

Parallel Latent Semantic Analysis using a Graphics Processing Unit

Joseph M. Cavanagh^{*}
Division of Science and
Mathematics
University of Minnesota -
Morris
Morris, Minnesota 56267
cava0093@umn.edu

Thomas E. Potok
Computational Sciences &
Engineering Division
Oak Ridge National
Laboratory
Oak Ridge, Tennessee 37831
potokte@ornl.gov

Xiaohui Cui[†]
Computational Sciences &
Engineering Division
Oak Ridge National
Laboratory
Oak Ridge, Tennessee 37831
cuix@ornl.gov

ABSTRACT

Latent Semantic Analysis (LSA) can be used to reduce the dimensions of large Term-Document datasets using Singular Value Decomposition. However, with the ever expanding size of data sets, current implementations are not fast enough to quickly and easily compute the results on a standard PC. The Graphics Processing Unit (GPU) can solve some highly parallel problems much faster than the traditional sequential processor (CPU). Thus, a deployable system using a GPU to speedup large-scale LSA processes would be a much more effective choice (in terms of cost/performance ratio) than using a computer cluster. In this paper, we presented a parallel LSA implementation on the GPU, using NVIDIA@Compute Unified Device Architecture (CUDA) and Compute Unified Basic Linear Algebra Subprograms (CUBLAS). The performance of this implementation is compared to traditional LSA implementation on CPU using an optimized Basic Linear Algebra Subprograms library. For large matrices that have dimensions divisible by 16, the GPU algorithm ran five to six times faster than the CPU version.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous;
D.2.8 [Software Engineering]: Metrics—*performance measures*

General Terms

Algorithms, Performance

^{*}The first author conducted the research at Oak Ridge National Laboratory under the support of the Department of Energy's Student Undergraduate Laboratory Internship program

[†]Correspond author

Keywords

GPU, Text Mining, Latent Semantic Indexing

1. INTRODUCTION

With the large amount of data being collected annually, methods are needed to extract valuable information from this data [1]. Latent Semantic Analysis is a numerical technique used to extract information from large collections of text documents [2, 3]. These document collections often contain more than 10,000 unique documents and 5,000 unique terms. LSA is employed on these collections in order to find relationships between various terms, sentences, and full documents. LSA works by taking the SVD of A where $A = D * D^T$, with D being the term-document matrix. The term document matrix is created before hand, using a term weighting and text stripping algorithm such as Term Frequency Inverse Document Frequency (TFIDF) [13]. The core of the SVD algorithm requires an eigen decomposition of the matrix, which has been a computational problem for many decades [2, 5, 6, 10]. This makes SVD a computationally expensive algorithm which makes it a prime candidate for decreasing processing times [2, 5, 10]. During the SVD process, many modern methods tridiagonalize A before computing the SVD due to the performance increase between calculating SVD on a normal matrix and a tridiagonal matrix [2, 5, 6, 14]. A tri-diagonal matrix is a matrix with values only in its main diagonal, one element above the main diagonal, and one element below the main diagonal. All other elements in the matrix are set to zero. This step takes up a large portion of the time for computing the SVD and will be the main focus of our parallel algorithm.

Recently, the GPU has become a focus for inexpensive, high performance computing in various scientific fields [9]. The GPU serves as a specialized processor that is tailored to make extremely fast graphics calculations. Demands for increasingly realistic visual representations in simulation and entertainment have driven the development of the GPU. As is evident in figure 1, the most recent generation of NVIDIA@GPU has a theoretical performance much greater than the current top-of-the-line desktop CPU (the Intel Core i7-965 Extreme Edition). This difference arose because the evolution of the GPU has centered on highly parallel, computationally intensive calculations rather than data caching and flow control [8]. The immense computational power

of the GPU was noticed by developers, and a move to exploit this power was made. A community of general-purpose GPU programmers (www.gpgpu.org) quickly arose and pioneered programming on the GPU. Due to the architecture of the GPU, it is able to perform floating point calculations much faster than a standard CPU, due to the massive parallelism in the GPU [9]. We performed the CPU benchmarks using SiSoftware®Sandra, a program for benchmarking various computer components. An Intel Core i7-965 Extreme Edition CPU costs around \$1000 and produces 69GFLOPS (GFLOP, equal to one billion floating point operations per second). An NVIDIA®1GB memory GTX 280 costs around \$350 and produces around 900 GFLOPS. The price to performance ratio for the CPU is 0.069 GFLOPS/dollar, while the GPU price to performance ratio is 2.66 GFLOPS/dollar. The impressive price to performance ratio of the GPU makes it a prime candidate for increasing the speed of LSA, while still using components found in many desktop and laptop personal computers.

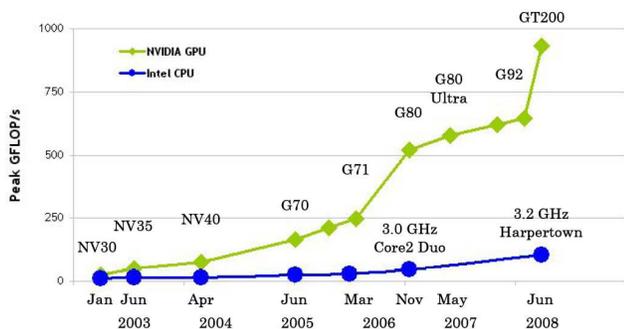


Figure 1: Floating-Point Operations per Second for the CPU and GPU [8]

Due to the GPU’s application-specific architecture, harnessing the GPU’s computational prowess for LSA is a great challenge. Memory transfer from system memory to GPU memory (host to device and device to host) remains relatively slow and can often be a bottleneck in the applications. Most current GPUs offer support for only single precision, while many scientific applications require double precision support. Also, certain algorithms are primarily serial, prohibiting much of the GPU’s processing power from being utilized. For each of these problems a solution must be found. Currently, the effect of slow memory transfer can be minimized by performing as many GPU based computations as possible between CPU memory to GPU memory transfer. Transferring large amounts of data at one time is generally faster than making numerous small memory transfers. Double precision can be either emulated by an algorithm or can be obtained by purchasing a new NVIDIA®200 series GPU.

2. RELATED RESEARCH

Our literature research shows that currently there is very limited researches in GPU based implementation of LSA. The related research includes: A team at the University of North Carolina at Chapel Hill used a GPU based algorithm for solving dense linear systems [4]. Their implementation of an LU decomposition algorithm performed 35% better than an ATLAS (Automatically Tuned Linear Algebra Subroutines) implementation on the CPU, for matrices of size 3500.

Manavski and Valle have implemented the Smith-Waterman algorithm on the GPU [7]. The Smith-Waterman algorithm explores alignments between two sequences in protein and DNA databases. Their implementation ran between 2 and 30 times faster than other implementations for commodity hardware. Another example of the power of the GPU is given in [12]. They used the GPU to implement ray tracing algorithms and then compared the results to implementation on a CPU. In an animated example, the GPU performed around 6 times better than the CPU. These examples show that the GPU is a powerful tool that when used correctly can be used to increase the speeds of a variety of algorithms in a variety of fields.

3. MATERIALS AND METHODS

For our implementation, we decided to use a Lanczos algorithm to assist with the SVD. The Lanczos algorithm tridiagonalizes a matrix which allows for the computation of SVD to be performed significantly faster [6]. This is done by using various matrix-vector and vector-vector operations in order to achieve a Krylov subspace. This subspace is a representation of the original matrix and maintains approximate eigenvalues and eigenvectors to that of the original matrix. The reason we chose Lanczos is that the only computationally expensive part of the algorithm is a matrix-vector multiplication. The algorithm can be seen in figure 2. The vector alpha is then used to form the main diagonal of the new matrix, where the vector beta will form the sub-diagonal and superdiagonal. This algorithm is proven to be accurate in an environment without rounding errors. However, due to the fact that our GPU supports single precision, rounding errors are inevitable. Maintaining accuracy while using the GPU will be discussed more in the Discussion and Conclusion section.

Our GPU implementation uses CUBLAS to perform the matrix-vector and vector-vector operations that our algorithm requires. CUBLAS is a CUDA implementation of BLAS which has been tuned to provide good performance across a variety of GPUs. To avoid bias, we compare our performance with that of a tuned CPU BLAS library. The linear algebra routines from the BLAS libraries used in both the CPU and GPU implementations were identical, with the only difference being the algorithmic designs each version used in order to extract performance out of their respective architectures. The main linear algebra routines used are *sgemv*, *saxpy*, *sdot* and *snrm2*. These routines are frequently used basic linear algebra functions that were developed to provide building blocks for larger applications. If $A = D * D^T$ with D representing the term-document matrix, only A needs to be stored in memory on the GPU, along with a few vectors which are a fraction of the data size that D is. This is very advantageous, as the memory on a GPU card can be a very limiting factor. The card used for our testing has 1GB of memory, which allows us to allocate about 950MB to use in our program. The extra memory which can not be allocated is due to a process behind the scenes preventing full allocation. The CUDA community currently regards this as a bug as it is unclear why this much memory is reserved for other uses. With 950MB of usable memory, we are able to allocate matrices that exceed 15000x15000. This is well within the matrix size that we are targeting our algorithm for.

After implementing the CPU and GPU based algorithm,

we timed each implementation for various matrix sizes. The computer testing the implementations has the following specifications: Dual 3.6 GHz Intel Pentium 4 CPUs, 3.00 GB of RAM, NVIDIA@8800gt with 1 GB device memory, 160GB hard drive. Matrix sizes were selected in an interval thought to clearly display the performance of each implementation. For each matrix size, three randomly generated matrices were used. For each of the three matrices, both CPU and GPU versions are run five times and the total time is averaged. This produces fifteen total runs for both the CPU and GPU at every matrix size interval. The times are then averaged for display purposes.

4. RESULTS

Our initial results can be seen in figure 3, which is for matrices up to 4000 x 4000. These initial results show a performance increase of two to six times. Figure 4 shows the average CPU and GPU run times for matrices that have dimensions divisible by 16. The tests were performed in the same manner as the original tests, with the only change being the matrix sizes. The GPU in this scenario is able to process the data between 4 times faster for a 1600 x 1600 matrix up to almost 7 times faster for a 5600 x 5600 matrix.

The CPU version takes about twice as long for the majority of matrices 1000 x 1000 and larger. For matrices smaller than this, the speed increase is less noticeable. For extremely small matrices (smaller than 600 x 600) the CPU version is faster than the GPU version. The reasoning for this is that using the GPU requires significant constant overhead, which can comprise a large percentage of the timings [9]. When matrix sizes are increased, the percentage of the total time that the overhead consumes is decreased. This is because the overhead time does not change, but the amount of computation being done significantly increases. Even for matrices near the high end of our selected sizes, the GPU seems to not be more than around twice as fast, except for a select few matrices.

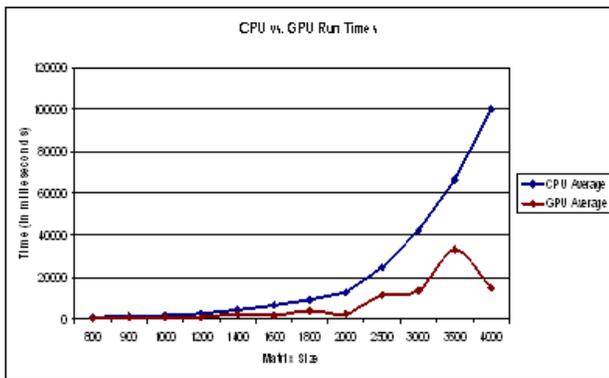


Figure 2: Test results for increasingly large matrices, up to 4000 x 4000

5. DISCUSSION

In our experiments, some select few matrices seem to be performing significantly better than average. These select few matrices all have one thing in common: The matrix dimensions are all divisible by 16. The reason behind the

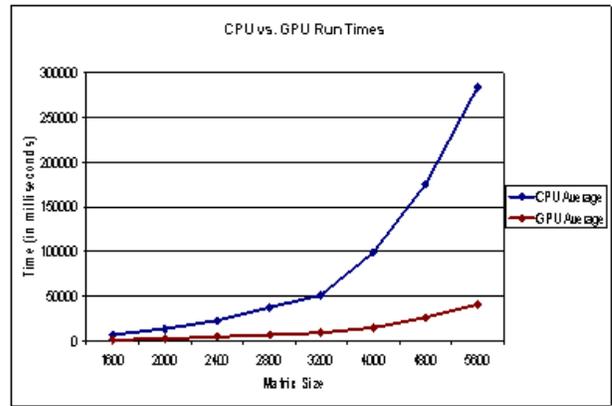


Figure 3: Test results for increasingly large matrices with dimensions divisible by 16, up to 5600 x 5600

significant speed increase lies in the architecture of the GPU. When the matrix is not divisible by 16, there are conflicts in shared memory regarding multiple threads accessing the same bank at the same time. This forces one thread to be put in a queue while the other thread is accessing the memory, increasing the amount of time for all memory accesses to be completed. This can be solved by using matrices with dimensions divisible by 16. CUBLAS will as a result provide coalesced memory access patterns, reducing time of the overall function greatly. This was not aware to us until after obtaining the results in fig. 3, which lead us to further test the implementations to verify this issue in fig. 4. It should be noted that the overall speeds for the CPU version did not vary from relative normal when the matrix dimensions were divisible by 16.

After testing our algorithm, it became evident what areas needed further research in order to produce an effective, fully implementable algorithm. The first area to be addressed is the accuracy of the algorithm. Currently, spurious eigenvalues and their resulting vectors are being created due to rounding errors. These are generated during the tridiagonalization process [6, 11, 14]. Two methods may be used in this situation. Either the spurious values can be removed after computation or reorthogonalization can be performed during computation [11, 14]. Removing spurious values after computation, while technically possible, is thought to be less effective than reorthogonalization. Reorthogonalization is the process of ensuring that the generated vectors are an accurate sub-space representation of the original matrix. For this reason, we will aim to implement partial reorthogonalization. Partial reorthogonalization has been proven to be able to preserve semiorthogonality [10, 14]. Partial reorthogonalization would allow us to stop our iteration sequence shorter than a simple Lanczos algorithm. These reasons make a partial reorthogonalization method necessary in a fully implementable LSA algorithm.

Another area of interest in future research would be further increases in the speed of the algorithm. Currently the GPU is only being used for the tridiagonalization computation. However the sstev BLAS routine, which computes the eigenvalues and eigenvectors of a tridiagonal matrix, is still being implemented on the CPU. Transitioning this to the GPU should further increase the performance of the algo-

rithm. The percentage of the total computation time that sstev occupies is only around 5% (for a 4000 x 4000 matrix). As matrix sizes grow, the total time that sstev requires will also grow, making a GPU version of the routine highly useful. This is especially true if the matrix sizes are approaching 10000 x 10000. Also, improvements upon the CUBLAS sgemv routine, which is a matrix-vector multiplication routine, would greatly increase performance as 90% of the GPU computation time is being occupied by this routine. This is entirely feasible as CUBLAS is not stated as being the optimal implementation.

A final area of research would be to attempt to implement the algorithm on a multiple GPU machine. Currently, in order to use multiple GPU's the algorithm must explicitly state how to use the GPUs. There is no automatic optimization allowing the use of multiple GPUs. For this reason, code needs to be modified for each individual problem. The use of multiple GPUs would be highly beneficial, as numerous computers are being released with multiple graphics cards built in. Even laptops are able to be ordered with multiple GPUs. Adding multi-GPU functionality can be tricky and does not always increase the speed of the code. This is due to the fact that there is very little effective communication done between GPUs. This makes sharing computation of problems very difficult. This is evident when computing a matrix-matrix product. In order to complete the computation, a GPU must have each matrix in its global memory. This adds a lot of increased overhead, as copying memory between GPUs currently requires to copy the data from GPU1 to the computer's memory then to GPU2. This is very slow and limits the effectiveness of multi-GPU implementation. Ideally, one GPU could send out small pieces of data for the other GPU(s) to compute. If this could be achieved, the full computational power of all GPUs could be utilized without worrying very much about transfer times. Achieving this level of performance is as much a hardware issue as it is a software issue, so our current implementation would require explicit declarations inside of the code.

6. CONCLUSIONS

In this research, we developed a parallel latent semantic analysis algorithm for the GPU. The results of our tests are very promising. The speed increase of the GPU based algorithm was 5-7 times for matrices with dimensions divisible by 16 and 2 times for matrices of other sizes. One solution to ensuring that all matrices have dimensions divisible by 16 is to add extra columns and rows of zeros to the matrix. The number of rows and columns to add would be equal to $x\%16$ where x is equal to the dimension size of the matrix. This number would always be between 1 and 15, requiring minimal computation time when adding this data. We hypothesize that adding these rows and columns would not add a noticeable increase to computation time and thus would still yield a speed increase of 5-7 times. With the GPU being so widespread in modern PC's, this algorithm is not just limited to expensive custom ordered workstations. Most mid range computers currently come with a graphics card, including laptops. This makes it possible to perform GPU-based LSA on a mobile computer. That being said, a top of the line desktop computer would be expected to see even better results. Currently, the NVIDIA®280GTX has a theoretical computation level of about 933 gflops. This is more than twice that of the card we used and costs ap-

proximately 40% more. With GPU computational power increasing at a higher rate than CPU computational power, it is very possible to see increased speed results in the near future [9]. We have shown that the GPU can be used to provide a performance increase to our algorithm. This should not only be useful to us, but provide evidence to further algorithmic development on the GPU.

7. ACKNOWLEDGMENTS

This research was done at Oak Ridge National Laboratory as part of the Department of Energy's Student Undergraduate Laboratory Internship program. Oak Ridge National Laboratory is managed by UT-Battelle LLC for the US Department of Energy under contract number DE-AC05-00OR22725. This work was supported in part by the Energy's Student Undergraduate Laboratory Internship program, Office of Naval Research (N0001408IP20066) and Oak Ridge National Laboratory Seed Money fund (3210-2276). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Oak Ridge National Laboratory, the Office of Naval Research, the Department of Energy or the U.S. government.

8. REFERENCES

- [1] N. Adams, G. Blunt, D. Hand, and M. Kelly. Data mining for fun and profit. *Statistical Science*, 15(2):111–131, 2000.
- [2] M. Berry. Large-scale sparse singular value computations. *The International Journal of Supercomputer Applications*, 6(1):13–49, 1992.
- [3] S. Dumais, G. Furnas, T. Lanerwester, R. Harshmandauer, S. Deerwester, and R. Harshman. Using latent semantic analyses to improve access to textual information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, Washington, D.C., United States, May 1988.
- [4] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 Coordinated and Multiple Views in Exploratory Visualization Conference*, Washington, D.C., United States, March 2005.
- [5] H.-P. Kersken and U. Kuster. A parallel lanczos algorithm for eigensystem calculation. Technical Report 310, University of Stuttgart, 1999.
- [6] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Natl. Bureau Stand.*, 45(1):255–282, 1950.
- [7] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(2), 2008.
- [8] Nvidia. Cuda:compute unied device architecture. Technical Report 2, NVIDIA, 2008.
- [9] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [10] C. Paige, B. Parlett, and H. V. der Vorst. Approximate solutions and eigenvalue bounds from

- krylov subspaces. *Numerical Linear Algebra with Applications*, 2(2):115–134, 1995.
- [11] B. Parlett and D. Scott. The lanczos algorithm with selective orthogonalization. *Mathematics of Computation*, 33(145):217–238, 1979.
- [12] P. Robert, S. Schoepke, and H. Bieri. Hybrid ray tracing - ray tracing using gpu-accelerated image-space methods. In *Proceedings of the 2007 International Conference on Computer Graphics Theory*, pages 305–311, Barcelona, Spain, 2007.
- [13] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.
- [14] H. Simon. The lanczos algorithm with partial reorthogonalization. *Mathematics of Computation*, 42(165):115–142, 1984.