforming, typically decompose matrices of smaller sizes than the majority of those tested, thus these limitations may not be fully realized in practice. Still, open questions remain on the suitability of the above approach used in production, with further work being done to address the following:

- Accurately quantifying runtime performance using measured data sets.
- Determining the overhead of performing host-side convergence testing.
- Measuring the relative error between the proposed approach and established SVD implementations such as `GESVD` from the Intel Math Kernel Library.

References

[1] N. Owsley, "Enhanced minimum variance beamforming", in *Underwater Acoustic Data Processing*, edited by Y. Chan, 285–291 (Kluwer Academic Publishers) (1989).

[2] L. Trefethen and D. Bau III, *Numerical Linear Algebra* (Society of Industrial Mathematics) (1997).

[3] M. Hestenes, "Inversion of matrices by biorthogonalization and related results", Journal of the Society for Industrial & Applied Mathematics **6**, 51–90 (1958).

[4] J. Nash, "A one-sided transformation method for the singular value decomposition and algebraic eigenproblem", The Computer Journal **18**, 74–76 (1975).

[5] G. Golub and C. Van Loan, *Matrix Computations*, 3rd edition (Johns Hopkins University Press) (1996).

[6] NVIDIA Corp., *CUDA C Programming Guide* (2012).