THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
24(4) 511–515

DOI: 10.1177/1094342010385729
hpc.sagepub.com
SAGE

An Improved Magma Gemm For Fermi Graphics Processing Units

Rajib Nath[1], Stanimire Tomov[1], and Jack Dongarra[1,2,3]

Abstract

We present an improved matrix–matrix multiplication routine (General Matrix Multiply [GEMM]) in the MAGMA BLAS library that targets the NVIDIA Fermi graphics processing units (GPUs) using Compute Unified Data Architecture (CUDA). We show how to modify the previous MAGMA GEMM kernels in order to make a more efficient use of the Fermi's new architectural features, most notably their extended memory hierarchy and memory sizes. The improved kernels run at up to 300 GFlop/s in double precision and up to 645 GFlop/s in single precision arithmetic (on a C2050), which is correspondingly 58% and 63% of the theoretical peak. We compare the improved kernels with the currently available version in CUBLAS 3.1. Further, we show the effect of the new kernels on higher-level dense linear algebra (DLA) routines such as the one-sided matrix factorizations, and compare their performances with corresponding, currently available routines running on homogeneous multicore systems.

Keywords

GPU BLAS, CUDA matrix mutiply, Fermi, dense linear algebra, hybrid computing

1 Introduction

Matrix–matrix multiplication is a fundamental linear algebra routine. Many numerical algorithms can be expressed in terms of General Matrix Multiply (GEMM), or at least designed to partially use it. Numerous examples from the area of dense linear algebra (DLA) can be seen in the LAPACK library (Anderson et al., 1999). The technique to achieve that in DLA is based on *delayed updates*: the application of basic linear transformations, e.g. expressed in terms of matrix–vector multiplications, are delayed and accumulated so that they are applied later at once as a matrix–matrix multiplication.

The importance of having algorithms rich in GEMM is because the computational intensity of GEMMs can be increased by increasing the sizes of the matrices involved, which in turn is crucial for the performance on modern architectures with memory hierarchy. Major hardware vendors such as Intel, IBM, AMD, and NVIDIA maintain their own highly optimized GEMM routines, e.g. included into their BLAS implementation libraries: MKL, ESSL, ACML, and CUBLAS correspondingly. Non-vendor optimized implementations for various architectures are also available, such as ATLAS (Whaley et al., 2001) and GotoBLAS[1].

In the area of graphics processing unit (GPU) computing, "high-performance" GEMM implementations were not possible in the "early" GPUs (Fatahalian et al., 2004). The reason is that they did not have developed memory hierarchy and, therefore, the GEMM's performance peak was memory bound. With the introduction of memory hierarchy, e.g. the shared memory in the NVIDIA Compute Unified Data Architecture (CUDA) GPU architectures[2], this has changed. Algorithms that would reuse data brought into the shared memory were developed to achieve high, compute-bound performance (Volkov and Demmel, 2008). The performance of these algorithms relied on a number of very well-selected parameters and optimizations (Wolfe, 2008).. Subsequent work in the area managed to "automate" or "auto-tune" the selection of these parameters and optimizations used, to quickly find the best performing implementations for particular cases of GEMM (Li et al., 2009; Nath et al., 2010).

The NVIDIA Fermi architecture introduced new features to CUDA (NVIDIA, 2009). Although software developed for the previous NVIDIA hardware would run on the newer Fermi hardware, the performance could be enhanced sometimes significantly by the use of the new architectural features. We show that this is the case with the previous state-of-the-art GEMM implementations. Moreover, we have found that even the auto-tuning frameworks cannot

[1]University of Tennessee, USA
[2]Oak Ridge National Laboratory, USA
[3]University Of Manchester, UK

Corresponding author:
Stanimire Tomov, University of Tennessee, 1122 Volunteer Boulevard, Suite 203 Knoxville, TN 37996-3450, USA
Email: tomov@eecs.utk.edu

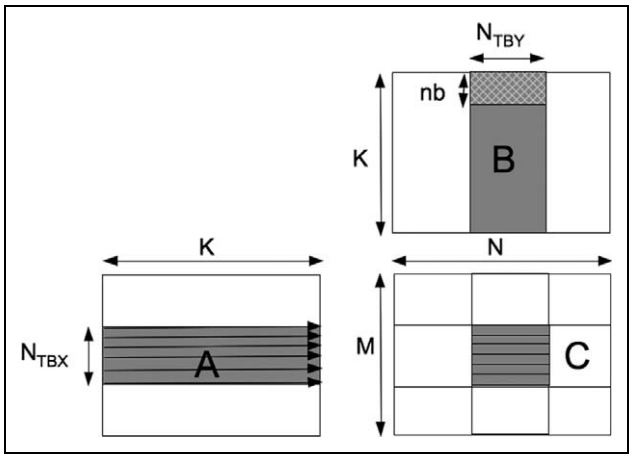

Figure 1. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for GTX280.

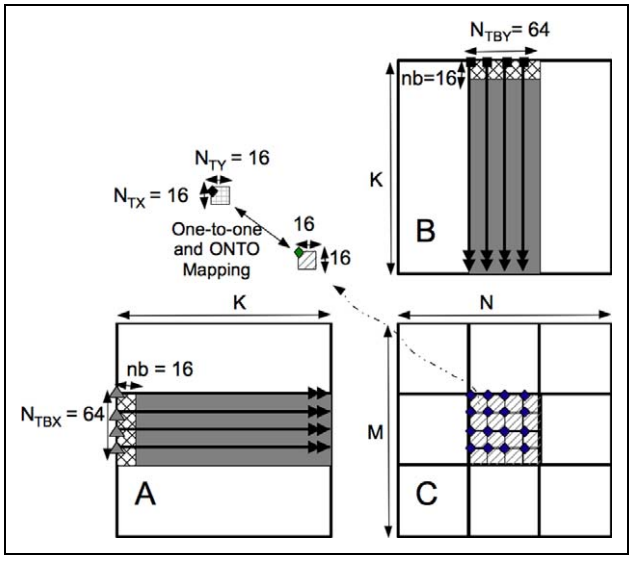

Figure 2. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for Fermi.

find the new "optimal" implementations, simply because their search space did not consider the newly introduced features in the architecture.

Section 2 is an overview of the GEMM for the previous generation GPUs. Section 3 gives the main contribution of this paper: an improved GEMM for the Fermi architecture. Section 4 shows the effect of the improved kernels on higher-level DLA routines such as the one-sided matrix factorizations from the MAGMA library (Tomov et al., 2009), and compares their performances with corresponding, currently available routines running on homogeneous multicore systems. Finally, Section 5 is on conclusions and future work.

2 GEMM for the NVIDIA GTX280

This section gives an overview of the GEMM targeting the old generation of GPUs, e.g. the GTX280 GPU. We consider a GEMM algorithm (Volkov and Demmel, 2008), parametrized to facilitate auto-tuning (Li et al., 2009; Nath et al., 2010) for the case $C := \alpha AB + \beta C$. Uniform rectangular sub-matrices of the resulting matrix C are computed in parallel (see Figure 1 for an illustration). Following CUDA terminology, we refer to these sub-matrices as thread blocks (TBs). The size of a TB, denoted further by $N_{TBX} \times N_{TBY}$, must be hard coded and its selection is crucial for the performance. The computation of each TB is done by a number of threads. The number is hard coded and denoted further by $N_T := N_{TX} \times N_{TY}$. An example (Volkov and Demmel, 2008) is $N_{TBX} \times N_{TBY} \equiv 64 \times 16$ and $N_T \equiv 64$.

For simplicity, take $N_T := N_{TBX}$. Then, each thread is coded to compute a row of the sub-matrix of C. To do that, it accesses the corresponding row of A (as indicated by an arrow in Figure 1), and uses the $K \times N_{TBY}$ sub-matrix of B for computing the final result. The TB computation is blocked, which is crucial for obtaining high performance. In particular, sub-matrices of B of size $nb \times N_{TBY}$ are loaded into shared memory and multiplied nb times by the corresponding $N_{TBX} \times 1$ sub-matrices of A. The $N_{TBX} \times 1$ elements are loaded and kept in registers while multiplying them with the $nb \times N_{TBY}$ part of B. The result is accumulated to the resulting $N_{TBX} \times N_{TBY}$ sub-matrix of C, which is kept in registers throughout the TB computation. All memory accesses are coalesced.

Kernels for various N_{TBX}, N_{TBY}, N_{TX}, N_{TY}, and nb are automatically generated in MAGMA BLAS to select the best performing for a particular architecture and for particular GEMM parameters (Nath et al., 2010). The theoretical peak of the GTX280 is 936 GFlop/s in single precision (240 cores $\times$ 1.30 GHz $\times$ 3 floating point instructions per cycle). The kernel described achieves up to 40% of that peak.

3 GEMM for Fermi

Many of the architectural changes in Fermi are related to scaling up the compute capabilities of the previous generation of NVIDIA GPUs (NVIDIA, 2010), e.g. increased shared memory, number of registers, number of CUDA cores in a multiprocessor, etc. The algorithm from Section 2, designed for the older NVIDIA GPUs, can be automatically adjusted to account for those changes in the Fermi GPUs. In addition, there are other changes that must be exploited (for performance) which, unfortunately, is impossible to accomplish by simply auto-tuning the old algorithm. For example, these are the changes that are related to added cache memories, and most importantly, that the latency to access registers and shared memory were comparable in the previous generation of NVIDIA GPUs, but not in the Fermi (where accessing data from registers is much faster). This motivates to add one more level of blocking in the algorithm, namely register blocking, to account for the added memory hierarchy. A way to do it is to have blocks of both matrices A and B first loaded into shared

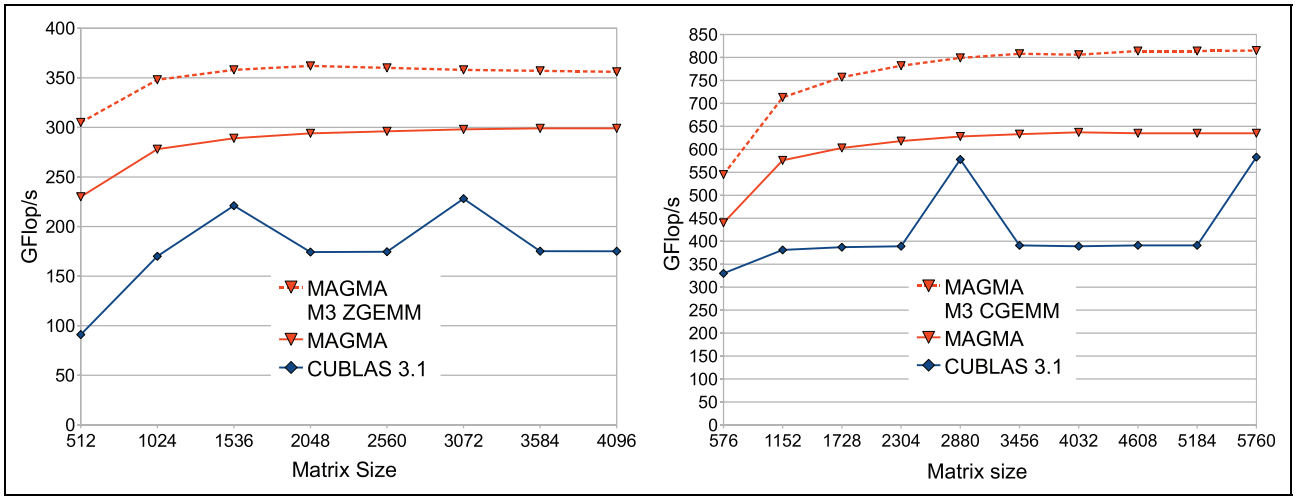

Figure 3. Performance of MAGMA BLAS DGEMM (left) and SGEMM (right) on Fermi (C2050). The performances achieved are correspondingly 58% and 63% of the theoretical peaks. Complex arithmetic versions of GEMMs are derived using the M3 method (Higham, 1992) and their performances are also shown, running at correspondingly 71% and 79% of the theoretical peaks. The performance peaks in CUBLAS 3.1 occur for matrix sizes divisible by their internal blocking sizes: 48 in double precision and 80 in single precision.

memory, and second, additionally block the computation with the matrices in shared memory by loading parts of them in registers to have reuse of the data in registers (versus reusing only data in the shared memory). Details on this new algorithm are given as follows.

The algorithmic view of the improved GEMM for Fermi is shown in Figure 2. Similarly to the old GEMM, the computation is divided into two-dimensional grid of TBs of size $N_{TBX} \times N_{TBY}$. Each TB is assigned to $N_T = N_{TX} \times N_{TY}$ threads. As mentioned above, sub-matrices of both A and B are loaded in shared memory. We take $N_{TBX} = N_{TBY} = 64$ and $N_{TX} = N_{TY} = 16$. With these parameter values, 16×16 threads will be computing 64×64 elements of matrix C. Hence, each thread will be computing 16 elements. The 64×64 block of matrix C is divided into 16 sub-blocks of dimension 16×16 as illustrated in Figure 2. Each of the 16×16 sub-blocks is computed by 16×16 threads, i.e. one element is computed by one thread. More precisely, element (x, y) of the 16×16 sub-block is computed by thread (x, y) of the 16×16 block of threads. All of the 16 elements computed by thread $(0, 0)$ are shown by diamonds in the figure. In summary, each thread will be computing a 4×4 matrix with stride 16. This distribution allows us to perform coalesced (parallel) writes of the final results from registers to the matrix C in global memory. Other distributions may not facilitate coalescent writes.

At each iteration of the shared memory blocking, all threads inside a TB load 64×16 elements of A and 16×64 elements of B to shared memory in a coalesced way. Depending upon $Op(A)$ and $Op(B)$, the 256 threads in the TB take one of the following shapes: 16×16 or 64×4. This reshaping helps coalesced memory access from global memory. The elements from matrices A and B, needed by thread $(0, 0)$, are shown by arrows. First, four elements from A (taken from the shared memory, shown with a gray triangle) and four elements from B (again from the shared memory, shown with a black rectangle in Figure 2) are loaded into registers. Then these 8 elements are used on 16 floating-point multiply-add (FMAD) operations. To get a further small performance increase, all of the accesses for matrices A and B are done through texture memory. This is a special memory with cached accesses. CUDA offers the ability to bind global memory, e.g. A and B, to it for direct access, avoiding the need to explicitly copy the global memory data into the texture memory. The performance of DGEMM in Fermi using this algorithm is shown in Figure 3, along with the DGEMM performance from CUBLAS 3.1. Note that the theoretical peak of the Fermi, in this case a C2050, is 515 GFlop/s in double precision (448 cores $\times$ 1.15 GHz $\times$ 1 instruction per cycle). The kernel described achieves up to 58% of that peak.

A similar idea is applied in single precision. The best blocking size is 96×96. The number of threads used is 16×16, where each tread computes 6×6 matrix with stride 16 using register blocking.

4 One-sided Factorizations on Fermi

In this section we discuss the performance of the one-sided matrix factorizations using the new kernels. Figure 4 compares the performance of LU factorization in double precision arithmetic from MAGMA on Fermi (C2050) with that of MKL 11.0, PLASMA (Agullo et al., 2009), and LAPACK on a 48-core system. The exact specifications are given in the figure. Note that the Fermi and the multicore system have the same theoretical peaks. The implementations of the LU factorizations in MAGMA, MKL and LAPACK use the same data layout and algorithm: LU with partial pivoting. The algorithm in PLASMA is different:

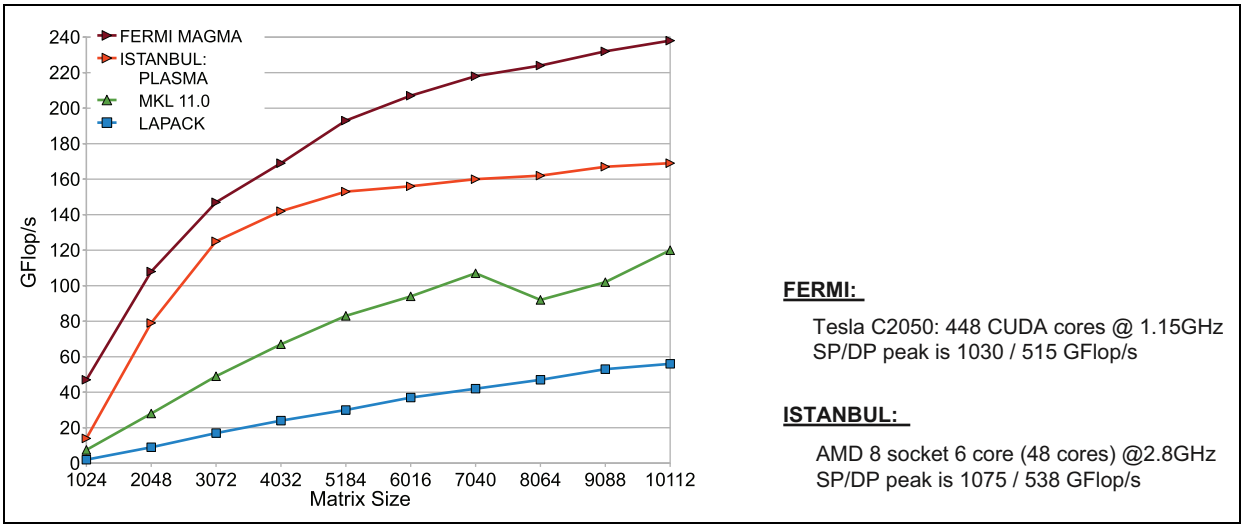

Figure 4. Performance of LU factorization in double precision: we compare MAGMA on Fermi (C2050) versus MKL 11.0, PLASMA and LAPACK on a 48-core system, having the same peak as a single Fermi GPU (C2050). The Fermi's host is a quad-core Intel Core 2 Q9300 @2.5 GHz.

LU with tile pairwise-pivoting on tile data layout. It is interesting to note that MAGMA achieves significantly higher percentage of the GPU's peak than the percentage of the peak that the other libraries achieve on the multicore system. That is, we showed a case where LU runs more efficiently on GPUs than on current, high-end homogeneous x86-based multicore systems. Moreover, GPUs have better power efficiency and better system cost. As an example, the cost of the 48-core system is approximately $\$30,000$ *vs* $\$3,000$ for the Fermi GPU and its host.

Similar performance is obtained on the QR and Cholesky factorizations. The performance in single precision is twice as high. Compared with other GPU libraries, e.g. the commercially available linear algebra library CULA 2.0, the results presented are currently about 65% faster [3].

5 Conclusions and Future Directions

The development of fast BLAS, and in particular GEMM, is crucial for the performance of many algorithms, and therefore is of extreme interest. We presented an improved GEMM algorithm for the Fermi GPUs, which significantly outperforms the currently available software. Moreover, this new kernel opens the possibility for further improvements, e.g. based on auto-tuning. Also, this GEMM can be used directly or auto-tuned for developing other GEMM-based Level 3 BLAS such as SYRK, TRSM, etc. An interesting (and straightforward) application was the development of complex GEMM using three real matrix multiplications (Higham, 1992) stabilityof to achieve up to 365 GFlop/s for ZGEMM (or 71% of peak) and 815 GFlop/s for CGEMM (or 79% of peak).

We also showed the effect of using the improved kernel on the performance of higher-level algorithms, e.g. the one-sided factorizations. MAGMA's LU is running 65% faster than the implementation in CULA 2.0, a commercial GPU-accelerated linear algebra library. Compared with vendor libraries for multicore x86-based systems, the results are similar: MAGMA's LU on single Fermi can significantly outperform the vendors' LU on high-end systems, such as the 48-core system in Figure 4. Further optimizations are possible, with one directions being tuning. For example, it is interesting to show that the performance can be as high as 300 GFlop/s for smaller matrices.

A general conclusion is that DLA has become a better fit for the evolving GPU architectures, to the point where DLA can run more efficiently on GPUs than on current, high-end homogeneous multicore-based systems. This progress has been partially enabled by the added memory hierarchy in the GPUs, which in effect enabled the development of fast GEMM. The current implementation achieves a higher fraction of the peak, namely 58% (and 63% in single), compared with the 40% on the previous generation of GPUs.

Acknowledgments

This work was supported by NVIDIA, Microsoft, the U.S. National Science Foundation, and the U.S. Department of Energy. We thank Everett Phillips and Massimiliano Fatica from NVIDIA for the useful discussions and optimization suggestions regarding the Fermi architecture.

Notes

1. See http://www.tacc.utexas.edu/tacc-projects/gotoblas2/GotoBLAS
2. See http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf
3. See http://www.culatools.com/features/performance/

References

Agullo, E., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Langou, J., Ltaief, H., Luszczek, P., and YarKhan, A. (2009). PLASMA users' guide. http://icl.cs.utk.edu/plasma/

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). LAPACK User's Guide, 3rd Ed. SIAM, Philadelphia, PA.

Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the efficiency of GPU algorithms for matrix–matrix multiplication. In Proceedings of HWWS'04. ACM Press, New York, pp. 133–137.

Higham, N. J. (1992). Stability of a method for multiplying complex matrices with three real matrix multiplications. *SIAM J. Matrix Anal. Appl.* **13**: 681–687.

Li, Y., Dongarra, J., and Tomov, S. (2009). A note on auto-tuning GEMM for GPUs. In Proceedings of ICCS'09. Springer-Verlag, Berlin, pp. 884–892.

Nath, R., Tomov, S., and Dongarra, J. (2010). Accelerating GPU kernels for dense linear algebra. In Proceedings of VECPAR'10, Berkeley, CA, 22–25 June 2010.

NVIDIA (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html

NVIDIA (2010). NVIDIA CUDA C Programming Guide, version 3.1.1. http://developer.nvidia.com/object/cuda_3_1_downloads.html

Tomov, S., Nath, R., Du, P., and Dongarra, J. (2009). MAGMA version 0.2 Users' Guide. http://icl.cs.utk.edu/magma

Volkov, V. and Demmel, J. (2008). Benchmarking GPUs to tune dense linear algebra. In Proceedings of SC'08. IEEE Press, Piscataway, NJ, pp. 1–11.

Whaley, R. C., Petitet, A., and Dongarra, J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* **27**(1–2): 3–35.

Wolfe, M. (2008). Special-purpose hardware and algorithms for accelerating dense linear algebra. *HPC Wire*, October. http://www.hpcwire.com/features/33607434.html

Author's Biographies

Rajib Nath received a Bachelor of Science in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in 2005 and a Master of Science in Electrical Engineering and Computer Science from the University of Tennessee, Knoxville in 2010. He joined the Ph.D. program at the UCSD Department of Computer Science and Engineering. His research focuses on high-performance computing. He is involved with the MAGMA (Matrix Algebra on GPU and Multicore Architectures) and PLASMA (Parallel Linear Algebra for Scalable Multicore Architectures) projects.

Stanimire (Stan) Tomov is Research Scientist at the Innovative Computing Laboratory (ICL) and Adjunct Assistant Professor in the Electrical Engineering and Computer Science Department at the University of Tennessee, Knoxville (UTK). He received his Bachelor and Master of Science degrees in Computer Science from Sofia University "St. Kliment Ohridski", Bulgaria, in 1994 and a Ph.D. in Mathematics from Texas A&M University in 2002. He held positions at the Lawrence Livermore National Laboratory and the Brookhaven National Laboratory before joining ICL. His research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing. Currently, he leads ICL projects on developing linear algebra libraries on emerging hybrid architectures.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow at Manchester University, and an Adjunct Professor in the Computer Science Department at Rice University. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research at the University of Tennessee which coordinates and facilitates IT research efforts at the University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high-quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports, and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches and in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.