A Note on Auto-tuning GEMM for GPUs

Yinan Li[1], Jack Dongarra[1,2,3], and Stanimire Tomov[1]

[1] University of Tennessee (USA)
[2] Oak Ridge National Laboratory (USA)
[3] University of Manchester (UK)

January 12, 2009

Abstract. The development of high performance dense linear algebra (DLA) critically depends on highly optimized BLAS, and especially on the matrix multiplication routine (GEMM). This is especially true for Graphics Processing Units (GPUs), as evidenced by recently published results on DLA for GPUs that rely on highly optimized GEMM [13, 11]. However, the current best GEMM performance, e.g. of up to 375 GFlop/s in single precision and of up to 75 GFlop/s in double precision arithmetic on NVIDIA's GTX 280, is difficult to achieve. The development involves extensive GPU knowledge and even backward engineering to understand some undocumented insides about the architecture that have been of key importance in the development [12]. In this paper, we describe some GPU GEMM **auto-tuning** optimization techniques that allow us to keep up with changing hardware by rapidly reusing, rather than reinventing, the existing ideas. Auto-tuning, as we show in this paper, is a very practical solution where in addition to getting an easy portability, we can often get substantial speedups even on current GPUs (e.g. up to 27% in certain cases for both single and double precision GEMMs on the GTX 280).
Keywords: Auto-tuning, matrix multiply, dense linear algebra, GPUs.

1 Introduction

Recent activities of major chip manufacturers, such as Intel, AMD, IBM and NVIDIA, make it more evident than ever that future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of two major types of components:

1. Multi/many-cores CPU technology, where the number of cores will continue to escalate while avoiding the power wall, instruction level parallelism wall, and the memory wall [2]; and
2. Special purpose hardware and accelerators, especially GPUs, which are in commodity production, have outpaced standard CPUs in performance, and have become as easy, if not easier to program than multicore CPUs.

The relative balance between these component types in future designs is not clear, and will likely vary over time, but there seems to be no doubt that future

generations of computer systems, ranging from laptops to supercomputers, will consist of a composition of heterogeneous components.

These hardware trends have inevitably brought up the need for updates on existing legacy software packages, such as the sequential LAPACK [1], from the area of DLA. To take advantage of the new computational environment, our current research shows that successors of LAPACK have to incorporate algorithms of three main characteristics: **high parallelism** (to efficiently account for the many-cores available), **reduced communication** (to account for the exponentially increasing memory wall), and **heterogeneity-awareness** (meaning, algorithms to be properly split between the components of the heterogeneous system so that the strengths of each component are properly matched to the requirement of the algorithm). This is reflected for example in the *Matrix Algebra on GPU and Multicore Architectures* (MAGMA) project [3], a recent effort on developing a successor to LAPACK but for heterogeneous/hybrid architectures, with current stress on Multicore + GPU systems.

Our overall goals, as related to auto-tuning and the MAGMA project, are to investigate opportunities for automating the transition to MAGMA and moreover, automating the tuning process of the newly discovered algorithms, both for the sake of productivity and for correctness in the new, complex, and rapidly changing computational environment. However, the techniques developed and incorporated in MAGMA so far show that a transition from LAPACK to MAGMA can not be done automatically or with minor modifications, as in many cases new algorithms that significantly differ from algorithms for conventional architectures will be needed [3]. Indeed, experiments show that the easy approach, that has been successful in the past, to use current LAPACK and simply call BLAS on the GPU leads to significant loss of performance (that can be of order 3× and higher). Nevertheless, this approach can lead to high performance, but only after some modifications and for routines that map well on the GPU, like Cholesky (e.g. Dongarra et al. [8] report up to 327 GFlop/s in single precision on a pre-released at the time NVIDIA T10P). Naturally, previous attempts to wrap some of the work needed in transitions like this in frameworks, have also failed to produce convincing results. For example the FLAME project [10], is a framework to facilitate the implementation of a class of DLA algorithms. Originally designed before the appearance of multicores, had to be continuously updated to meet the challenges of emerging architectures. Still, in the context of GPUs in particular, the latest results from FLAME developers [4] show a single precision Cholesky factorization running at up to only 156.2 Gflop/s. A single precision LU factorization from FLAME runs at up to 142 Gflop/s. Compare that with, accordingly, 315 Gflop/s and 309 Gflop/s [12] for the new algorithms (all these results are for the GTX 280). The main point here is that emerging architectures have motivated the development of new algorithms that have a much larger design space than previously needed. Early autotuners for example only targeted the BLAS, under the assumption that a few parameters (e.g. block sizes) were enough to capture enough of the algorithmic design space of higher level algorithms (LU, etc.) to attain a large fraction of peak performance. This assumption

was adequate to keep LAPACK and ScaLAPACK reasonably efficient for many years, but as described above it is far from adequate going forward. This is even more true for frameworks that are rooted in the old sequential environments preceding the introduction of multicores.

This brief outline motivates us to use auto-tuning (as a major component of the new MAGMA efforts) to keep up with the rapidly innovating hardware and continually growing design space so that we get to rapidly reuse, rather than reinvent, the new ideas. Indeed, the work that we describe in this paper on developing GEMM autotuners shows that we can significantly accelerate current results not only on emerging GPUs (e.g. when GPUs recently added support for double precision arithmetic) but also on current GPUs for which the algorithms were originally designed. Moreover, we have discovered that in the new hardware environment our design spaces critically depend not only on the architecture but also on problem sizes. The implication is that there may be different optimal algorithms for the same problem, and discovering these algorithms and their tuning on a case by case study may be impractical even for an expert. Auto-tuning is preferable.

The rest of the paper is organized as follows. In Section 2, we give background information on auto-tuning for DLA. Section 3 describe our GEMM autotuner for GPUs. Next are performance results (Section 4) and finally conclusions and future directions (Section 5).

2 Auto-tuning for CPUs

Automatic performance tuning (optimization), or auto-tuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [14, 7] and PHiPAC [5] are used to generate highly optimized BLAS. In addition, FFTW [9] is successfully used to generate optimized libraries for FFT, which is one of the most important techniques for digital signal processing. There are generally two kinds of approaches for doing auto-tuning, specifically model-driven optimization and empirical optimization. The idea of model-driven optimization comes from the compiler community. The compiler community has developed various optimization techniques that can be effectively used to transform code written in high-level languages such as C and Fortran to run efficiently on modern CPU architectures. These optimization techniques include loop blocking, loop unrolling, loop permutation, fusion and distribution, prefetching, and software pipelining. The parameters for these transformations such as the block size and the amount of unrolling are determined by analytical models, which are commonly used in the compiler community. While model-driven optimization is generally effective to make programs run faster, it may not give optimal performance to special-purpose libraries for linear algebra and signal processing. The reason is that analytical models used by compilers are only simplified abstractions of the underlying processor architectures, and they must be general enough to be applicable to all kinds of programs. Thus, the limited accuracy of analytical models makes the

model-driven approach not so attractive for the optimization of highly special kernels for linear algebra and signal processing, if the approach is solely used. In contrast to model-driven optimization, empirical optimization techniques generated a large number of parametrized code variants for a given algorithm and run these variants on a given platform to discover the one that gives the best performance. The effectiveness of empirical optimization depends on the chosen parameters to optimize, and the search heuristic used. A disadvantage of empirical optimization is the time cost of searching for the best code variant, which is usually proportional to the number of variants generated and evaluated. Contrarily, model-driven optimization has a O(1) cost, since the parameters can be derived from the analytical model. Therefore, a natural idea is to combine these two approaches, and it gives the third approach, a hybrid approach that uses the model-driven approach in the first stage to limit the search space for the second stage of empirical search.

Another aspect of the auto-tuning, besides the compiler and empirical tuning where an optimal computational kernel is generated as it is installed on one system, is **adaptivity** which can be regarded in various aspects [6]. The main aspect is to treat cases where tuning can not be restricted to optimizations at design time, installation time, or even compile time. In those cases, mechanisms of adaptivity can be incorporated in software, where tuning information captured in prior runs can be used to tune future runs.

With the success of auto-tuning techniques on generating highly optimized DLA kernels on CPUs, it is interesting to see how the idea can be used to generate near-optimal DLA kernels on modern high-performance GPUs.

3 GEMM Autotuner for GPUs

In this section we present our preliminary study on the idea of auto-tuning on modern GPUs. In particular, we design a GEMM "autotuner" for NVIDIA CUDA-enabled GPUs. Here autotuner refers to an auto-tuning system that automatically generates and searches a space of algorithms.

There are two core components in a complete auto-tuning system: a code generator and a heuristical search engine. The code generator generates parametrized code variants according to a pre-defined code template. The heuristical search engine then runs these variants and finds out the best one using a feedback loop, i.e., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

In [12], Volkov and Demmel presents kernels for single-precision matrix multiplication (SGEMM) that significantly outperforms CUBLAS on CUDA-enabled GPUs, using an approach that challenges those optimization strategies and programming guidelines that are commonly accepted. In this paper, we will focus on the GEMM kernel that computes $C = \alpha A \times B + \beta C$. Additionally, we will investigate auto-tuning on both single precision and double precision GEMM kernels (i.e., SGEMM and DGEMM). The SGEMM kernel proposed in [12] takes advantage of the vector capability of NVIDIA CUDA-enabled GPUs. The

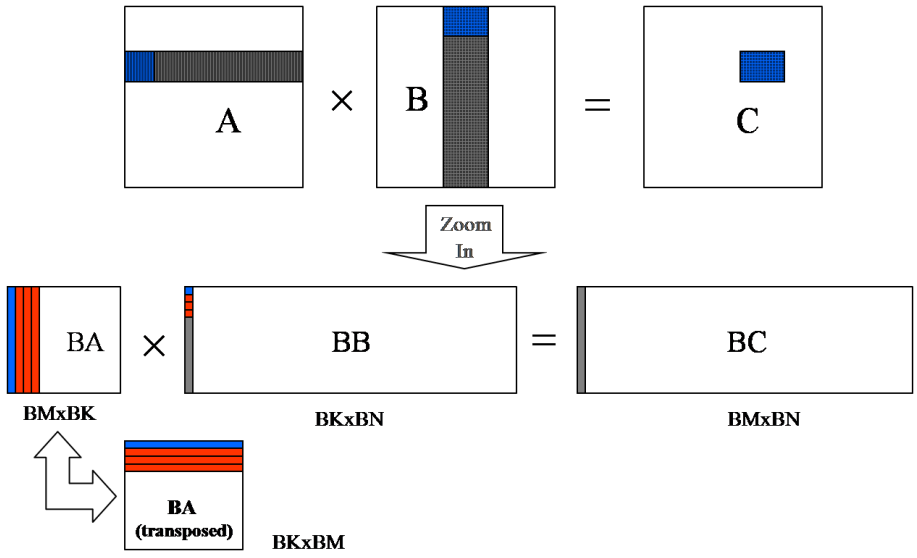

Fig. 1. The algorithmic view of the code template for GEMM.

authors argue that modern GPUs should be viewed as multi-threaded vector units, and their algorithms for matrix multiplication resemble those earlier ones developed for vector processors. We take their SGEMM kernel for computing $C = \alpha A \times B + \beta C$ as our code template, with modifications to make the template accept row-major input matrices, instead of column major used in their original kernel.

Figure 1 depicts the algorithmic view of the code templates respectively for both SGEMM and DGEMM. Suppose A, B, and C are M×K, K×N, and M×N matrices, and that M, N, and K are correspondingly divisible by BM, BN, and BK (otherwise "padding" by zero has to be applied or using the host for part of the computation). Then the matrices A, B, and C are partitioned into blocks of sizes BM×BK, BK×BN, and BM×BN, respectively (as illustrated on the figure). The elements of each BM×BN block of the matrix C (denoted by BC on the figure, standing for 'block of C') are computed by a $t_x \times t_y$ thread block. Depending on the number of threads in each thread block, each thread will compute either an entire column or part of a column of BC. For example, suppose BM = 16 and BN = 64, and the thread block has 16×4 threads, then each thread will compute exactly one column of BC. If the thread block has 16×8 threads, then each thread will compute half of a column of BC. After each thread finishes its assigned portion of the computation, it writes the results (i.e., an entire column or part of a column of BC back to the global memory where

the matrix C resides. In each iteration, a BM×BK block BA of the matrix A is brought into the on-chip shared memory and kept there until the computation of BC is finished. Similarly to the matrix C, matrix B always resides in the global memory, and the elements of each block BB are brought from the global memory to the on-chip registers as necessary in each iteration. Because modern GPUs have a large register file within each multiprocessor, a significant amount of the computation can be done in registers. This is critical to achieving near-optimal performance. As in [12], the computation of each block BC = BC + BA×BB is fully unrolled. It is also worth pointing out that in our SGEMM, 4 `saxpy` calls and 4 memory accesses to BB are grouped together, as in [12], while in our DGEMM, each group contains 2 `saxpy` and 2 memory accesses to BB. This is critical to achieving maximum utilization of memory bandwidth in both cases, considering that the different widths between `float` and `double`.

As outlined above, 5 parameters (BM, BK, BN, t_x, and t_y) determine the actual implementation of the code template. There is one additional parameter that is of interest to the actual implementation. This additional parameter determines the layout of each block BA of the matrix A in the shared memory, i.e., whether the copy of each block BA in the shared memory is transposed or not. Since the share memory is divided into banks and two or more simultaneous accesses to the same bank cause the so-called bank conflicts, transposing the layout of each block BA in the shared memory may help reduce the possibility of bank conflicts, thus potentially improving the performance. Therefore, the actual implementation of the above code template is determined or parametrized by 6 parameters, namely BM, BK, BN, t_x, t_y, and a flag *trans* indicating whether to transpose the copy of each block BA in the shared memory.

We implemented code generators for both SGEMM and DGEMM on NVIDIA CUDA-enabled GPUs. The code generator takes the 6 parameters as inputs, and generates the kernel, the timing utilities, the header file, and the Makefile to build the kernel. The code generator first checks the validity of the input parameters before actually generating the files. By validity we mean 1) the input parameters confirm to hardware constraints, e.g., the maximum number of threads per thread block $t_x \times t_y \leq 512$, and 2) the input parameters are mutually compatible, e.g., $(t_x \times t_y)\%\text{BK} = 0$, $\text{BM}\%t_y = 0$, and $\text{BN}\%t_x = 0$. By varying the input parameters, we can generate different variants of the kernel, and evaluate their performance, in order to identify the best variant. One way to implement auto-tuning is to generate a small number of variants for some matrices with typical sizes during installation time, and choose the best variant during run time, depending on the input matrix size.

4 Performance Results

The performance results in this section are for NVIDIA's GeForce GTX 280.

First, we evaluates the performance of the GEMM autotuner in both single and double precision. Figure 2, Left compares the performance of the GEMM autotuner in single precision with the CUBLAS 2.0 SGEMM for multiplying

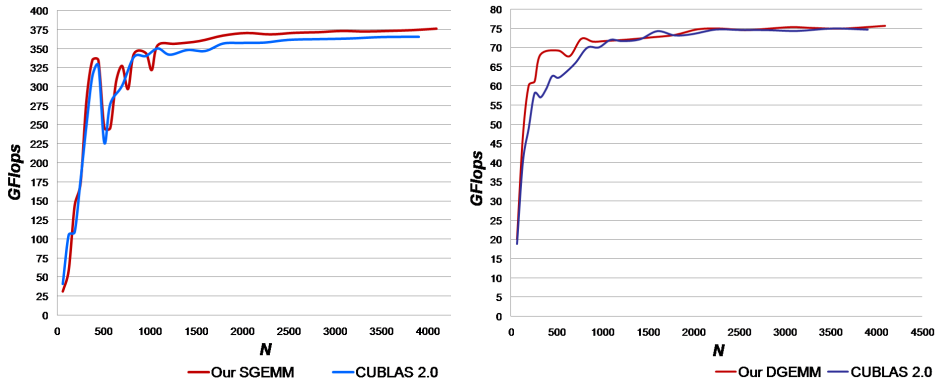

Fig. 2. Performance comparison of CUBLAS 2.0 *vs* auto-tuned SGEMM (left) and DGEMM (right) on square matrices.

square matrices. We note that both CUBLAS 2.0 SGEMM and our auto-tuned SGEMM are based on V.Volkov's SGEMM [12]. The GEMM autotuner selects the best performing one among several variants. It can be seen that the performance of the autotuner is apparently slightly better than the CUBLAS 2.0 SGEMM. Figure 2, Rigth shows that the autotuner also performs better than CUBLAS in double precision. These preliminary results demonstrate that auto-tuning is promising in automatically producing near-optimal GEMM kernels on GPUs. The most attractive feature of auto-tuning is that it allows us to keep up with changing hardware by automatically and rapidly generating near-optimal BLAS kernels, given any newly developed GPUs.

The fact that the two performances are so close is not surprising because our auto-tuned code and CUBLAS 2.0's code are based on the same kernel, and this kernel was designed and tuned for current GPUs (and in particular the GTX 280), targeting high performance for large matrices. In practice though, and in particular in developing DLA algorithms, it is very important to have high performance GEMMs on rectangular matrices, where one size is large, and the other is fixed within a certain block size (BS), e.g. BS = 64, 128, up to about 256 on current architectures. For example, in an LU factorization (with look-ahead) we need two types of GEMM, namely one for multiplying matrices of size N×BS and BS×N−BS, and another for multiplying N×BS and BS×BS matrices. This situation is illustrated on Figure 3, where we compare the performances of the CUBLAS 2.0 *vs* auto-tuned DGEMMs occurring in the block LU factorization of a matrix of size 6144 × 6144. The graphs show that our auto-tuned code significantly outperforms (up to 27%) the DGEMM from CUBLAS 2.0.

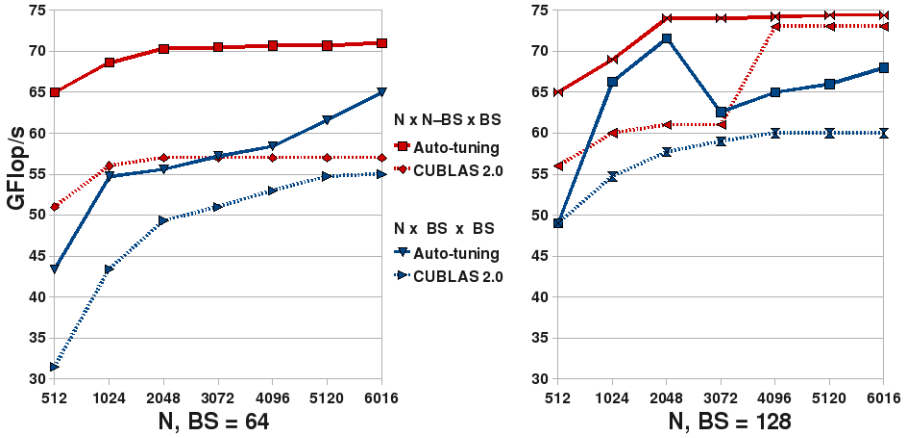

Fig. 3. Performance comparison of the auto-tuned (solid line) *vs* CUBLAS (dotted line) DGEMMs occurring in the block LU factorization (for block sizes BS = 64 on the left and 128 on the right) of a matrix of size 6144 × 6144. The two kernels shown are for multiplying N×BS and BS×N−BS matrices (denoted by N×N−BS×BS), and N×BS and BS×BS matrices (denoted by N×BS×BS).

5 Conclusions and Future Directions

We highlighted the difficulty in developing highly optimized codes for new architectures, and in particular GEMM for GPUs. On the other side, we have shown an auto-tuning approach that is very practical and can lead to optimal performance. In particular, our auto-tuning approach allowed us

- To easily port existing ideas on quickly evolving architectures (e.g. demonstrated here by transferring single precision to double precision GEMM designs for GPUs), and
- To substantially speed up even highly tuned kernels (e.g. up to 27% in this particular study).

These results also underline the need to incorporate auto-tuning ideas in our software. This is especially needed now for the new, complex, and rapidly changing computational environment. Therefore our future directions are, as we develop new algorithms (e.g. within the MAGMA project), to systematically define their design/search space, so that we can easily automate the tuning process as demonstrated in this paper.

Acknowledgments. Part of this work was supported by the U.S. National Science Foundation, and the U.S. Department of Energy. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.
2. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick, *The landscape of parallel computing research: A view from berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
3. M. Baboulin, J. Demmel, J. Dongarra, S. Tomov, and V. Volkov, *Enhancing the performance of dense linear algebra solvers on GPUs [in the MAGMA project]*, Poster at Supercomputing 08, November 18, 2008, http://www.cs.utk.edu/~tomov/SC08-poster.pdf.
4. S. Barrachina, M. Castillo, F. Igual, R. Mayo, E. Quintana-Orti, and G. Quintana-Orti, *Exploiting the capabilities of modern GPUs for dense matrix computations*, Technical Report ICC 01-11-2008, Universidad Jaime I (Spain), 2008.
5. Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel, *Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology*, International Conference on Supercomputing, 1997, pp. 340–347.
6. George Bosilca, Zizhong Chen, Jack Dongarra, Victor Eijkhout, Graham Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, , and Satish S. Vadiyar, *Self adapting numerical software (SANS) effort*, IBM Journal of Reseach and Development **50** (2006), no. 2/3, 223–238.
7. Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick, *Self adapting linear algebra algorithms and software*, Proceedings of the IEEE **93** (2005), no. 2, special issue on "Program Generation, Optimization, and Adaptation".
8. Jack Dongarra, Shirley Moore, Gregory Peterson, Stanimire Tomov, Jeff Allred, Vincent Natoli, and David Richie, *Exploring new architectures in accelerating CFD for Air Force applications*, Proceedings of HPCMP Users Group Conference 2008 (July 14-17, 2008), http://www.cs.utk.edu/~tomov/ugc2008_final.pdf.
9. Matteo Frigo and Steven G. Johnson, *FFTW: An Adaptive Software Architecture for the FFT*, Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.
10. John A. Gunnels, Robert A. Van De Geijn, and Greg M. Henry, *Flame: Formal linear algebra methods environment*, ACM Transactions on Mathematical Software **27** (2001), 422–455.
11. Stanimire Tomov, Jack Dongarra, and Marc Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Technical Report UT-CS-08-632, University of Tennessee, 2008, LAPACK Working Note 210.
12. V. Volkov and J. Demmel, *Benchmarking GPUs to tune dense linear algebra*, Supercomputing 08, IEEE, 2008, to appear.
13. Vasily Volkov and James Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Tech. Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.
14. R. Clinton Whaley, Antoine Petitet, and Jack Dongarra, *Automated Empirical Optimizations of Software and the ATLAS Project.*, Parallel Computing **27** (2001), no. 1-2, 3–35.