Open access • Proceedings Article • DOI:10.1117/12.850538

CULA: hybrid GPU accelerated linear algebra routines — Source link

John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron Paolini ...+1 more authors

Published on: 23 Apr 2010 - Proceedings of SPIE (International Society for Optics and Photonics)

Topics: CUDA, Graphics processing unit, LU decomposition, Linear algebra and QR decomposition

CULA: Hybrid GPU Accelerated Linear Algebra Routines

John R. Humphrey *, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini, Eric J. Kelmelis
EM Photonics, Inc, 51 E Main St, Suite 203, Newark, DE, USA 19711

ABSTRACT

The modern graphics processing unit (GPU) found in many standard personal computers is a highly parallel math processor capable of nearly 1 TFLOPS peak throughput at a cost similar to a high-end CPU and an excellent FLOPS/watt ratio. High-level linear algebra operations are computationally intense, often requiring $O(N^3)$ operations and would seem a natural fit for the processing power of the GPU. Our work is on CULA, a GPU accelerated implementation of linear algebra routines. We present results from factorizations such as LU decomposition, singular value decomposition and QR decomposition along with applications like system solution and least squares. The GPU execution model featured by NVIDIA GPUs based on CUDA demands very strong parallelism, requiring between hundreds and thousands of simultaneous operations to achieve high performance. Some constructs from linear algebra map extremely well to the GPU and others map poorly. CPUs, on the other hand, do well at smaller order parallelism and perform acceptably during low-parallelism code segments. Our work addresses this via hybrid a processing model, in which the CPU and GPU work simultaneously to produce results. In many cases, this is accomplished by allowing each platform to do the work it performs most naturally.

Keywords: CULA, graphics processing unit, GPU, accelerated linear algebra, parallel computing

1. INTRODUCTION

The modern graphics processing unit (GPU) found in many standard personal computers is a highly parallel math processor capable of nearly 1 TFLOPS peak throughput at a cost similar to a high-end CPU and an excellent FLOPS/watt. ratio There is strong desire to utilize such power, but it can be difficult to harness given the features and limitations of the platform. As such, libraries can be of great utility, allowing novices and experts alike to access high computational performance without knowledge of GPU programming. Some routines such as FFTs are quite common in scientific and numerical computing and it would be wasteful for each user to implement such routines that could instead be provided in a centralized library. Moreover, a library routine that is tuned by an expert will often outperform and be more feature robust, allowing the user to instead focus on their particular area of expertise. In this paper we present CULA, a library of linear algebra routines developed using a hybrid computation model employing both CPU and GPU power.

When performing numerical analysis there are numerous recurring building blocks, many of which fall under the umbrella term of "linear algebra." At the lowest level, there are important fundamental operations dealing with the manipulation of matrices and vectors, typically in simple ways such as multiplication and addition. Building a level upwards are a series of algorithms that cover a broad scope of concepts including, but not limited to: linear system solution, various decompositions (QR, SVD), eigenproblem analysis, and least squares solution. Such operations have nearly limitless applications, such as: electromagnetic analysis, financial computations, image processing, and statistics.

Over time, the numerical computing community has settled on a tiered representation of linear algebra operations, as described above. This scheme is embodied in the Basic Linear Algebra Subprograms (BLAS)[1] and Linear Algebra Package (LAPACK)[2] libraries. These have also become standardized interfaces, with the typical usage scenario being to use a separate offering that has been specially tuned or completely rewritten for a given platform. Our offering, CULA, is a product of this nature. Specifically, it is a unified BLAS/LAPACK package that is tuned for the hybrid CPU/GPU machine.

* humphrey@emphotonics.com; phone 1 302 456-9003; fax 1 302 456-9004; emphotonics.com

2. CULA

CULA is a high-performance linear algebra library that executes in a unified CPU/GPU hybrid environment. In this section, we discuss the functions that CULA provides and the interfaces through which a developer can integrate CULA into his or her code. We follow this with an introduction to some the specialized techniques employed by CULA to obtain significant speedups over existing packages.

CULA features a wide variety of linear algebra functions, including but not limited to, least squares solvers (constrained and unconstrained), system solvers (general and symmetric positive definite), eigenproblem solvers (general and symmetric), singular value decompositions, and many useful factorizations (QR, Hessenberg, etc.) All such routines are presented in the four standard data types in LAPACK computations: single precision real (S), double precision real (D), single precision complex (C), and double precision complex (Z).

We support a number of methods for interfacing with CULA. The two major interfaces are Host and Device which accept data via host memory and device memory, respectively. The Host interface boasts high convenience while the Device interface is more manual but can avoid data transfer times.[†] Additionally there are facilities for interfacing with MATLAB and the FORTRAN language. Lastly is a special interface, called the Bridge interface, which aids in porting existing codes that employ Intel MKL, AMD ACML, and Netlib CLAPACK.

CULA employs several specialized techniques in order to attain its speedups. In the following sections, we describe two of the major internal features that are key contributors to the performance of this package: the hybrid execution model and low-level BLAS improvements.

3. HYBRID PROCESSING MODEL

In the GPU-computing field, a hybrid code such as CULA, is one that utilizes both CPU and GPU for its computation. CULA, as it exists today, differs somewhat from our original intent at the outset of development, which was to create a purely GPU-based library. We discovered that the GPU's poor performance for certain types of operations made it very difficult to achieve speedups over the CPU. For example, the LU decomposition features a number of "panel factorizations" and a number of BLAS routines. The panel factorize when implemented on the GPU would often result in the GPU being a *slowdown* compared to the CPU.

As suggested in the literature[3], it proves worthwhile to bring the panel back to the CPU for processing. The total time of *transfer+factorize+transfer* will often be shorter than the time for the pure GPU version. This happens because the GPU will be asked to perform operations that it does not excel at - for instance, the scan to find the appropriate pivot element for the LU pivoting operation.

Using the CPU for processing these operations introduces a second chance for optimization, which is to *overlap* the operations. We have produced a thorough treatment on this topic in[4], but it is also mentioned here for completeness. The notion is that while the panel is being transferred to and factorized by the CPU, the GPU can continue doing other operations. This is possible so long there is work that doesn't immediately depend on the CPU results, but for many linear algebra algorithms this is often the case. The result is that the work shifted to the CPU essentially becomes *free* in terms of overall time.

In the end, these two concepts are key to performance. By allowing the CPU and GPU to perform operations for which they are naturally well suited we can avoid a bottleneck. Furthermore, by overlapping these we can then leverage the power of both platforms simultaneously.

[†] See our paper, "Analyzing the Impact of Data Movement on GPU Computations" (reference #4) for a thorough discussion.

4. BLAS-LEVEL IMPROVEMENTS

LAPACK style computing often concentrates the so-called "heavy lifting" into calls performed by the underlying BLAS layer. This requires a very highly tuned BLAS, which is a feature on many computing platforms. Almost every high level function in the LAPACK library is comprised of low level linear algebra building blocks know as BLAS. These routines contain the vector and matrix operations such as vector dot product, matrix-vector multiplication, matrix-matrix multiplication, and triangular matrix solve. Since these functions are critical to having a high performance LAPACK library, many hardware vendors provide their own tuned BLAS library to fully utilize their hardware. For example, Intel provides a Core architecture optimized library through MKL BLAS and NVIDIA provides a G200 architecture optimized BLAS library with CUBLAS.

NVIDIA provides their CUBLAS offering packaged with their GPU development tools[5], but this library has been found to be deficient by the literature and by our own experiments.[6] There are two classes of ill-performing routines that we have identified. The first class is when a routine simply was not tuned for the sizes and parameters commonly used in LAPACK computations. The second class is when the parallel implementation used by CUBLAS is not well suited to the GPU platform - a modified algorithm can greatly improve performance. We will describe examples of both of these, below.

When examining CUBLAS's performance, we identified the general matrix-vector multiplication routine to have sub-optimal performance considering the parallelism of the algorithm. With this in mind, we developed a highly tuned matrix-vector routine to fully exploit the parallelism and memory hierarchy of the GPU. In matrix-vector multiplication, every row of the matrix can calculate its contribution to solution vector in parallel. Additionally, in an effort to maximum memory reuse, the input vector can be shared amongst the parallel row calculations. Utilizing these two concepts through careful GPU thread mapping and shared memory reuse, we developed a solution that performs up to 50% better than NVIDIA's general matrix-vector routine and up to 300% better for transposed matrix-vector multiplication. This low level acceleration accounts for a 25% speedup in LAPACK routines heavy in matrix-vector operations.

Matrix-matrix multiplication is widely regarded as the most important routine in the BLAS library. From a performance standpoint, matrix-matrix multiplication is a highly parallel algorithm with a very large amount of memory reuse. This allows the routine to perform a very large amount of floating point operations per second. Knowing that matrix-matrix multiplication is a high performance operation, many LAPACK routines are written such that operations are pushed into matrix-matrix multiplications through various blocking schemes and algorithms. Since matrix-matrix multiplication is the most critical component in the BLAS library, it was an obvious candidate for an in-depth performance examination in the CUBLAS library.

Matrix-matrix multiplication can be divided into four distinct varieties: panel-rectangle multiplication, panel-panel multiplication, rectangle-panel multiplication, or rectangle-rectangle multiplication. In these cases, a panel is a skinny matrix that is much larger in one direction while a rectangle matrix is arbitrarily shaped. The CUBLAS library achieves very high performance in rectangle-rectangle multiplication; however, the throughputs of the panel varieties are typically much lower. As the panel variety is necessary for our work, we implemented our own matrix-matrix multiplication routine that is specifically optimized for panel multiplication cases. This is a critical optimization because a number of widely used LAPACK functions rely heavily on panel multiplication, the most common of which is LU decomposition (*getrf* routine). Our custom matrix-matrix multiplication routines achieve speedups of 10% to 30%, when compared to CUBLAS for panel sizes of 32 and 64. This speedup translates to approximately a 10% speedup for many LAPACK routines that are heavily dependent on panel based matrix-matrix multiplication.

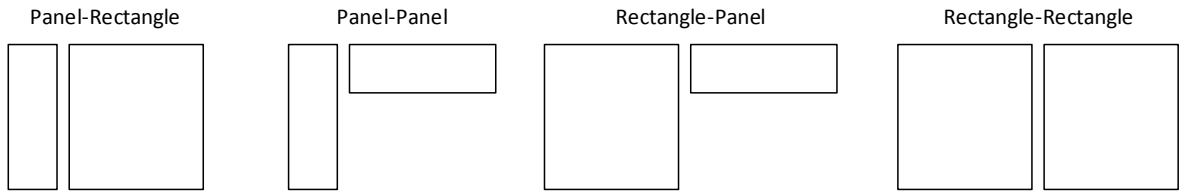

Figure 1 - Showing the types of matrix-matrix multiply inputs. NVIDIA CUBLAS was optimized primarily for the 4th case shown, where there are two large rectangular inputs. Our work covers the other cases, where one or both of the inputs is "panel" shaped. This is a critical usage pattern for LAPACK operations.

5. LIBRARY-BASED COMPUTING CONCERNS

As a provider of a library that falls into a fairly novel space like hybridized accelerated computing, we face some unique concerns. For instance, the LAPACK interface which we have attempted to follow was designed during an era in which there was only one kind of memory and it was considered to be a very scarce resource. The GPU hybrid model involves two types of processors (one of which, the GPU, might not even be present in the machine) and two types of memory. We will discuss this concern from both a usability perspective and then from a technological perspective below.

5.1 LAPACK Interface

Beyond the inputs and outputs of a particular routine, the LAPACK interface often requires the user provide a workspace region. One of the changes we have made from the LAPACK interface is in the elimination of these workspace parameters. For many routines, this will reduce the parameters by 2 and additionally reduce workspace lookups and allocations, greatly simplifying the code:

Table 1 - Workspace vs. No Workspace. The code shown is a workspace query (essentially asking the routine how much additional memory it requires) followed by the actual call. Thus many operations are two stage with a memory allocation between them. Four lines of code becomes one line, and the shorter version is less error prone.

With Workspace	`LWORK = -1;`
	`sgels(&TRANS, &M, &N, &NRHS, A, &LDA, B, &LDB, &QUERY, &LWORK);`
	`WORK = (float*)malloc((size_t)QUERY*sizeof(float));`
	`LWORK = (size_t)QUERY;`
	`sgels(&TRANS, &M, &N, &NRHS, A, &LDA, B, &LDB, WORK, &LWORK);`

No Workspace	`sgels(&TRANS, &M, &N, &NRHS, A, &LDA, B, &LDB);`

We made this decision for several reasons. As Table 1 listed, this decision had a direct impact on the number of calls to CULA, which in turn reduces the potential for user error. Beyond this, however, the decision was also approached due to concerns related to complications arising from implementing these routines on the GPU and the distinction that the GPU programming model places on host and GPU memory. In many instances, a combination of both host and GPU workspaces were required to allow a certain function to meet its performance goals. Rather than add both host-side and GPU-side parameters to an interface for which many users already make mistakes, we chose to eliminate the notion of explicit workspaces entirely.

In removing workspaces as a parameter from our interface, we shifted the responsibility of workspace allocation and tracking from the user to our library. While this decision has all of the benefits listed above, it does yield introduce performance considerations when compared with the traditional approach. When LAPACK was first designed, a computer's main memory was an extremely scarce resource. By allowing the user to specify workspaces, the LAPACK designers entrusted these users with the task of best managing this scarce resource. This arrangement could allow a user to utilize the knowledge of the algorithm they are implementing to reuse a single workspace in the case that the user were to make several successive calls for which a workspace is required. With CULA's approach of tracking their workspace internally, however, this approach prevents the user from explicitly using a workspace as intelligently as possible. Instead, because CULA is now responsible for workspaces, any decisions about workspace optimizations must

be made internally to the CULA interface. To avoid the cost of repeated workspace allocations, we utilize memory pools.

5.2 Memory Pools

Typically, when a developer has faced a resource that is used often but is expensive to allocate, developers have turned to pools. Whether it is memory or threads, pools can provide an effective solution to avoiding the repeated cost of an allocation.

The risk in writing a pool is that the developer's pool will perform less effectively than the interface that is provided by the allocation system. For example, consider that the traditional *malloc* operation may be implemented atop a segment of memory that has already been pooled by a lower level allocator. For many applications, creating a custom pool doesn't make sense, as the OS developer has put many more hours into the optimization of this routine than an individual developer can likely provide. Additionally, a user must worry about concerns over fragmentation and partitioning, issues that he or she need not consider when using the operating system's facilities.

GPU memory, on the other hand, has different costs and implementation details when compared with host memory. The application of memory pools must therefore be reconsidered with these new costs in mind. We completed a study of the costs of allocation of CPU and GPU memory by comparing the time to allocate various sizes of memory with the *malloc* and *cudaMalloc* routines.

Table 2 - Allocation Cost CPU vs. GPU

Size	Time CPU (s)	Time GPU (s)
1 KB	3.26e-06	9.10e-04
2 KB	2.20e-06	9.09e-04
4 KB	2.38e-06	9.06e-04
8 KB	3.56e-06	9.05e-04
16 KB	3.20e-06	9.12e-04
32 KB	3.20e-06	9.05e-04
64 KB	5.98e-06	9.08e-04
128 KB	3.70e-06	9.07e-04
256 KB	8.72e-06	9.08e-04
512 KB	8.34e-06	9.38e-04
1 MB	6.48e-06	9.89e-04
2 MB	9.36e-06	1.09e-03
4 MB	1.26e-05	1.30e-03
8 MB	1.16e-05	1.73e-03
16 MB	1.21e-05	2.62e-03
32 MB	1.21e-05	4.39e-03
64 MB	1.33e-05	7.94e-03
128 MB	1.35e-05	1.52e-02

As Table 2 shows, the *cudaMalloc* allocator is two to three orders of magnitude more expensive on an allocation than the traditional *malloc* operation. As sizes smaller than 1MB show, the *cudaMalloc* operation is dominated by overhead, it is not until memory sizes of approximately 2 MB that the allocation cost begins to grow beyond this overhead.

Even at relatively large sizes, the *malloc* operation takes a minimal amount of time. The CUDA allocation, on the other hand, takes a non-trivial amount of time. As such, these costs make a strong case for pooling GPU memory while leaving the host memory store to the operating system. Once the memory is pooled, GPU allocation time drops to a trivial amount.

In CULA, we use a custom memory pool for internal GPU memory allocations. This allows us to efficiently support the removal workspaces and additionally provides us with the flexibility to use additional workspaces where they have not been traditionally used by the LAPACK interface. By explicitly controlling workspaces, we can always ensure that they are used in an efficient manner.

6. BENCHMARKING

In this section we present collected benchmarking statistics for CULA. We surveyed a wide array of routines and have noted the results in **Error! Reference source not found.**, below. The benchmark system consists of an NVIDIA Tesla C1060 GPU, an Intel Core i7 920 processor, and 6 GB RAM. All problems fit within memory.

These results include all optimizations described above, including hybridized processing, memory pooling, and improved underlying BLAS operations.

Our CULA routines were compared to the Intel MKL 10.2 offering, running using all four processor cores in the system. It should be noted that MKL 10.2 has been fully optimized for the Core i7 processor on which it was running. As such, this benchmark is of the highest possible quality.

The typical measured problem was a square matrix sized around 8000x8000. In some cases, there were different circumstances, i.e. for routines that are intended to function on non-square matrices.

In all cases, the speedup was calculated by dividing the wall-clock time of the competitor (MKL) by the wall-clock time of the corresponding CULA routine. The data inputs to each routine were identical, so the routines will behave in identical ways internally.

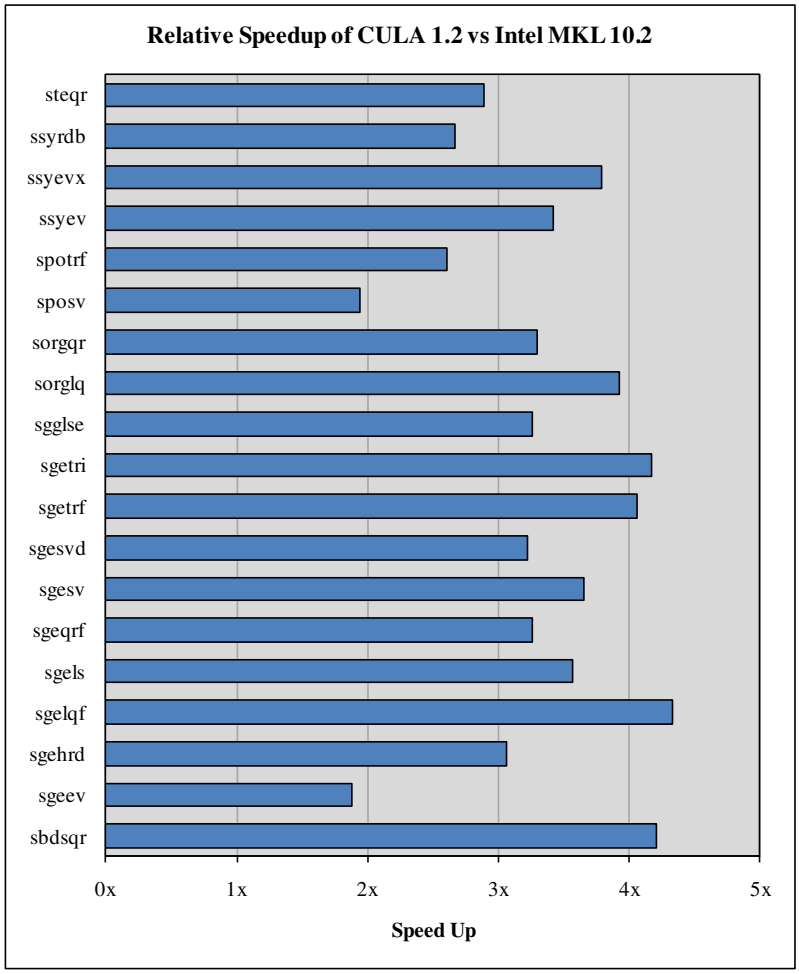

Figure 2: Benchmarking data for a large number of routines. The CPU platform is an Intel Core i7 920. The GPU is a NVIDIA Tesla C1060.

Double precision routines have been benchmarked as well, and performance has shown to be lower. The typical speedup for these is in the 1.5x to 2x region, which is often not significant enough to justify the use of the GPU. The differential is due to the architecture of the GT200 series of GPUs, which have a very low ratio of double-precision hardware to single-precision. This should be cured in the following generation which is referred to as "Fermi." The Fermi chip brings double precision performance much closer to single-precision[7], but benchmark figures were not available at time of publication.

7. CONCLUSION & FUTURE WORK

In this paper we presented CULA, a robust linear algebra library for computations in a hybrid CPU/GPU environment. The GPU is an attractive candidate for performing the highly parallel operations that arise in such computations, and the CPU is very strong at handling the more irregular or serial operations. Combining the two and using both at once is thus a good solution and leads to strong speedups. Our work on CULA has resulted in a comprehensive variety of accelerated mathematical functionality.

While the GPU provides effective speedups for problems of moderate size, small problems will cause the GPU to be outperformed by a modern CPU. This result is an inherent limitation of the GPUs status as a co-processor. This is often not a concern, as for many of these smaller problems, the runtime is often not significant enough to warrant acceleration. A problem arises, however, when many of these small problems need to be computed in parallel. This need re-introduces the GPU as an effective candidate for acceleration, as the nature of the problem changes from a small, serial one to a batched, parallel one. While existing linear algebra frameworks do not support this style of batched computation, in the future we will implement this as a first-class processing paradigm of our library so as to better utilize the full capabilities of the GPU and provide significant speedups for this class of problem.

REFERENCES

[1] www.netlib.org/blas/

[2] www.netlib.org/lapack/

[3] Volkov, V., and Demmel, J. W., "Benchmarking GPUs to tune dense linear algebra," In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-11.C. Jones, Director, Miscellaneous Optics Corporation, interview, Sept. 23, (2008).

[4] Price, D. K., Humphrey, J. R, Spagnoli, K. E., and Paolini, A. L. "Analyzing the Impact of Data Movement on GPU Computations," Presented at SPIE Defense and Security Symposium, April, (2010).

[5] http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUBLAS_Library_2.3.pdf

[6] Li, Y., Dongarra, J., Tomov, S. "A note on auto-tuning GEMM for GPUs," University of Tennessee Computer Science Technical Report, UT-CS-09-635, January 12, (2009).

[7] http://www.nvidia.com/object/fermi_architecture.html