# Contents

# Chapter 3

# Empirical Performance Tuning of Dense Linear Algebra Software

**Jack Dongarra**

*University of Tennessee/Oak Ridge National Laboratory*

**Shirley Moore**

*University of Tennessee*

Dense linear algebra (DLA) forms the core of many scientific computing applications. Consequently, there is continuous interest and demand for the development of efficient algorithms and implementations on new architectures. One response to this demand has been the development of the ATLAS (Automatic Tuning of Linear Algebra Software) system to automatically produce implementations of the BLAS (Basic Linear Algebra Subroutines) routines that underlie all of dense linear algebra. ATLAS generates efficient code by running a series of timing experiments using standard techniques for improving performance (loop unrolling, blocking, etc.) to determine optimal parameters and code structures. While ATLAS has been highly successful in tuning DLA for cache-based architectures, we are developing new auto-tuning techniques for multicore and heterogeneous architectures that exploit higher levels of parallelism and asynchronous scheduling. This chapter describes the ATLAS techniques as well as recent research on empirical tuning of dense linear algebra routines for multicore and GPU architectures.

## 3.1 Background and Motivation

This section begins with a a discussion of how scientific computing relies on the efficient solution of numerical linear algebra problems. It discusses the critical performance issues involved, emphasizing performance on multicore and heterogeneous architectures. It then motivates the remainder of the chapter by introducing the empirical approach to DLA performance tuning.

### 3.1.1 Importance of Dense Linear Algebra Software

The standard problems of numerical linear algebra include linear systems of equations, least squares problems, eigenvalue problems, and singular value problems [9]. Linear algebra software routines for solving these problems are widely used in the computational sciences in general, and in scientific modeling in particular. In many of these applications, the performance of the linear algebra operations are the main constraint preventing the scientist from modeling more complex problems, which would then more closely match reality. This then dictates an ongoing need for highly efficient routines; as more compute power becomes available the scientist typically increases the complexity/accuracy of the model until the limits of the computational power are reached. Therefore, since many applications have no practical limit of "enough" accuracy, it is important that each generation of increasingly powerful computers have optimized linear algebra routines available.

### 3.1.2 Dense Linear Algebra Performance Issues

Linear algebra is rich in operations that are highly optimizable, in the sense that a highly tuned code may run multiple orders of magnitude faster than a naively coded routine. However, these optimizations are platform specific, such that an optimization for a given computer architecture will actually cause a slow-down on another architecture. The traditional method of handling this problem has been to produce hand-optimized routines for a given machine. This is a painstaking process, typically requiring many man-months from personnel who are highly trained in both linear algebra and computational optimization. The incredible pace of hardware evolution makes this approach untenable in the long run, particularly so when considering that there are many software layers (eg., operating systems, compilers, etc) that also effect these kinds of architectures.

### 3.1.3 Idea of Empirical Tuning

Automatic performance tuning, or auto-tuning, has been used extensively to automatically generate near-optimal numerical libraries for modern CPUs. For example, ATLAS [14, 10] and PHiPAC [8] are used to generate optimized libraries for FFT, which is one of the most important algorithms for digital signal processing. There are two general approaches to auto-tuning, namely model-driven optimization and empirical optimization. The idea of model-driven optimization comes from the compiler community. The compiler research community has developed various optimization techniques that can effectively tranform code written in high-level languages such as C and Fortran to run efficiently on modern 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 at compile time by analytical methods. While model-driven optimization is generally effective in making programs runs faster, it may not give optimal performance for linear algebra and signal processing libraries. The reason is that analytical models used by ocmpilers are simplified abstractions of the underlying processor architecture, 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 less effective for the optimization of linear algebra and signal processing kernels if the approach is solely used. In contrast to model-driven optimization, empirical optimization techniques generate a large number of parameterized code variants for a given algorithm and run these variants on a given platform to find the one that gives the best performance. The effectiveness of empirical optimization depends on the parameters chosen to optimize and on 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. However, this cost may be justified for frequently used code, where the cost is amortized over the total useful lifetime of the generated code.

## 3.2 ATLAS

This section describes the goals and approach of ATLAS. ATLAS provides highly optimized linear algebra kernels for arbitrary cache-based architectures. The initial goal of ATLAS was to provide a portably efficient implemenatation of the BLAS (Basic Linear Algebra Subroutines). ATLAS was originally released in 1997. The most recent stable release is version 3.8.3, released in February 2009. ATLAS now provides at least some level of support for all of the BLAS and has been extended to some higher level routines from LAPACK.

As explained in [14], ATLAS uses three different methods of software adaptation, described as follows:

- *parameterized adaptation* This method involves parameterizing charactertistics that vary from machine to machine. The most important such parameter in linear algebra is probably the blocking factor used in blocked algorithms which affects data cache utilization. Not all important architectural variables can be handled by this method, however, since some of them, such as choice of combined or separate multipley and add instructions, length of floating point and fetch pipelines, etc.) can be varied only by changing the underlying source code. For these variables, the two source code adaptations described below may be used.

- *multiple implemementation* This source code adaptation method involves searching a collection of various hand-tuned implementations until the best is found. ATLAS adds a search and timing layer to accomplish what would otherwise be done by hand. An advantage of thrs method is that multiple authors can contribute implementations without having to understand the entire package.

- *source generation* This source code adaptation method uses a program called a source generator which takes the various source code adapatations to be made as input and produces a source code routine with the specified characteristics. This method is flexible but complicated.

The BLAS are building block routines for performing basic vector and matrix operations. The BLAS are divided into three levels: Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations. The performance gains from optimized implementations depends on the level of the BLAS. commonly occurring problems in numerical linear algebra. ATLAS natively provides only a handful of LAPACK routines, but the ATLAS-provided routines can be automatically added to the standard LAPACK library from netlib [3] to produce a complete LAPACK library. In the following subsections, we describe ATLAS's BLAS and LAPACK support.

### 3.2.1 Level 3 BLAS Support

The Level 3 BLAS perform matrix matrix operations. They have $O(N^3)$ operations but need only $O(N^2)$ data. These routines can be effectively reordered and blocked for cache reuse and thus made to run fairly close to theoretical peak on most architectures. All the Level 3 BLAS routines can be efficiently implemented given an efficient matrix-matrix multiply (hereafter shortened to the BLAS matrix multiplication routine name, GEMM). Hence, the main performance kernel is GEMM. GEMM itself is further reduced to an even smaller kernel, called *gemmK*, before code generation takes place. *gemmK* is blocked to constant dimensions, usually for Level 1 Cache, and then heavily optimized for both the floating point unit and memory hierarchy using parameterization combined with multiple implementation and code generation. ATLAS first empirically searches the optimization space of the *gemmK* code and then optimizes the same code using multiple implementation. The *gemmK* that is finally used is the best performing kernel from these two searches.

### 3.2.2 Level 2 BLAS Support

The Level 2 BLAS perform matrix-vector operations such as matrix-vector multiply, rank 1 update and triangular forward/backward solve. ATLAS requires only one kernel to support all Level 3 BLAS, but this is not true of the Level 2 BLAS. The Level 2 BLAS have $O(N^2)$ operations and $O(N^2)$ data. Two classes of kernels are needed, a tuned general matrix-vector multiply (GEMV) and a tuned rank-1 update (GER) to support a GEMV- and GER-based Level 2 BLAS. However, since the matrix cannot be copied without incurring as much cost as an operation, a different GEMV kernel is required for each transpose setting. ATLAS tunes these kernels using only parameterization for cache blocking and multiple implementation. The Level 3 BLAS performance is determined by the peak of the machine. The Level 2 and Level 1 BLAS performance, however, is usually determined by the speed of the data bus, and thus the amount gained by optimization is less.

### 3.2.3 Level 1 BLAS Support

The Level 1 BLAS do vector-vector operations such as dot product. These routines have $O(N)$ operations on $O(N)$ data. ATLAS tunes the Level 1 BLAS using only multiple implementation along with some simple parameterization. Essentially, the only optimizations to be done at this level involve floating point unit usage and some loop optimizations. Since these routines are very simple, a compiler can usually do an excellent job with these optimizations; hence, performance gains from auto-tuning are typically found only when a compiler is poorly adapted to a given platform.

### 3.2.4 LAPACK Support

ATLAS currently provides ten basic routines from LAPACK, each of which is available in all four data types. These routines all use or provide for the LU and Cholesky factorizations and are implemented using recursion rather than static blocking.

### 3.2.5 Blocking for Higher Levels of Cache

Note that this chapter defines the Level 1 (L1) cache as the “lowest” level of cache: the one closest to the processor. Subsequent levels are “higher”: further from the processor and thus usually larger and slower. Typically, L1 caches are relatively small, employ least recently used replacement policies, have separate data and instruction caches, and are often non-associative and write-through. Higher levels of cache are more often non-write-through, with varying degrees of associativity, differing replacement polices, and combined instruction and data cache.

ATLAS detects the actual size of the L1 data cache. However, due to the wide variance in high level cache behaviors, in particular the difficulty of determining how much of such caches are usable after line conflicts and data/instruction partitioning is done, ATLAS does not detect and use an explicit Level 2 cache size as such. Rather, ATLAS employs a empirically determined value called `CacheEdge`, which represents the amount of the cache that is usable by ATLAS for its particular kind of blocking.

### 3.2.6 Use of Assembly Code

Hand-tuned implementations used by ATLAS that allow for extreme architectural specialization are sometimes written in assembly code. Sometimes the compiler is not able to generate efficient backend code. ATLAS also uses assembly code to achieve persistent performance in spite of compiler changes.

### 3.2.7 Use of Architectural Defaults

The architectural defaults provided with ATLAS are the result of several guided installations – that is, the search has been run multiple times with intervention by hand if necessary. ATLAS’s empirical search is meant to be used only when architectural defaults are unavailable or have become non-optimal due to compiler changes. As pointed out in the UMD Autotuning chapter, empirical search run on real machines with unrelated load can result in high variance in the timing results. Thus, use of architectural defaults, which serve as a type of performance database of past results, can yield the best results.

### 3.2.8 Search Algorithm

ATLAS uses a ”relaxed 1-D line search”, where the ”relaxed” means that interacting transforms are usually handled by 2- or 3-D searches [1]. This basic search technique is adequate, given that the ATLAS developers understand good start values and the interactions between optimizations.

## 3.3 Auto-tuning for Multicore

To deliver on the promise of multicore petascale systems, library designers must find methods and algorithms that can effectively exploit levels of parallelism that are orders of magnitude greater than most of today's systems offer. To meet this challenge, the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project is developing dense linear algebra routines for multicore architectures [4]. In PLASMA, parallelism is no longer hidden inside Basic Linear Algebra Subprograms (BLAS) [2] but is brought to the fore to yield much better performance. PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms can be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks and edges represent dependencies among them. Asynchronous, out of order scheduling of operations is used as the basis for a scalable and highly efficient software framework for computational linear algebra applications. PLASMA is currently statically scheduled with a tradeoff between load balancing and data reuse. PLASMA performance depends strongly on tunable execution parameters, the outer and inner blocking sizes, that trade off utilization of different system resources, as illustrated in Figure 3.1.

### 3.3.1 Tuning outer and inner block sizes

The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra-flops due to redundant calculations. Only the QR and LU factorizations use inner blocking. If no inner blocking occurs, the resulting extra-flops overhead may represent 25% and 50% for the QR and LU factorization, respectively [5]. Tuning PLASMA consists of finding the (NB,IB) pairs that maximize the performance depending on the matrix size and on the number of cores. An *exhaustive search* is cumbersome since the search space is huge. For instance, in the QR and LU cases, there are 1352 possible combinations for (NB,IB) even if we constrain NB to be an even integer between 40 and 500 and if we constrain IB to divide NB. All these combinations cannot be explored for large matrices ($N >> 1000$) on effective factorizations in a reasonable time. Knowing that this process should be repeated for each number of cores and each matrix size motivates us to consider a *pruned search*. The idea is that tuning the serial level-3 kernel (dgemm-seq, dssrfb-seq and dssssm-seq) is not time-consuming since peak performance is reached on relatively small input matrices ($NB < 500$) that can be processed fast. Therefore, we first tune those serial kernels. As illustrated in Figure 3.2, not all the (NB,IB) pairs result in a high performance. For instance, the (480,6) pair leads to a performance of 6.0 Gflop/s whereas the (480,96) pair achieves 12.2 Gflop/s, for the dssrfb-seq kernel on Power6

(a) DPOTRF - Intel64 - 16 cores (b) DGEQRF - Intel64 - 16 cores (c) DGETRF - Intel64 - 16 cores

(d) DPOTRF - Power6 - 16 cores (e) DGEQRF - Power6 - 16 cores (f) DGETRF - Power6 - 16 cores

(g) DPOTRF - Power6 - 32 cores (h) DGEQRF - Power6 - 32 cores (i) DGETRF - Power6 - 32 cores

FIGURE 3.1: Effect of (NB,IB) on PLASMA performance (Gflop/s).

(a) dgemm-seq - Intel64 (b) dssrfb-seq - Intel64 (c) dssssm-seq - Intel64

(d) dgemm-seq - Power6 (e) dssrfb-seq - Power6 (f) dssssm-seq - Power6

**FIGURE 3.2**: Effect of (NB,IB) on the performance of the serial PLASMA computational intensive level-3 BLAS kernels (Gflop/s).

(Figure 3.2(e)). We select a limited number of (NB,IB) pairs (*pruning* step) that achieve a local maximum performance on the range of NB. We have selected five or six pairs on the Intel64 machine for each factorization and eight on the Power6 machine (Figure 3.2). We then benchmark the performance of PLASMA factorizations only with this limited number of combinations (as seen in Figure 3.1). Finally, the best performance obtained is selected.

The dssssm-seq efficiency depends on the amount of pivoting performed. The average amount of pivoting effectively performed during a factorization is matrix-dependent. Because the test matrices used for our LU benchmark are randomly generated with a uniform distribution, the amount of pivoting is likely to be important. Therefore, we have selected the (NB,IB) pairs from dssssm-seq executions with full pivoting (figures 3.2(c) and 3.2(f)). The dssssm-seq performance drop due to pivoting can reach more than 2 Gflop/s on Power6 (Figure 3.2(f)).

### 3.3.2 Validation of pruned search

We have validated our *pruned search* methodology for the three one-sided factorizations on Intel64 16 cores. To do so, we have measured the relative performance overhead (percentage) of the pruned search (PS) over the exhaustive search (ES), that is: $100 \times (\frac{ES}{PS} - 1)$. Table 3.1 shows that the pruned

**TABLE 3.1**: Overhead (in %) of Pruned search (Gflop/s) over Exhaustive search (Gflop/s) on Intel64 16 cores

| | DPOTRF | | | DGEQRF | | | DGETRF | | |
|---|---|---|---|---|---|---|---|---|---|
| Matrix Size | Pruned Search | Exhaustive Search | Over-head | Pruned Search | Exhaustive Search | Over-head | Pruned Search | Exhaustive Search | Over-head |
| 1000 | 53.44 | 52.93 | -0.95 | 46.35 | 46.91 | 1.20 | 36.85 | 36.54 | -0.84 |
| 2000 | 79.71 | 81.08 | 1.72 | 74.45 | 74.95 | 0.67 | 61.57 | 62.17 | 0.97 |
| 4000 | 101.34 | 101.09 | -0.25 | 93.72 | 93.82 | 0.11 | 81.17 | 80.91 | -0.32 |
| 6000 | 108.78 | 109.21 | 0.39 | 100.42 | 100.79 | 0.37 | 86.95 | 88.23 | 1.47 |
| 8000 | 112.62 | 112.58 | -0.03 | 102.81 | 102.95 | 0.14 | 89.43 | 89.47 | 0.04 |

search performance overhead is bounded by 2%. Because the performance may slightly vary from one run to another on cache-based architectures [6], we could furthermore observe in some cases higher performance (up to 0.95%) with pruned search (negative overheads in Table 3.1). However, the (NB,IB) pair that leads to the highest performance obtained with one method consistently matches the pair leading to the highest performance obtained with the other method.

We expect that the results will generalize to other linear algebra problems and even to any algorithm that can be expressed by a DAG of fine-grain tasks. Compiler techniques allow for the DAG of tasks to be generated from the polyhedral model applied to code that is free of runtime dependeces such as Cholesky or LU factorization without pivoting [7]. In comparison, by working at a much higher abstraction layer (a whole matrix tile as opposed to individual matrix elements) and with semantic knowledge of functions called from within the loop nests, PLASMA is able to produce highly-tuned kernels.

## 3.4 Auto-tuning for GPUs

As mentioned above, the development of high performance dense linear algebra (DLA) depends critically on highly optimized BLAS, and especially on the matrix multiplication routine (GEMM). This statement is especially true for Graphics Processing Units (GPUs), as evidenced by recently published results on DLA for GPUs that rely on highly optimized GEMM [12, 13]. However, the current best GEMM performance on GPUs, e.g., up to 375 GFlop/s in single precision and up to 75 GFlops/s in double precision arithmetic on NVIDIA's GTX 280, is difficult to achieve. The software development requires extensive GPU knowledge and even backward engineering to understand undocumented aspects of the architecture. This section describes preliminary

FIGURE 3.3: The algorithmic view of the code template for GEMM.

work on some GPU GEMM auto-tuning tecnhiques that allow keeping up with changing hardware by rapidly reusing existing ideas. Preliminary results show auto-tuning to be a practical solution that, in addition to enabling easy portability, can achieve substantial speedups on current GPUs (e.g., up to 27% in certain cases for both single and double precision GEMM on the GTX 280).

### 3.4.1 GEMM auto-tuner

This section presents preliminary work on the design of a GEMM "auto-tuner" for NVIDIA CUDA-enabled GPUs. Here auto-tuner means a system that automatically generates and searches a space of algorithms. More details may be found in [11].

In [13], 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 [13] takes advantage of the vector capability of NVIDIA CUDA-enabled GPUs. The 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 3.3 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 $16 \times 4$ threads, then each thread will compute exactly one column of BC. If the thread block has $16 \times 8$ 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 [13], 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 [13], 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.

### 3.4.2 Performance Results

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

First, we evaluate the performance of the GEMM autotuner in both single and double precision. Figure 3.4, Left compares the performance of the

**FIGURE 3.4**: Performance comparison of CUBLAS 2.0 *vs* auto-tuned SGEMM (left) and DGEMM (right) on square matrices.

GEMM autotuner in single precision with the CUBLAS 2.0 SGEMM for multiplying square matrices. We note that both CUBLAS 2.0 SGEMM and our auto-tuned SGEMM are based on V.Volkov's SGEMM [13]. 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 3.4, 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 3.5, 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 $6144 \times 6144$. The graphs show that our auto-tuned code significantly outperforms (up to 27%) the DGEMM from CUBLAS 2.0.

Using the new DGEMM for example in the block LU (of block size BS = 64) with partial pivoting [13] improved the performance from 53 to 65 GFlop/s on a matrix of size $6144 \times 6144$.

**FIGURE 3.5**: 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 $6144 \times 6144$. 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).

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.

## 3.5 Conclusions

Auto-tuning is crucial for the performance and maintenance of modern numerical libraries, especially for algorithms designed for multicore and hybrid architectures. It is an elegant and very practical solution for easy maintenance and performance portability. While an empirically based exhaustive search can find the best performance kernels for a specific hardware configuration, applying performance models can often effectively prune the search space.

# *Bibliography*

[1] ATLAS Frequently Asked Questions. `http://math-atlas.sourceforge.net/faq.html`.

[2] BLAS: Basic linear algebra subprograms. `http://www.netlib.org/blas/`.

[3] Netlib repository. `http://www.netlib.org/`.

[4] Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project. `http://icl.cs.utk.edu/plasma/`.

[5] Buttari A., Langou J., Kurzak J., and Dongarra J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[6] Sloss A., Symes D., and Wright C. *ARM System Developer's Guide: Designing and Optimizing System Software.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[7] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Bonkhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, North Carolina, February 2009.

[8] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[9] James W. Demmel. *Applied numerical linear algebra.* Society for Industrial and Applied Mathematics, 1997.

[10] 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(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[11] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *9th International Conference on Computation Science (ICCS'09)*, Baton Rouge, LA, May 2009.

[12] 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.

[13] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008. to appear.

[14] R. Clint Whaley. Atlas version 3.8: Status and overview. In *International Workshop on Automatic Performance Tuning (iWAPT07)*, Tokyo, Japan, September 2007.