Analytical Models for the BLIS Framework
FLAME Working Note #74

Tze Meng Low[1], Francisco D. Igual[2], Tyler M. Smith[1], and Enrique S. Quintana-Orti[3]

[1]The University of Texas at Austin
[2]Universidad Complutense de Madrid
[3]Universidad Jaume I

June 7, 2014

Abstract

We show how the BLAS-like Library Instantiation Software (BLIS) framework, which provides a more detailed layering of the GotoBLAS (now maintained as OpenBLAS) implementation, allows one to analytically determine optimal tuning parameters for high-end instantiations of the matrix-matrix multiplication. This is of both practical and scientific importance, as it greatly reduces the development effort required for the implementation of the level-3 BLAS while also advancing our understanding of how hierarchically layered memories interact with high performance software. This allows the community to move on from valuable engineering solutions (empirically autotuning) to scientific understanding (analytical insight).

1 Introduction

The field of dense linear algebra (DLA) was among the first to realize, understand, and promote that standardizing (*de facto*) an interface to fundamental primitives supports portable high performance. For almost four decades, those primitives have been the *Basic Linear Algebra Subprograms* (BLAS) [19, 9, 8]. The expectation was and still remains that some expert will provide to the community a high performance implementation of the BLAS every time a new architecture reaches the market. For this purpose, many hardware vendors currently enroll a numerical libraries group and distribute well-maintained libraries (e.g., Intel's MKL [15], AMD's ACML [1], and IBM's ESSL [14] libraries), while over the years we have enjoyed a number of alternative Open Source solutions (e.g., GotoBLAS [11, 10], OpenBLAS [20], ATLAS [27]). Nevertheless, these solutions all require considerable effort in time and labor for each supported target architecture.

In order to reduce the exertion of developing high performance implementations, ATLAS and its predecessor PHiPAC [6] introduced autotuning to the field of high performance computing. The fundamental rationale behind these two libraries is that optimizing is too difficult and time-consuming for a human expert and that autogeneration in combination with autotuning is the answer.

The work in [28] shows that the ATLAS approach to optimizing the general matrix-matrix product (GEMM) can be, by and large, analytically modelled. Thus, it reveals that autotuning is unnecessary for the operation that has been tauted by the autotuning community as *the* example of the success of autotuning. The problem with that work ([28]) is that the ATLAS approach to optimizing GEMM had been previously shown to be suboptimal, first in theory [12] and then in practice [11]. Furthermore, ATLAS leverages an inner kernel optimized by a human expert, which still involves a substantial manual encoding. Precisely, the presence of this black-box kernel makes analytical modeling problematic for ATLAS, since some of the tuning parameters are hidden inside.

GotoBLAS (now supported as OpenBLAS) also builds on a substantial inner kernel that is implemented by a human, turning the analysis equally difficult without a complete understanding of the complex details embedded into the inner kernel. (Until recently, no paper that detailed the intricacies of the GotoBLAS inner kernel existed.) Interestingly, the human expert behind GotoBLAS did not use autotuning, hinting that perhaps this was not necessary.

BLIS (*BLAS-like Library Instantiation Software*) [25] is a new framework for instantiating the BLAS. A key benefit of BLIS is that it is a productivity multiplier, as the framework allows developers to rapidly unfold new high-performance implementations of BLAS and BLAS-like operations on current and future architectures [24]. From the implementation point of view, BLIS modularizes the approach underlying GotoBLAS2 (now adopted also by many other implementations) to isolate a *micro-kernel* that performs GEMM upon which all the level-3 BLAS functionality can be built. Thus, for a given architecture, the programmer only needs to develop an efficient micro-kernel for BLIS, and adjust a few key parameter values[1] that optimize the loops around the micro-kernel, to automatically instantiate all the level-3 BLAS. Importantly, BLIS features a layering that naturally exposes the parameters that need to be tuned.

While BLIS simplifies the implementation of BLAS(-like) operations, a critical step to optimize BLIS for a target architecture is for the developer to identify the specific parameter values for both the micro-kernel and the loops around it. Conventional wisdom would dictate that empirical search must be applied. In this paper, we adopt an alternative model-driven approach to analytically identify the optimal parameter values for BLIS. In particular, we apply an analysis of the algorithm and architecture, similar to that performed in [28], to determine the data usage pattern of the algorithm ingrained within the BLIS framework, and to build an analytical model based on hardware features in modern processor architectures.

The specific contributions of this work include:

- **An analysis of the best known algorithm.** We address the algorithm underlying BLIS (and therefore GotoBLAS and OpenBLAS) instead of the algorithm in ATLAS. This is relevant because it has been shown that, on most architectures, a hand-coded BLIS/GotoBLAS/OpenBLAS implementation (almost) always yields higher performance than an implementation automatically generated via ATLAS [24], even if ATLAS starts with a hand-coded inner kernel.

- **A more modern model.** We accommodate a processor model that is more representative of modern architectures than that adopted in [28]. Concretely, our model includes hardware features such as a vector register file and a memory hierarchy with multiple levels of set-associative data caches. This considerably improves upon the analytical model in [28] which considered one level only of fully associative cache and ignored vector instructions.

- **A more comprehensive model.** Our analytical model is more comprehensive in that it includes the parameter values that also characterize the micro-kernel. These are values that a developer would otherwise have to determine empirically. This is important because the best implementations provided by ATLAS often involve loops around a hand-coded (black-box) kernel. Since the internals of the hand-coded kernel are not known, the model in [28] was not able to identify the globally optimal parameter values.

[1]The micro-kernel itself is characterized by three additional parameters, which the programmer has to consider when implementing it.

- **Competitive with expert-tuned implementations.** Unlike previous work comparing the performance attained against auto-generated implementations in C obtained by ATLAS [28, 16], which are typically not competitive with those that use hand-coded inner kernels (so-called "ATLAS unleashed" in [28]), this paper shows that analytically-obtained parameter values can yield performance that is competitive with manually-tuned implementations that achieve among best-in-class performance. Hence, this paper provides an answer to the question posed in [28] —i.e. *whether empirical search is really necessary in this context*—, by demonstrating that *analytical modeling suffices for high performance BLIS implementations* which then shows that careful layering combined with analytical modeling is competitive with GotoBLAS, OpenBLAS, and vendor BLAS (since other papers show BLIS to be competitive with all these implementations).

In this paper, we restrict ourselves to the case of a single-threaded implementation of GEMM in double precision. How to determine optimal parameter values for a multithreaded implementation is orthogonal to this paper, and is at least partially answered in [23].

2 Approaches to identify the optimal parameter values for GEMM

It has been argued that empirical search is the only way to obtain highly optimized implementations for DLA operations [7, 5, 27], and an increasing number of recent projects (Build-To-Order BLAS [4] and AuGEMM [26]) now adopt empirical search to identify optimal parameter values for DLA algorithms.

The problem with empirical-based approaches is that they unleash a walloping search space, due to the combination of a large number of possible values for a substantial set of parameters. Therefore, even with the help of heuristics, generating, executing, and timing all variants of the program, with their unique combinations of parameter values, requires a considerable amount of time (sometimes measured in days). In addition, empirical search has to be performed on the target machine architecture. This implies that a high performance implementation of BLAS for a new machine architecture cannot be developed until the programmer is granted access to the target machine, and then it still requires a significant amount of time. This is especially critical on state-of-the-art embedded architectures, where cross-compiling processes are necessary, greatly complicating or even invalidating in practice the application of empirical-based approaches.

A refinement of the empirical approach is iterative optimization [18, 17, 17]. As the name suggests, instead of a single run that explores the entire search space, multiple optimization passes are executed. There exist different techniques to perform iterative optimization, but they all share the following similarities: An initial sampling run is performed while a monitoring tool captures specific performance information such as the number of cache misses. Using the captured information and the application of some heuristics, the search space is trimmed/refined, and new program variants (with new sets of parameter values) are generated for further exploration in subsequent passes.

Although iterative optimization represents an improvement over empirical search, the information obtained, e.g. from the performance counters, may not be sufficient to limit the search space. Consider the graphs in Figure 1, which report the GFLOPS (i.e., billions of floating-point arithmetic operations, or flops, per second), and L1, L2 and TLB cache misses observed for the execution of a tuned implementation of `dgemm` from BLIS on a single core of an Intel Xeon E3-1220 processor (3.1 GHz), while varying two of the optimization parameters only (m_c and k_c, to be introduced later). The top-left plot illustrates the challenge for empirical search: generate and evaluate all different combinations of an exponentially growing search space. (BLIS, e.g., needs to optimize 5–6 different parameters, which is a reduced quantity when compared with other solutions including ATLAS.) In contrast to this, iterative optimization runs a coarse-grained grid of experiments that can help to identify contour lines which roughly delimit the area which

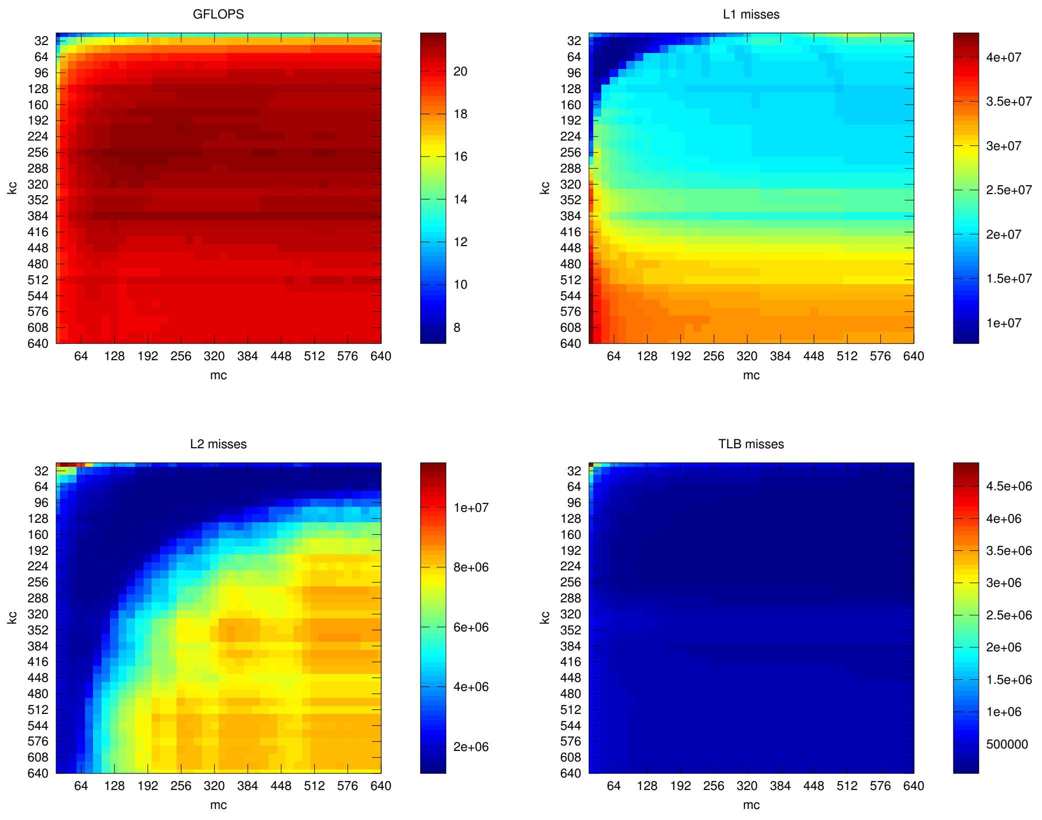


Figure 1: Performance metrics when empirically testing different parameter values. Clockwise from top-left: GFLOPS, and misses in the L1, L2 and TLB caches. A heuristic for reducing the search space is to find the parameter values that minimizes the cache misses and TLB misses. From the above data, this still leaves a relatively large space that needs to be searched.

simultaneously minimizes the amount of cache misses for all three caches (L1, L2, and TLB). However, even armed with that information, we can observe from the three cache miss plots that such region still comprises an enormous number of cases that nonetheless have to be individually evaluated to finally reveal the optimal combination for the architecture (m_c=96 and k_c=256 for this particular case).

A third approach to identify the optimal parameter values is to use analytical models as advocated by some in the compiler community. Concretely, simple analytical models have been previously developed in the compiler domain in order to determine optimizing parameter values for tile sizes and unrolling factors [3, 28, 16]. These models are based on the data access pattern and exploit a fair knowledge of the hardware features commonly found in modern processor architectures. Previous work has shown that parameter values derived analytically can deliver performance rates that are similar to those attained using values determined through empirical search [28]. More importantly, the time it takes to analytically identify the optimal parameter combination is often a fraction of that required by empirical search. Let us distinguish this from autotuning by calling it auto-*fine*-tuning. We adopt this in our quest for the parameter values that optimize the algorithm in BLIS, with the aim to make even auto-fine-tuning unnecessary for many architectures.

3 Expert Implementation of GEMM

The GEMM algorithm embedded within the BLIS framework is described in [25], and the same approach is instantiated in BLAS libraries optimized by DLA experts such as GotoBLAS [11, 10] and its successor, OpenBLAS [20]. Importantly, BLIS exposes three loops within the inner kernel used by the GotoBLAS and OpenBLAS, which then facilitates the analytical determination of the parameters. For completeness, we provide a brief overview of the algorithm next. In addition, we identify the data usage pattern and highlight the parameters that characterize the algorithm.

3.1 An expert's implementation of GEMM

Without loss of generality, we consider the special case of the matrix-matrix multiplication, $C := AB + C$, where the sizes of A, B, C are $m \times k$, $k \times n$, and $m \times n$, respectively. In the following elaboration, we will consider different partitionings of the m, n and k-dimensions of the problem. For simplicity, when considering a (vertical or horizontal) partitioning of one of the problem dimensions, say q, into panels/blocks of size (length or width) r, we will assume that q is an integer multiple of r.

BLIS implements GEMM as three nested loops around a *macro-kernel*[2], plus two packing routines. The macro-kernel is implemented in terms of two additional loops around a *micro-kernel*. The micro-kernel is a loop around a rank-1 (i.e., outer product) update, and it is typically implemented in assembly or with vector intrinsics. In BLIS, the remaining five loops are implemented in C.

Pseudo-code for the GEMM algorithm is given in Figure 2. The outermost loop (Loop 1, indexed in the figure by j_c) traverses the n-dimension of the problem, partitioning both C and B into column panels of width n_c. The second loop (indexed by p_c) processes the k-dimension, partitioning A into column panels of width k_c and the current panel of B into blocks of length k_c. The third loop (indexed by i_c) iterates over the m-dimension, yielding partitionings of the current column panels of C and A into blocks of length m_c. The following loops comprise the macro-kernel. Let us define $C_c = C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$. Loop 4 (indexed by j_r) traverses the n-dimension, partitioning C_c and the packed block B_c into *micro-panels* of width n_r. Loop 5 (indexed by i_r) then progresses along the m-dimension, partitioning the current micro-panel of C_c into *micro-tiles* of length m_r and the packed block A_c into micro-panels of the same length. The innermost loop (Loop 6, indexed by p_r), inside the micro-kernel, computes the product of a micro-panel of A_c and micro-panel of B_c as a sequence of k_c rank-1 updates, accumulating the result to a micro-tile of C_c, which can be described mathematically as

$$C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \mathrel{+}= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1) \cdot B_c(0 : k_c - 1, j_r : j_r + n_r - 1).$$

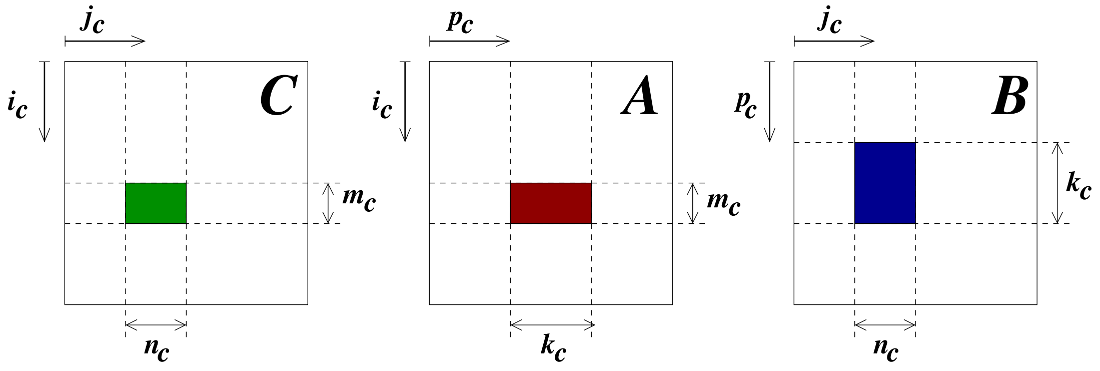
At this point it is worth emphasizing the difference between C_c and A_c, B_c. While the former is just a notation artifact, introduced to ease the presentation of the algorithm, the latter two correspond to actual buffers that are involved in data copies.

3.2 The importance of packing

It is generally known that accessing consecutive memory locations (also known as in-stride access) is usually faster than non-consecutive memory accesses. By packing elements of A in Loop 3 and B in Loop 2 in a special manner into A_c and B_c, respectively, we can ensure that the elements of these two matrices will be read with in-stride access inside the micro-kernel.

Concretely, each $m_c \times k_c$ block of A is packed into A_c. Elements are organized as row micro-panels of size $m_r \times k_c$, and within each micro-panel of A_c, the elements are stored in column-major order. Each

[2]The macro-kernel is also known as the inner kernel in GotoBLAS.



```
Loop 1  for $j_c = 0, \ldots, n-1$ in steps of $n_c$
Loop 2      for $p_c = 0, \ldots, k-1$ in steps of $k_c$
                $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$        // Pack into $B_c$
Loop 3          for $i_c = 0, \ldots, m-1$ in steps of $m_c$
                    $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$    // Pack into $A_c$
        ------------------------------------------------------------------------------------
Loop 4              for $j_r = 0, \ldots, n_c - 1$ in steps of $n_r$                  // Macro-kernel
Loop 5                  for $i_r = 0, \ldots, m_c - 1$ in steps of $m_r$
        ----------------------------------------------------------------
Loop 6                      for $p_r = 0, \ldots, k_c - 1$ in steps of 1   // Micro-kernel
                                $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
                                    $\mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r)$
                                     $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$
                            endfor
        ----------------------------------------------------------------
                        endfor
                    endfor
        ------------------------------------------------------------------------------------
                endfor
            endfor
        endfor
```

Figure 2: High performance implementation of GEMM by an expert.

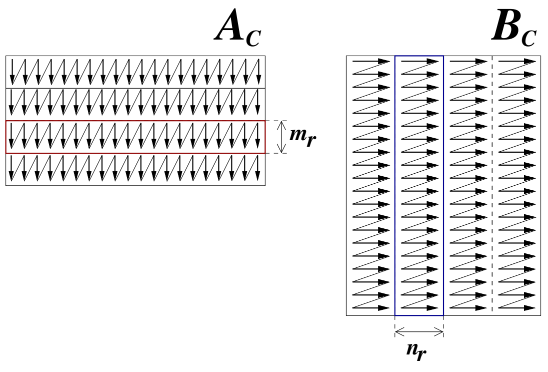


Figure 3: Packing in the BLIS and GotoBLAS implementations of GEMM.

Loop	Reused data	Size of reused data	Reuse factor
6	Micro-tile, C_r	$m_r \times n_r$	k_c
5	Micro-panel, B_r	$k_c \times n_r$	m_c/m_r
4	Packed block A_c	$m_c \times k_c$	n_c/n_r
3	Packed block B_c	$k_c \times n_c$	m/m_c
2	Matrix C	$m \times n$	k/k_c
1	Matrix A	$m \times k$	m/m_c

Table 1: Analysis of data reuse performed in the different loops of the BLIS implementation of GEMM.

$k_c \times n_c$ block of B is packed into B_c. In this case, elements are packed into column micro-panels of size $k_c \times n_r$, and each column micro-panel is stored into row-major order. The reorganization of the entries of A and B into blocks of A_c and B_c with the packing layout illustrated in Figure 3 ensures that these elements are accessed with unit stride when used to update a micro-tile of C. Packing A_c and B_c also has the additional benefit of aligning data from A_c and B_c to cache lines boundaries and page boundaries. This enables to use instructions for accessing aligned data, which are typically faster than their non-aligned counterparts.

A third benefit of packing is that data from A_c and B_c are preloaded into certain cache levels of the memory hierarchy. This reduces the time required to access the elements of A_c and B_c when using them to update a micro-tile of C. Since A_c is packed in Loop 3, after B_c has been packed in Loop 2, the elements of A_c will likely be higher up in the memory hierarchy (i.e. closer to the registers) than those of B_c. By carefully picking the sizes for A_c and B_c, the exact location of A_c and B_c within the memory hierarchy can be determined.

3.3 Data usage pattern

Consider Loop 6. This innermost loop updates a micro-tile of C, say C_r, via a series of k_c rank-1 updates involving a micro-panel A_r, from the packed matrix A_c, and a micro-panel B_r, from the packed matrix B_c. At each iteration, m_r elements from A_r and n_r elements from B_r are multiplied to compute $m_r n_r$ intermediate results that will be accumulated into the micro-tile of C. Between two iterations of Loop 6, different elements from A_r and B_r are used, but the results from the rank-1 updates are accumulated into the same micro-tile. Hence, the micro-tile C_r is the data reused between iterations (*Reused data*) in Loop 6. In addition, because k_c rank-1 updates are performed each time Loop 6 is executed, C_r is said to have a reuse factor of k_c times. Since the columns of A_r and the rows of B_r are used exactly once each time through Loop 6, there is no reuse of these elements.

Identifying pieces of data that are reused is important because it tells us which data portions should be kept in cache in order to allow fast access and reduce cache trashing. An analysis similar to that with Loop 6, performed on the remaining loops to identify the data that is reused in each one of them, the size of the reused data, and the number of times the data is being reused, yields the results summarized in Table 1.

Notice that the size of the reused data becomes smaller as the loop depth increases. This nicely matches the memory hierarchy, where faster memory layers (caches) feature lower capacity than slower caches. As such, GEMM maps smaller reused data to faster layers in the memory hierarchy, as depicted in Figure 4.

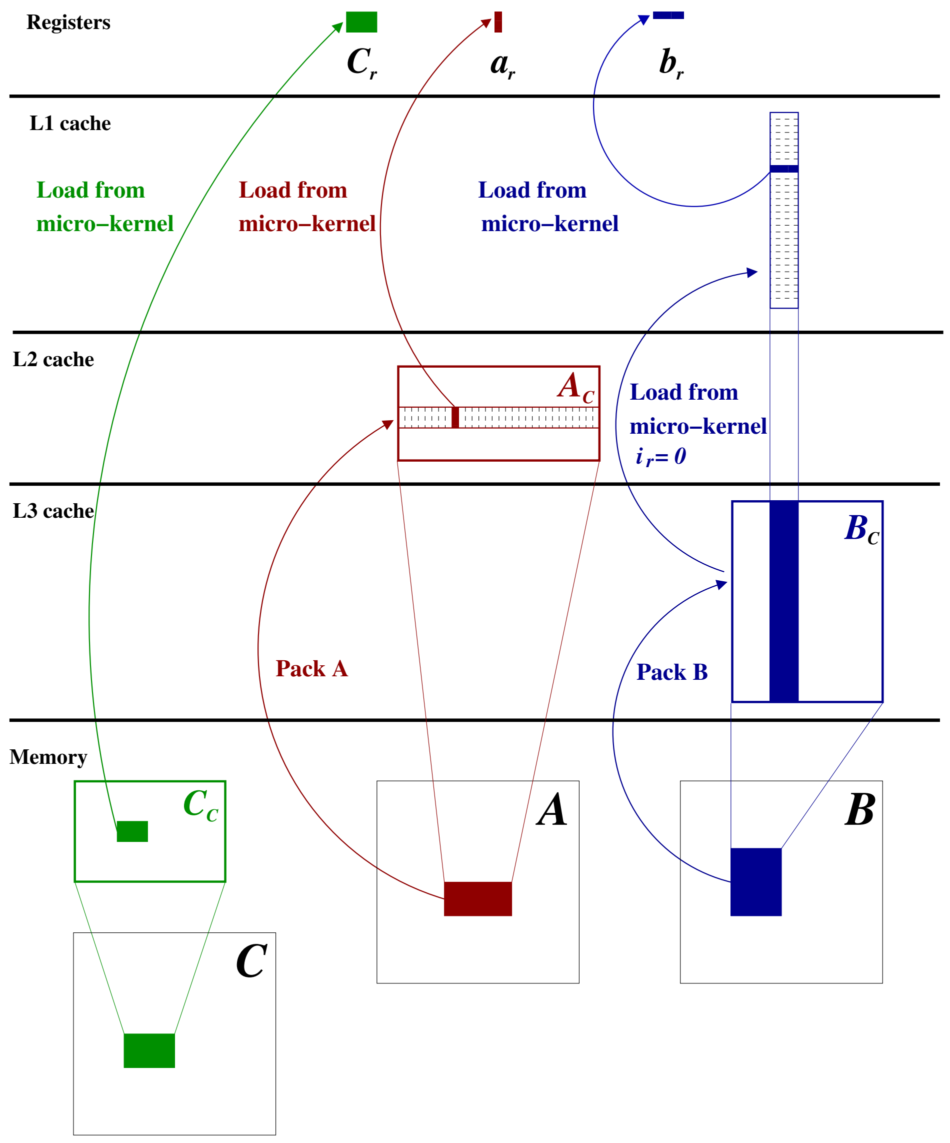


Figure 4: Packing in the BLIS and GotoBLAS implementations of GEMM.

3.4 Characteristic parameters that drive performance

As described in the previous subsection and captured in Table 1, the GEMM algorithm that underlies BLIS is characterized by the following five parameters:

$$m_c, k_c, n_c, m_r \text{ and } n_r,$$

which correspond to the block/panel/tile size for each of the loops around the micro-kernel. In addition, this set of parameters also determines the dimension of the reused data and the reuse factor of each of the loops.

Loop 6 is defined by three of the five parameters (m_r, n_r and k_c), while Loops 3, 4 and 5 are characterized by four of the five parameters. In addition, three of the four parameters that characterize Loops 3, 4 and 5 carry beyond to the next inner loop as well. Now, when the parameters that characterize Loop 5 have been identified, the only unknown parameter for Loop 4 is n_c. This observation suggests that the parameters should be identified from the innermost loop outwards. In the next section, we leverage this observation to optimize this collection of parameters analytically.

4 Mapping Algorithm to Architecture

An effective mapping of the micro-kernel and the loops around it to the architecture is fundamental to attain high performance with the BLIS approach. This means that we need to identify values for the characteristic parameters that are tuned for the target machine architecture. As discussed in the previous section, we start by mapping the micro-kernel to the architecture and work our way outwards in order to identify the optimal values for the different characteristic parameters.

4.1 Architecture model

To develop an optimization model for mapping the GEMM algorithm onto an architecture, we first need to define a model for our target architecture. We make the following assumptions regarding our hypothetical processor:

- **Load/store architecture and vector registers**. Data have to be loaded into the processor registers before computation can be performed with them. A total of N_{REG} vector registers exist, where each can hold N_{VEC} elements (floating-point numbers) of size S_{DATA}. (In a machine with no vector registers, $N_{\text{VEC}} = 1$.) In addition, we assume that memory instructions can be issued in parallel with floating-point arithmetic instructions.

- **Vector instructions**. The processor floating-point arithmetic units have a throughput of N_{VFMA} vector (or SIMD) *fused multiply-add instructions* (VFMA) per clock cycle (i.e., $2N_{\text{VEC}}N_{\text{VFMA}}$ flops per cycle). A single VFMA instruction combines N_{VEC} regular (non-vector) instructions and produces N_{VEC} scalar results. Furthermore, each VFMA has a latency of L_{VFMA} cycles, which is the minimum number of cycles between the issuance of two dependent consecutive VFMA instructions. On an architecture without VFMA instructions, L_{VFMA} is computed by adding the latencies for a multiply and an addition instruction.

- **Caches**. All data caches are set-associative, and each cache level Li is characterized by four parameters as follows:

  - $C_{\text{L}i}$: size of cache line,
  - $W_{\text{L}i}$: associativity degree,
  - $S_{\text{L}i}$: size, and
  - $N_{\text{L}i}$: number of sets,

  where $S_{\text{L}i} = N_{\text{L}i}C_{\text{L}i}W_{\text{L}i}$. A fully-associative cache can be modelled by setting $N_{\text{L}i} = 1$ and $W_{\text{L}i} = S_{\text{L}i}/C_{\text{L}i}$.

  The replacement policy for all caches is least-recently-used (LRU) [13]. For simplicity, we assume that the size of a cache line is the same for all cache levels.

4.2 Parameters for the body of the inner-most loop: m_r and n_r

Recall that the micro-kernel is characterized by three parameters: m_r, n_r and k_c, where the former two determine the size of the micro-tile of C that is reused across every iteration. In addition, these two parameters also define the number of elements of A_r and B_r that are involved in the rank-1 update at each iteration of Loop 6. In this subsection, we discuss how m_r and n_r can be identified analytically.

4.2.1 Strategy for finding m_r and n_r

The main strategy behind our approach to identify m_r, n_r is to choose them "large enough" so that no stalls due to the combination of dependencies and latencies are introduced in the floating-point pipelines during the repeated updates of the micro-tile C_r. In addition, the smallest values of m_r, n_r which satisfy this condition should be chosen because this implies that C_r occupies the minimum number of registers, releasing more registers for the entries of A_r, B_r. This in turn enables larger loop unrolling factors and more aggressive data preloading [13] to reduce loop overhead.

4.2.2 Latency of instructions

Consider the k_c successive updates of C_r occurring in Loop 6. In each of the k_c iterations, each element of C_r is updated by accumulating the appropriate intermediate result to it (obtained from multiplying corresponding elements of A_r and B_r). This update can be achieved with a VFMA instruction for each element of C_r in each iteration.

Now, recall that there is a latency of L_{VFMA} cycles between the issuance of a VFMA instruction and the issuance of another dependent VFMA instruction. This implies that L_{VFMA} cycles must lapse between two successive updates to the same element of C_r. During those L_{VFMA} cycles, a minimum of $N_{\text{VFMA}} L_{\text{VFMA}}$ VFMA instructions must be issued without introducing a stall in the floating-point pipelines. This means that, in order to avoid introducing stalls in the pipelines, at least $N_{\text{VFMA}} L_{\text{VFMA}} N_{\text{VEC}}$ output elements must be computed. Therefore, the size of the micro-tile C_r must satisfy:

$$m_r n_r \geq N_{\text{VEC}}\, L_{\text{VFMA}}\, N_{\text{VFMA}}. \tag{1}$$

4.2.3 Maximizing Register Usage

Ideally, m_r should be equal to n_r as this maximizes the ratio of computation to data movement during the update of C_r in Loop 6 ($2m_r n_r k_c$ flops vs $2m_r n_r + m_r k_c + n_r k_c$ memory operations). However, there are two reasons why this is not always possible in practice. First, m_r and n_r have to be integer values as they correspond to the dimensions of the micro-tile C_r. Second, it is desirable for either m_r or n_r to be integer multiples of N_{VEC}. As the number of registers is limited in any architecture, it is necessary to maximize their use, and choosing m_r (or n_r) to be an integer multiple of N_{VEC} will ensure that the registers are filled with elements from A_r, B_r and C_r such that data not used in a particular iteration is not kept in registers.

Therefore, in order to satisfy this criterion as well as (1), m_r and n_r are computed as follows:

$$m_r = \left\lceil \frac{\sqrt{N_{\text{VEC}}\, L_{\text{VFMA}}\, N_{\text{VFMA}}}}{N_{\text{VEC}}} \right\rceil N_{\text{VEC}} \tag{2}$$

and

$$n_r = \left\lceil \frac{N_{\text{VEC}}\, L_{\text{VFMA}}\, N_{\text{VFMA}}}{m_r} \right\rceil. \tag{3}$$

The astute reader will recognize that one could have computed n_r before computing m_r. Doing this is analogous to swapping the values of m_r and n_r, and also yields a pair of values that avoid the introduction of stalls in the pipeline. We choose the values of m_r and n_r that maximize the value of k_c, which will be discussed in the following section.

4.3 Parameters for the remaining loops: k_c, m_c and n_c

After deriving optimal values for m_r, n_r analytically, we discuss next how to proceed for k_c, m_c and n_c.

4.3.1 Strategy for identifying k_c, m_c and m_c

We recall that k_c, m_c and n_c (together with n_r) define the dimensions of the reused data for Loops 3 to 5; see Table 1. Since the reused blocks are mapped to the different layers of the memory hierarchy, this implies that there is a natural upper bound on k_c, m_c and n_c, imposed by the sizes of the caches. In addition, because the reused data should ideally be kept in cache between iterations, the cache replacement policy and the cache organization impose further restrictions on the optimal values for k_c, m_c and n_c.

In the remainder of this section, we describe our analytically model for identifying values for these characteristic parameters. For clarity, the discussion will focus on identifying the optimal value for k_c. The values for m_c and n_c can be derived in a similar manner.

4.3.2 Keeping B_r in the L1 cache

Recall that the micro-panel B_r is reused in every iteration of Loop 5. As such, it is desirable for B_r to remain in the L1 cache while Loop 5 is being executed. Since the L1 cache implements an LRU replacement policy, conventional wisdom suggests choosing the sizes of the different parts of the matrices such that data required for two consecutive iterations of the loop are kept in cache. This implies that the cache should be filled with B_r, two micro-panels of A_c, and two micro-tiles of C_c. The problem with this approach is that keeping two micro-panels of A_c and two micro-tiles of C_c consequently reduces the size of the micro-panel B_r that fits into the L1 cache, which means that the value k_c is smaller, hence decreasing the amount of data that could possibly be reused. In addition, k_c is the number of iterations for Loop 6, and reducing this value also means less opportunities to amortize data movement with enough computation in the micro-kernel.

Instead, we make the following observations:

1. At each iteration of Loop 5, a micro-panel A_r from A_c and the reused micro-panel B_r are accessed exactly once. This implies that the micro-panel that is loaded first will contain the least-recently-used elements. Therefore, because of the LRU cache replacement policy, as long as B_r is loaded after A_r, the entries from A_r will be evicted from the cache before those from B_r.

2. Since each micro-panel A_r is used exactly once per iteration, it is advantageous to overwrite the entries of the micro-panel A_r^{prev} which was used in the previous iteration with the corresponding entries of the new A_r^{next} that will be used in the present iteration. Doing so potentially allows a larger B_r to fit into the cache, hence increasing the amount of data being reused.

4.3.3 Evicting A_r^{prev} from cache

We assume that within the micro-kernel (Loop 6), the elements from B_r are loaded after those of A_r. From the second observation above, a strategy to keep a large micro-panel B_r in cache is to evict the old micro-panel A_r^{prev} from the cache, loaded in the previous iteration of Loop 5, by replacing its entries with those of the new micro-panel A_r^{next} to be used in the current iteration of the loop. To ensure this, we need to enforce that the same entries of all micro-panels of A_c are mapped to the same cache sets. Since the L1 cache comprises N_{L1} sets, then the memory addresses that are $N_{\text{L1}}C_{\text{L1}}$ bytes apart will be mapped to the same set in the L1 cache. This implies that the corresponding elements of consecutive micro-panels of A_c must lie in memory an integer multiple (say C_{A_r}) of $N_{\text{L1}}C_{\text{L1}}$ bytes apart.

Recall that consecutive micro-panels of A_c are packed contiguously in memory, and each micro-panel of A_c contains exactly $m_r \times k_c$ elements. This means that the distance between the same entries of two consecutive micro-panels of A_c must be $m_r k_c S_{\text{DATA}}$ bytes. Therefore, A_r^{prev} will be replaced by A_r^{next} only if

$$m_r k_c S_{\text{DATA}} = C_{A_r} N_{\text{L1}} C_{\text{L1}}.$$

Rearranging the above expression yields the following expression for k_c,

$$k_c = \frac{C_{A_r} N_{\text{L1}} C_{\text{L1}}}{m_r S_{\text{DATA}}}, \tag{4}$$

which ensures that a newly read micro-panel of A_c is mapped to the same set as the existing micro-panel of A_c. Thus, solving for k_c in (4) is equivalent to finding C_{A_r}, since the remaining variables in the equation are hardware parameters.

4.3.4 Finding C_{A_r}

The astute reader will recognize that C_{A_r} is the number of cache lines taken up by a micro-panel A_r in each set of the L1 cache. Similarly, we can define C_{B_r} as the number of cache lines in each set dedicated to the micro-panel B_r. In order for B_r to remain in a W_{L1}-associative cache, it is necessary that

$$C_{A_r} + C_{B_r} \leq W_{\text{L1}}.$$

In practice, the number of cache lines filled with elements of A_r and B_r has to be strictly less than the degree of associativity of the cache. This is because at least one cache line must be used when the entries of a micro-tile C_r are loaded into the registers. Since C_r is not packed, it is possible that its entries are loaded into the same set as the entries of A_r and B_r. Hence, we update the previous expression to account for the loading of C_r as follows

$$C_{A_r} + C_{B_r} \leq W_{\text{L1}} - 1. \tag{5}$$

As the micro-panel of B_c is packed into $n_r k_c S_{\text{DATA}}$ contiguous memory, we can compute C_{B_r} as follows:

$$C_{B_r} \;=\; \left\lceil \tfrac{n_r k_c S_{\text{DATA}}}{N_{\text{L1}} C_{\text{L1}}} \right\rceil = \left\lceil \tfrac{n_r}{m_r} C_{A_r} \right\rceil.$$

Therefore, replacing this expression in the inequality in (5), we get

$$C_{A_r} \;\leq\; \left\lfloor \tfrac{W_{\text{L1}}-1}{1+\frac{n_r}{m_r}} \right\rfloor,$$

which suggests choosing C_{A_r} as large as possible and then allows us to solve for k_c.

4.3.5 Two-way set associativity

Our value for k_c was chosen to enforce A_r^{next} to be loaded into those cache lines previously occupied by A_r^{prev}. To achieve this effect, we made two assumptions:

- Each set in the cache has to be filled equally with the micro-panel of A_r; i.e. the number of cache lines in each set containing elements from A_r is the same across all sets. This assumption ensures that the corresponding elements of different micro-panels will be assigned to the same cache set.
- One cache line in each set of the cache is reserved for the elements of C_r so that loading them will not evict the micro-panel B_r that already resides in cache.

The problem with a two-way set associative cache is that satisfying these two assumptions would imply that there remain no available cache lines to hold the elements of B_r, precisely the block that we want to keep in cache.

However, if the size of A_r is $N_{\mathrm{L1}}C_{\mathrm{L1}}/k$, where N_{L1} is an integer multiple of k, then the micro-panel of A_r that is loaded in the $(k+1)$-th iteration will be mapped to the same cache sets as the first micro-panel of A_r. When $k = 2$, this is identical to keeping two iterations worth of data in the cache, which ensures that the micro-panel of B_r is kept in cache. Any larger value of k decreases the size of the micro-panel of A_r, which implies that k_c is reduced. Therefore,

$$m_r k_c S_{\mathrm{DATA}} \quad = \quad \tfrac{N_{\mathrm{L1}}C_{\mathrm{L1}}}{2},$$

which implies that k_c is given by the formula:

$$k_c = \frac{N_{\mathrm{L1}}C_{\mathrm{L1}}}{2m_r S_{\mathrm{DATA}}} \tag{6}$$

when the cache is 2-way set associative.

5 Validation

In this section, we compare the optimal values derived via our analytical model with those employed by implementations that were either manually tuned by experts or empirically tuned. Unlike previous work [28, 16] that compared performance against an implementation based on ATLAS, we chose to compare our parameter values against the parameter values from manually-optimized implementations using the BLIS framework. As the algorithms used are identical, any deviation is the result of a difference in the parameter values.

For this experiment, we chose the following mixture of traditional (x86) architectures and low-power processors: Intel Dunnington (X7660), Intel SandyBridge (E3-1220), AMD Piledriver (A10-5800K), and Texas Instrument C6678. This sample includes both dated (end-of-life) and recent architectures in order to evaluate the accuracy and validity of the model, while taking into account different trends in processor architecture design. We consider double precision real arithmetic to obtain our parameter values.

5.1 Evaluating the model for m_r and n_r

Table 2 lists the machine parameters necessary for computing m_r and n_r, the optimal values analytically derived for m_r and n_r using the model, and the values of m_r and n_r chosen by the expert when implementing a micro-kernel using the BLIS framework. In addition we include the expert's values for the micro-kernel

Architecture	N_{VEC}	L_{VFMA}	N_{VFMA}	Analytically derived		Expert BLIS		Expert OpenBLAS	
				m_r	n_r	m_r	n_r	m_r	n_r
Intel Dunnington	2	5+3	1	4	4	4	4	4	4
Intel SandyBridge	4	5+3	1	8	4	8	4	8	4
AMD Piledriver	2	6	2	4	6	4	6	8	2
TI C6678	2	3+4	1	4	4	4	4	-	-

Table 2: Comparison between analytical values for m_r, n_r and empirical values chosen by experts.

underlying OpenBLAS. The instruction latency for the different architectures were obtained from the vendors' instruction set/optimization manuals. In these cases where the architecture did not include an actual VFMA instruction, we computed L_{VFMA} by adding the latencies for a floating-point multiply and a floating-point addition.

Note that for all the architectures our analytical values match those chosen by the BLIS experts who were not aware of this parallel research. Furthermore, the values for m_r and n_r used in OpenBLAS also match our analytical values in two out of three cases[3]. We note with interest that OpenBLAS utilized a 8×2 micro-kernel for the AMD Piledriver. While it differs from our analytical 4×6 micro-kernel, we note that AuGEMM [26], an empirical search tool developed by the authors of OpenBLAS in order to automatically generate the micro-kernel, generated a micro-kernel that operates on a 6×4 micro-tile. This suggests that the original 8×2 micro-kernel currently used by OpenBLAS may not be optimal for the AMD architecture.

The results provide evidence that our analytical model for m_r and n_r is reasonably robust across a variety of architectures.

5.2 Evaluating the model for k_c and m_c

Table 3 presents the machine parameters to derive k_c and m_c using the model, the optimal analytical values, and the empirical values adopted in BLIS after a manual optimization process (inside parenthesis). We do not report results for n_c because most of these processor architectures do not include an L3 cache, which means that n_c is, for all practical purposes, redundant. For the SandyBridge processor, there was minimal variation in performance when n_c was modified.

Again, our analytical model yields similar values if not identical for both k_c and m_c. The analytical model offered the same values the expert picked when optimizing for k_c, with the only exception of the AMD Piledriver. On investigation, we found that the expert picked $k_c = 120$, instead of $k_c = 128$ for this particular architecture because the BLIS framework required k_c to be a multiple of both m_r and n_r. With this constraint, the expert chose a value for k_c that our analytical model deemed to be smaller than the optimal one for the L1 cache.

We expected to encounter more variation between the expert-chosen values and the manually-tuned ones for the m_c parameter. This is because most L2 caches are unified (i.e., they contain both instructions and data), which makes predicting the cache replacement behavior of the L2 cache more difficult, as it depends on both the amount of data and instructions in that level of the cache. Nonetheless, we note that with the exception, once more, of the AMD Piledriver, the m_c values computed by our analytical model were similar if not identical to those chosen by the expert.

[3]There exist no OpenBLAS implementations for the remaining architectures.

Architecture	S_{L1} (Kbytes)	W_{L1}	N_{L1}	k_c	S_{L2} (Kbytes)	W_{L2}	N_{L2}	m_c
Intel Dunnington	32	8	64	256 (256)	3,072	12	4,096	384 (384)
Intel SandyBridge	32	8	64	256 (256)	256	8	512	96 (96)
AMD PileDriver	16	4	64	128 (120)	2,048	16	2,048	$1,792$ $(1,088)$
TI C6678	32	4	256	256 (256)	512	4	2,048	128 (128)

Table 3: Comparison between analytical values for k_c, m_c and empirical (manually-optimized) values from existing implementations of BLIS (inside parenthesis).

It is interesting to note that the analytical m_c value for the AMD Piledriver was less than half of the empirical value. The empirical value suggests that only half the L2 cache is filled with the packed block of A_c, which implies that the remaining half of the L2 cache is (mostly) filled with the packed block of B_c. One possible reason for this is that the bandwidth between main memory and the L2 cache is limited. This would mean that more elements of B_c are required to be kept in the L2 cache in order to hide the time it would take to bring elements from main memory into the L2 cache. We are currently investigating this.

6 Conclusion and Future Directions

We have developed an analytical model that yields optimal parameter values for the GEMM algorithm that underlies BLIS as well as other manually-tuned high performance dense linear algebra libraries. The key to our approach is the recognition that many of the parameters that characterize BLIS are bounded by hardware features. By understanding how the algorithm interacts with the processor architecture, we can choose parameter values that leverage, more aggressively, hardware features such as the multi-layered cache memory of the system.

We compare the values obtained via our analytical approach with the values from manually-optimized implementations and report that they are similar if not identical. Unlike similar work that compares with ATLAS, this is the first paper, to the best of our knowledge, that demonstrates that an analytical approach to identifying parameter values can be competitive with best-in-class, expert-optimized implementations. This demonstrates that a high performance implementation for the GEMM algorithm can be obtained without relying on a costly empirical search.

We believe that more can be done in terms of enhancing the analytical model. One possible direction we are exploring is to extend the analytical model to more complicated linear algebra operations such as those in LAPACK [2]. In general these operations, e.g. matrix factorizations, are implemented as blocked algorithms, and the GEMM operation is often a sub-operation in the loop body. It would be interesting to determine how to identify optimal block sizes, given that the parameters that favor performance for the GEMM operation are known.

A second direction to enhance the analytical model is to incorporate data movement considerations. Currently, the parameters for the micro-kernel (m_r and n_r) are determined by only considering the latency of the floating-point arithmetic instructions. However, this makes the assumption that bandwidth is large enough. On low-power systems, this assumption may no longer hold. In such scenario, m_r and n_r may have to be larger, so that the time for computation is sufficiently long to hide the time it takes to load the next elements of A_r and B_r into the processor registers.

Finally, our current analytical model assumes double precision arithmetic. With complex arithmetic, a micro-tile of the same size incurs four times as many flops, and involves twice as much data. These changes necessarily mean that the formulas in this paper have to be updated. Nonetheless, we believe that a similar analysis of the algorithm and the hardware features yields an analytical model for a micro-kernel that operates with complex arithmetic.

A question becomes whether our analysis can be used to better design future architectures. This question is at least partially answered in [22, 21], which examines how to design specialized hardware (both compute core and entire processor) for linear algebra computation. The models used for such purpose have much in common with our model for determining the parameter values for the micro-kernel. This suggests that our model for determining the parameter values for loops around the micro-kernel can potentially be leveraged to either determine the ideal cache size and/or cache replacement policy.

Acknowledgments

This research was sponsored in part by NSF grants ACI-1148125/1340293 and CCF-0917167.

Enrique S. Quintana-Ortí was supported by project TIN2011-23283 of the *Ministerio de Ciencia e Innovación* and FEDER. Francisco D. Igual was supported by project TIN2012-32180 of the *Ministerio de Ciencia e Innovación*. This work was partially performed during their visit to The University of Texas at Austin (UT), funded by the JTO visitor applications programme from the Institute for Computational Engineering and Sciences (ICES) at UT.

References

[1] AMD. AMD Core Math Library. `http://developer.amd.com/tools/cpu/acml/pages/default.aspx`, 2012.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide*. SIAM, 3rd edition, 1999.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, Dec. 1994.

[4] G. Belter, J. G. Siek, I. Karlin, and E. R. Jessup. Automatic generation of tiled and parallel linear algebra routines: A partitioning framework for the BTO compiler. In *Proceedings of the Fifth International Workshop on Automatic Performance Tuning (iWAPT10)*, pages 1–15, 2010.

[5] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[6] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE*, page 2005, 2005.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[10] Kazushige Goto and Robert van de Geijn. High performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, July 2008.

[11] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[12] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub., San Francisco, 2003.

[14] IBM. Engineering and Scientific Subroutine Library. `http://www.ibm.com/systems/software/essl/`, 2012.

[15] Intel. Math Kernel Library. `http://developer.intel.com/software/products/mkl/`, 2012.

[16] Vasilios Kelefouras, Angeliki Kritikakou, and Costas Goutis. A matrixmatrix multiplication methodology for single/multi-core architectures using simd. *The Journal of Supercomputing*, pages 1–23, 2014.

[17] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H. A. G. Wijshoff. Iterative compilation in program optimization, 2000.

[18] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. Iterative compilation. In Ed F. Deprettere, Jrgen Teich, and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2002.

[19] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[20] `http://xianyi.github.com/OpenBLAS/`, 2012.

[21] Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn. On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 24th International Symposium on*, October 2012.

[22] Ardavan Pedram, Robert A. van de Geijn, and Andreas Gerstlauer. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers*, 61:1724–1736, December 2012.

[23] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *IPDPS '14: Proceedings of the International Parallel and Distributed Processing Symposium*, 2014. To appear.

[24] Field G. Van Zee, Tyler Smith, Bryan Marker, Tze Meng Low, Robert A. van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, and Lee Killough. The BLIS framework: Experiments in portability. *ACM Trans. Math. Soft.*, 2014. In review.

[25] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for generating blas-like libraries. *ACM Trans. Math. Soft.*, 2014. To appear.

[26] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013. ACM.

[27] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[28] Kamen Yotov, Xiaoming Li, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.