Implementing Level-3 BLAS with BLIS: Early Experience

FLAME Working Note #69

Field G. Van Zee[*†] Tyler Smith[*†] Francisco D. Igual[‡] Mikhail Smelyanskiy[§]
Xianyi Zhang[¶] Michael Kistler[‖] Vernon Austel[**] John Gunnels[**] Tze Meng Low[*†]
Bryan Marker[*] Lee Killough[††] Robert A. van de Geijn[*†]

April 27, 2013

Abstract

BLIS is a new software framework for instantiating high-performance BLAS-like dense linear algebra libraries. We demonstrate how BLIS acts as a productivity multiplier by using it to implement the level-3 BLAS on a variety of current architectures. The systems for which we demonstrate the framework include state-of-the-art general purpose, low-power, and special purpose architectures. We show how, with very little effort, the BLIS framework yields sequential and parallel implementations that are competitive with the performance of ATLAS, OpenBLAS (an effort to maintain and extend the GotoBLAS), and commercial vendor implementations such as AMD's ACML, IBM's ESSL, and Intel's MKL libraries. While most of this paper focuses on single core implementation, we also provide compelling results that suggest the framework's leverage extends to the multithreaded domain.

1 Introduction

This paper discusses early results for BLIS (**B**LAS-like **L**ibrary **I**nstantiation **S**oftware) [26], a new framework for instantiating Basic Linear Algebra Subprograms (BLAS) [17, 5, 4] libraries. BLIS provides a novel infrastructure that refactors, modularizes, and expands existing BLAS implementations [7, 8], thereby accelerating DLA portability across the wide range of current and futures architectures. An overview of the framework is given in [26].

The current paper demonstrates the ways in which BLIS is a productivity multiplier: it greatly simplifies the task of instantiating BLAS functionality. We do so by focusing on the level-3 BLAS (a collection of fundamental matrix-matrix operations). Here, the essence of BLIS is its micro-kernel, in terms of which all level-3 functionality is expressed and implemented. Only this micro-kernel needs to be customized for a given architecture; routines for packing as well as the loops that block through matrices (to improve data locality) are provided as part of the framework. Aside from coding the micro-kernel, the developer needs only choose appropriate cache and register block sizes to instantiate highly tuned DLA implementations on a given architecture, with virtually no additional work.

A team of BLAS developers, most with no previous exposure to BLIS, was asked to evaluate the framework. They wrote only the micro-kernel for double-precision real general matrix multiplication (DGEMM) for architectures of their choosing. The architectures on which we report include general purpose processors (AMD A10, Intel® Xeon™ E3-1220 "Sandy Bridge E3", and IBM Power7), low-power processors (ARM Cortex-A9, Loongson 3A, and IBM Blue Gene/Q PowerPC A2), and special purpose architectures (Intel® Xeon Phi™ and Texas Instruments C6678 DSP). We believe this selection of architectures is illustrative of the main solutions available in the HPC arena, with the exception of GPUs (see comment in the conclusion). With moderate effort, not only were full implementations of

[*]Department of Computer Science, The University of Texas at Austin, Austin, TX 78712.
[†]Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712.
[‡]Dept. Arquitectura de Computadores y Automática, Unividad Complutense de Madrid, 28040, Madrid (Spain).
[§]Parallel Computing Lab, Intel Corporation, Santa Clara, CA 95054.
[¶]Institute of Software and Graduate University of Chinese Academy of Sciences, Beijing 100190 (China).
[‖]Austin Research Laboratory, IBM Corporation, Austin, TX 78758.
[**]Thomas J. Watson Research Center, IBM Corporation, Yorktown Heights, NY 10598.
[††]Cray Inc., Seattle, WA 98164.

DGEMM created for all of these architectures, but the implementations attained performance that rivals that of the best available BLAS libraries. Furthermore, the same micro-kernels, without modification, are shown to support high performance for the remaining level-3 BLAS (for a single core, at present). Finally, by introducing simple OpenMP [20] pragmas, multithreaded parallelism was extracted.

2 Why a new open source library?

Open source libraries for BLAS-like functionality provide obvious benefits to the computational science community. While vendor libraries like ESSL from IBM, MKL from Intel, and AMCL from AMD provide high performance at nominal or no cost, they are proprietary implementations. As a result, when new architectures arrive, a vendor must either (re)implement a library or depend on an open source implementation that can be easily retargeted to the new architecture. This is particularly true when a new vendor joins the high-performance computing scene.

In addition, open source libraries can facilitate research. For example, as architectures strive to achieve lower power consumption, hardware support for fault detection and correction may be sacrificed. Incorporation of algorithmic fault-tolerance [13] into BLAS libraries is one possible solution [11]. For this, a well-structured open source implementation facilitates research related to algorithmic fault-tolerance.

Prior to BLIS, there were two open source options for high performance BLAS: ATLAS [27] and OpenBLAS [19], a fork of the widely used GotoBLAS [7, 8] implementation that is itself no longer supported. The problem is that neither is easy to maintain or modify and we would argue that neither will easily facilitate such research[1].

As a framework, BLIS has commonality with ATLAS. In practice, ATLAS requires a hand-optimized kernel. Given this kernel, ATLAS tunes the blocking of the computation in an attempt to optimize the performance of matrix-matrix routines (although in [29] it is shown parameters can be determined analytically). Similarly, BLIS requires a kernel to be optimized. But there are many important differences. BLIS mimics the algorithmic blocking performed in the GotoBLAS. To our knowledge, on nearly all architectures, the GotoBLAS outperforms ATLAS, often by a wide margin. BLIS implements a fundamentally better approach, which is based on theoretical insights [10] that almost always outperform ATLAS. We believe BLIS to be layered in a way that makes the code easier to understand than both the autogenerator that generates ATLAS implementations and those generated implementations themselves. ATLAS-like approaches have other drawbacks. For example, some architectures require the use of cross compilation processes, which invalidate the possibility of auto-tuning at installation time. ATLAS is also impractical when used with simulators when designing future processors. This we believe will make BLIS both higher-performing and more flexible than ATLAS.

While BLIS mimics the algorithms that Goto developed for the GotoBLAS (and thus OpenBLAS), we consider his implementations difficult to understand, maintain, and extend. Thus, they will be harder to port to future architectures and more cumbersome when pursuing new research directions. In addition, as we will discuss later, BLIS casts Goto's "inner kernel" in terms of a smaller micro-kernel that requires less code to be optimized and, importantly, facilitates the optimization of all level-3 BLAS. These extra loops also expose convenient opportunities for parallelism.

3 A layered implementation

In many ways, the BLIS framework is a reimplementation of the GotoBLAS software that increases the reuse of code via careful layering. We now describe how such an approach layers the implementation of matrix-matrix multiplication, how BLIS mimics this, and how BLIS differs.

A layered approach The GEMM operation computes $C := \alpha AB + \beta C$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$, respectively. For simplicity, we will assume that $\alpha = \beta = 1$.

It is well-known that near-peak performance can already be attained for the case where A and B are $m \times k_c$ and $k_c \times n$, respectively, where block size k_c will be explained shortly. A loop around this special case implements the general case, as illustrated in the bottom layer of Figure 1.

[1]We acknowledge that in this paper we present no evidence that BLIS is any easier to decipher, maintain, or extend. The reader will have to investigate the source code itself.

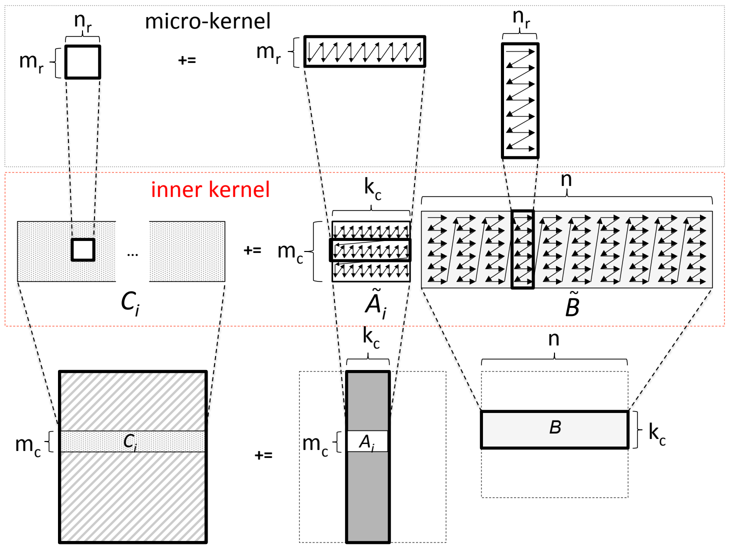


Figure 1: Illustration from [26] of the various levels of blocking and related packing when implementing GEMM in the style of [7]. The bottom layer shows the general GEMM and exposes the special case where $k = k_c$ (known as a rank-k update, with $k = k_c$). The middle layer shows what is the inner kernel in the GotoBLAS. The top layer shows the micro-kernel upon which the BLIS implementation is layered. Here, m_c and k_c serve as cache block sizes used by the higher-level blocked algorithms to partition the matrix problem down to a so-called "block-panel" subproblem (depicted in the middle of the diagram), implemented in BLIS as a portable "macro-kernel". Similarly, m_r and n_r serve as register block sizes for the micro-kernel in the m and n dimensions, respectively, which also correspond to the length and width of the individual packed panels of matrices $\tilde{A}_i$ and $\tilde{B}$, respectively.

To implement this special case ($k = k_c$) matrix C is partitioned into row panels, C_i, that are $m_c \times n$, while A (which is $m \times k_c$) is partitioned into $m_c \times k_c$ blocks, A_i. Then $C := AB + C$ means that $C_i := A_i B + C_i$. Now, B is first "packed" into contiguous memory (array $\tilde{B}$ in Figure 1). The packing layout in memory is indicated by the arrows in that array. Next, for each C_i and A_i, the block A_i is packed into contiguous memory as indicated by the arrows in $\tilde{A}_i$. Then, $C_i := \tilde{A}_i \tilde{B} + C_i$ is computed with a so-called "inner kernel," which an expert assembly codes for a specific architecture. In his approach, $\tilde{A}_i$ typically occupies half of the L2 cache and $\tilde{B}$ is in main memory (or the L3 cache).

The BLIS approach The BLIS framework takes the inner kernel and breaks it down into a double loop over what we call "the micro-kernel." The outer loop[2] of this is already described above: it loops over the n columns of B, as stored in $\tilde{B}$, n_r columns at a time. The inner loop views A_i, stored in $\tilde{A}_i$, as panels of m_r rows. These loops (as well as all loops required to block down to this point) are coded in C99. It is then the multiplication of the row panel of $\tilde{A}_i$ times the column panel of $\tilde{B}$ that updates an $m_r \times n_r$ block of C, which is typically kept in registers during this computation. The dimensions m_r and n_r refer to the "register block sizes," which determine the size of the small block of C_i that is updated by the micro-kernel, which is illustrated in the top layer of Figure 1. It is only this micro-kernel, which contains only a single loop over the k dimension, that needs to be highly optimized for a new architecture. Notice that this micro-kernel is implicitly present within the inner kernel of GotoBLAS. BLIS exposes it explicitly as the only routine that needs to be highly optimized for high-performance level-3 functionality. All loops implementing layers above the micro-kernel are written in C and thus fully portable to other architectures.

[2]Actually, there is one more loop over blocks of columns of C and B that reduces the work space requirements when packing into $\tilde{B}$. See [26] for details.

Beyond $C := AB + C$ The BLAS GEMM operation supports many cases, including those where A and/or B are (conjugate) transposed. The BLIS interface allows even more cases, namely, cases where only conjugation is needed as well as mappings to memory beyond column- or row-major storage, in any combination. The way the framework handles this burgeoning space of possible cases is by exploiting the fact that submatrices must always be packed to facilitate high performance; BLIS strategically casts an arbitrary special case into a common "base case" for which a high performance implementation is provided. Details can be found in [26].

Other matrix-matrix operations A key feature of the layering of BLIS in terms of the micro-kernel is that it makes the porting of other matrix-matrix operations (level-3 BLAS) simple. In [16] it was observed that other level-3 BLAS can be cast in terms of GEMM. The GotoBLAS took this one step further, casting the implementation of most of the level-3 BLAS in terms of its inner kernel [8]. (ATLAS has its own, slightly different, inner kernel.) However, this approach still required coding separate inner kernels for some operations (HERK, HER2K, SYRK, SYR2K, TRMM, and TRSM) due to the changing assumptions of the structure of either the input matrix A or the output matrix (B or C). BLIS takes this casting of computation in terms of fewer and smaller units to what we conjecture is the limit: the BLIS micro-kernel.

The basic idea is that an optimal implementation will exploit symmetric, Hermitian, and/or triangular structure when such a matrix is present (as is the case for all level-3 operations except GEMM). Let us consider the case of the Hermitian rank-k update (HERK), which computes $C := AA^H + C$, where only the lower triangle of C is updated. Because GotoBLAS expresses its fundamental kernel as the inner kernel (i.e.: the middle layer of Figure 1), a GEMM kernel cannot be used when updating blocks that intersect the diagonal of matrix C, because such a kernel would illegally update parts of the upper triangle. To address this, GotoBLAS provides a separate specialized inner kernel that is used to update these diagonal blocks. (Yet *another* inner kernel is needed for cases where C is stored in the upper triangle.) Similar kernel specialization is required for other level-3 operations. These special, structure-aware kernels share many similarities, yet they contribute to, in the humble opinion of the authors, the worst kind of redundancy: assembly code bloat.

BLIS eliminates most of this code bloat by simply requiring a smaller kernel. It turns out that virtually *all* of the differences between these structure-aware kernels reside in the two loops around the inner-most loop corresponding to the micro-kernel. But because of its chosen design, the GotoBLAS implementation is forced to bury these slight differences in assembly code, which significantly hinders the maintainer, or anyone trying to read and understand the implementation. By contrast, since BLIS *already* compartmentalizes all architecture-sensitive code within the micro-kernel, the framework naturally allows us to provide generic and portable instances of these specialized inner kernels (which we call "macro-kernels"). Thus, BLIS simultaneously reduces the *size* the assembly kernel required as well as the *number* of kernels required. This also has the side effect of improving performance of the instruction cache at runtime.

4 Targeting a single core

The first BLIS paper [26] discussed preliminary performance on only one architecture, the Intel Xeon 7400 *"Dunnington"* processor. This section reports *first impressions* on many architectures, focusing first on single core and/or single thread performance. Multithreaded, multicore performance is discussed in the next section.

For all but one architecture, *only* the micro-kernel (that iterates over the k_c dimension in Figure 1) was created and adapted to the architectural particularities of the target platform. In addition, the various block sizes were chosen for the target architecture (we do not discuss here how those were chosen, for space reasons). Figure 2 summarizes the main BLIS parameters used. On one architecture, the Intel Xeon Phi, a more extensive implementation was attempted in order to examine modifications that may to be required in order to support many-core architectures.

The performance experiments examined double precision (DP) real versions of the following representative set of operations: DGEMM ($C := AB + C$); DSYMM ($C := AB + C$, A is stored in lower triangle); DSYRK and DSYR2K ($C := AB^T + BA^T + C$, C is stored in lower triangle); DTRMM ($C := AB$, A is lower triangular) and DTRSM ($C := A^{-1}B$, A is lower triangular). In all cases, the matrices were stored in column-major order.

We now describe a few details of our BLIS port to each architecture. Descriptions focus on issues of interest for the BLIS developer, and are not intended to be exhaustive. Our goal is to demonstrate how competitive performance can be reached with basic optimizations using the BLIS framework. For all architectures, there is room for improvement

Architecture	Clock (Ghz)	DP flops /cycle /FPU	# cores	FPUs/ core	DP Peak (GFLOPS)		Cache(Kbytes)			BLIS parameters		
					one core	system	L1	L2	L3	$m_r \times n_r$	$m_c \times k_c$	n_c
AMD A10 5800K	3.7	8	4	0.5[i]	29.6	60.8	16	2048	–	4×6	1088×128	8192
Sandy Bridge E3	3.1	8	4	1	24.8	99.2	32	256	8092	8×4	96×256	3072
IBM Power7	3.864	2	8	4	30.9	247.3	32	256	4096	8×4	64×256	8192
ARM Cortex A9	1	1	2	1	1	2	32	512	–	4×4	128×256	512
Loongson 3A	0.8	4	4	2	3.2	12.8	64	4096	–	4×4	32×128	1024
IBM BG/Q A2	1.6	8	16	1[ii]	12.8	204.8	16	32K	–	8×8	576×256	4096
Intel Xeon Phi	1.09	16	60	1	17.44	1046.4	32	512	–	30×8	120×240	–
TI C6678	1	1	8	4	4	32	32	512	4096	4×4	128×256	4096

[i] One FPU shared by 2 cores. [ii] Only one can be used in a given cycle.

Figure 2: Architecture summary.

Architecture	Compiler (version)	Compiler optimizations and architecture-specific flags	Micro-kernel comments
AMD A10 5800K	gcc (4.7)	-O3 -mavx -mfma3 -march=bdver2	C code + inline assembly code
Sandy Bridge E3	gcc (4.6)	-O3 -mavx -march=nocona -mfpmath=sse	C code + AVX intrinsics
IBM Power7	gcc (4.7.3)	-O3 -mcpu=power7 -mtune=power7	C code + ALTIVEC instrinsics
ARM Cortex A9	gcc (4.6)	-O3 -march=armv7-a -mtune=cortex-a9 -mfpu=neon -mfloat-abi=hard	C code
Loongson 3A	gcc (4.6)	-O3 -march=loongson3a -mtune-loongson3a -mabi=64	C code + inline assembly code
IBM BG/Q A2	gcc (4.4.6)	-O3	C code + inline assembly code
Intel Xeon Phi	icc (13.1.0)	-O3 -mmic	C code + inline assembly code
TI C6678	cl6x (7.4.1)	-O2 -mv6600 –abi=eabi	C code

Figure 3: Summary of compiler/implementation information.

within the framework. The architectures selected for our evaluation are briefly described in Figure 2. Details about compiler version and optimization flags are reported in Figure 3.

On each architecture, we timed the BLIS implementations as well as the vendor library, ATLAS, and/or OpenBLAS. In each set of graphs, the left graph reports the performance of the different DGEMM implementations, the middle graph the performance of BLIS for various level-3 BLAS operations, and the right graph speedup attained by the BLIS implementation for the various level-3 BLAS operations, relative to the vendor implementation or, if no vendor implementation was available, ATLAS or OpenBLAS. For the left and middle graphs, the top of the graph represents the peak of the architecture, when using the indicated number of threads/cores. The left graph reports how quickly the performance of DGEMM ramps up when $m = n = 2000$ and the k dimension is varied (in increments of 16). This matrix shape is important because a matrix-matrix multiply with relatively small k (known as a rank-k update) is often at the heart of higher level operations that have been cast in terms of DGEMM [2, 25]. Thus, the implementation should quickly ramp up to its asymptote as k is varied. The middle and right graphs report performance for square matrices, again in increments of 16.

4.1 Conventional architectures

The **AMD A10** processor implements the Trinity micro-architecture, an APU (Accelerated Processing Unit) that combines a reduced number of CPU cores (between 2 and 4) and a large number of GPU cores. The chosen processor (A10 5800K) incorporates 4 CPU cores and 384 GPU cores. Our implementation targets the 64-bit, x86 cores, which support out-of-order execution, and ignores the GPU cores. The CPU cores are organized by pairs, where each pair shares a single floating point unit (FPU). We investigate performance when one thread is used to drive the single, shared FPU.

The micro-kernel developed for the AMD A10 processor was written using assembly code and the GNU toolchain. Optimization techniques such as loop unrolling and cache prefetching were incorporated in the micro-kernel. The Trinity micro-architecture supports two formats of fused multiply-accumulate instructions, the FMA3 instruction, and the FMA4 instruction. We found the FMA3 instructions to be the faster of the two, hence we used the faster

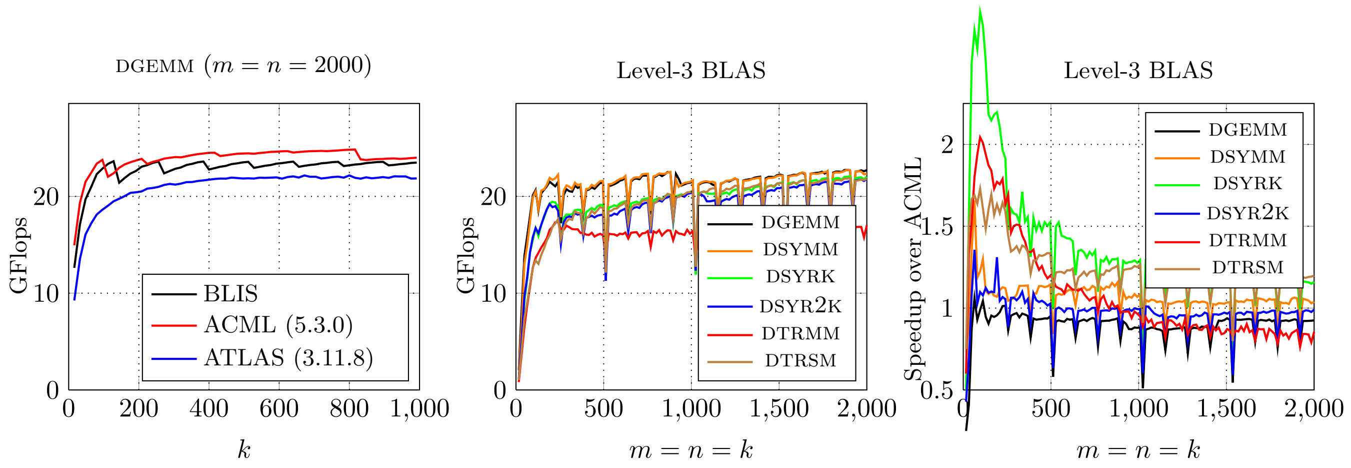


Figure 4: Performance on the AMD A10 processor (one core).

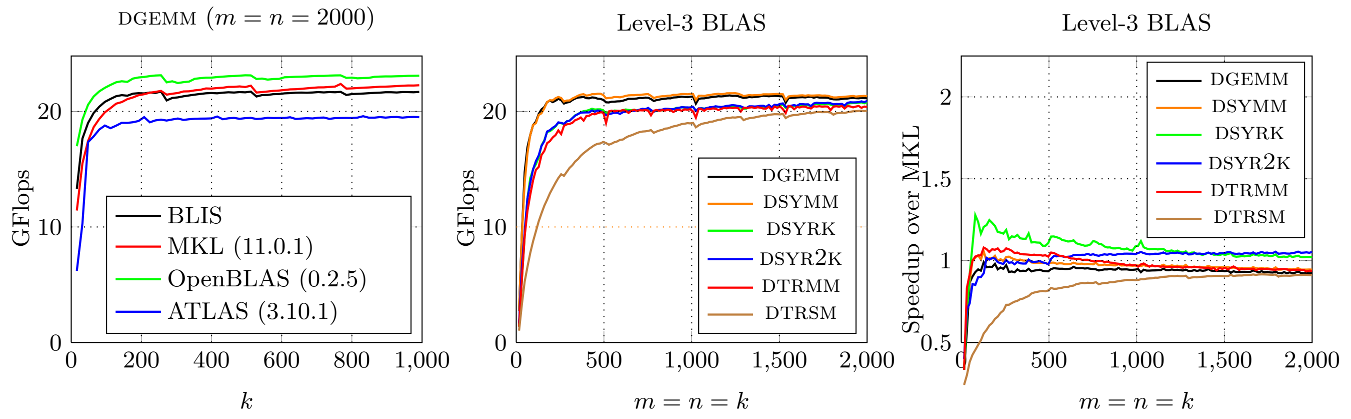


Figure 5: Performance on the Sandy Bridge E3 processor (one core).

FMA3 instruction in our micro-kernel.

Initial performance is reported in Figure 4. We compare against ACML 5.3.0 with the FMA3 instructions enabled and ATLAS. Although our DGEMM implementation does not (yet) match the performance of the ACML library, it is striking that the same micro-kernel facilitates highly competitive implementations of the other level-3 BLAS operations. Furthermore, it clearly outperforms ATLAS. The drop in performance for problem sizes that are multiples of 128 can be fixed by adjusting the leading dimension of C. How to more generally fix this problem is the focus of future improvements to the BLIS framework and/or the micro-kernel. While ACML's DGEMM outperforms BLIS, for the other level-3 BLAS the BLIS implementation outperforms ACML.

The **Sandy Bridge E3** processor is a 64-bit, x86 superscalar, out-of-order micro-architecture. It features three different building blocks, namely the CPU cores, GPU cores and System Agent. Each Sandy Bridge E3 core presents 15 different execution units, including general-purpose, vector integer and vector floating point (FP) units. Vector integer and floating point units support multiply, add, register shuffling and blending operations on up to 256-bit registers.

The micro-kernel developed for the Sandy Bridge E3 processor was written exclusively using plain C plus AVX intrinsics and the GNU toolchain. Common optimization techniques like loop unrolling, instruction reordering, and cache prefetching were incorporated in the micro-kernel.

Performance results for Sandy Bridge E3 are reported in Figure 5. BLIS DGEMM implementation clearly outperforms that of ATLAS for rectangular matrices, see left plot. OpenBLAS yields the best performance, improving BLIS by roughly 10% for large problem sizes, and attaining similar performance for small values of k. Intel MKL DGEMM yields an intermediate performance and outperforms BLIS by a small margin. The performance for the remaining Level-3 BLAS routines in BLIS is comparable with that of DGEMM. On this architecture, the implementation of

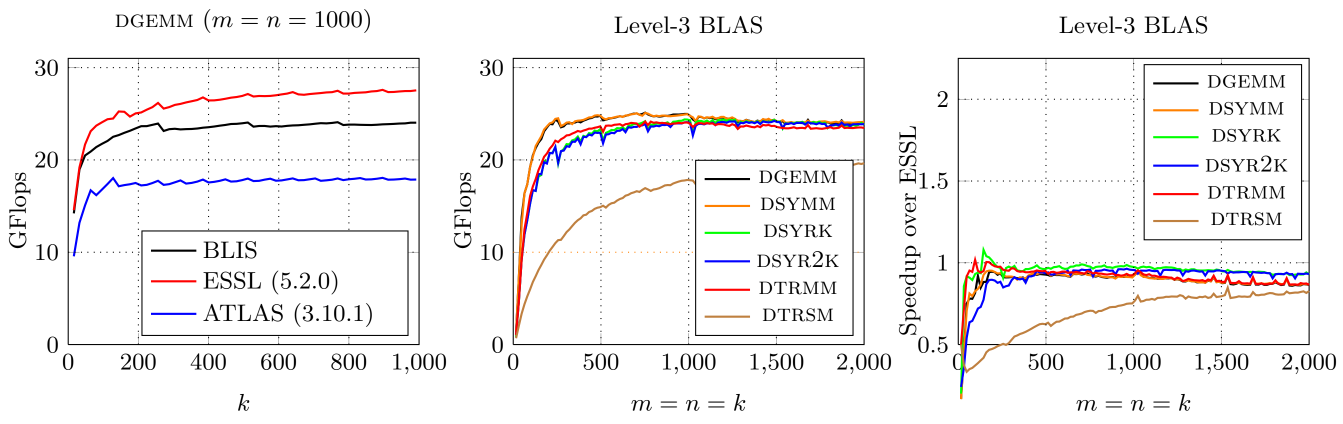


Figure 6: Performance on the IBM Power7 processor (one core).

DTRSM can clearly benefit from implementing the optional TRSM micro-kernel supported by the BLIS framework.

The **IBM Power7** processor is an 8-core server processor designed for both commercial and scientific applications [21]. Each Power7 core implements the full 64-bit Power architecture and supports up to four-way simultaneous multithreading (SMT). Power7 includes support for Vector-Scalar Extension (VSX) instructions, which operate on 128-bit VSX registers that can each contain 2 double-precision floating point values. Each Power7 thread has 64 architected VSX registers, but to conserve resources these are aliased on top of the existing FP and VMX architected registers. Other relevant details are catalogued in Figures 2 and 3.

The BLIS micro-kernel developed for Power7 was written almost entirely in plain C with ALTIVEC intrinsics for vector operations [6]. We compiled BLIS with Version 6.0 of the Advance Toolchain GCC, which is an open source GCC implementation that includes support for the latest features of IBM's Power architecture. The micro-kernel exploits the VSX feature of Power7 to perform all operations on vector data.

We conducted our experiments on an IBM Power 780 system with two 3.864 GHz Power7 processors running Red Hat Enterprise Linux 6.3. We configured ATLAS to use the architectural defaults for Power7 and built with the Advance Toolchain GCC. All executions are performed on one core with large (16MB) pages enabled.

Figure 6 presents BLIS performance on one core (executing one thread) of the IBM Power7 processor. BLIS DGEMM performance falls short of ESSL by about 10%, but with further optimization this gap could likely be narrowed considerably. The Level-3 BLAS performance graph shows that all of the level-3 operations except DTRSM achieve consistently high performance. Unlike other level-3 operations, a non-trivial portion of TRSM's computations does not employ the micro-kernel. BLIS does allow developers to provide supplementary optimized TRSM kernels, but these were not yet implemented for Power7.

4.2 Low power architectures

The following architectures distinguish themselves by requiring low power relative to the performance they achieve. It is well known that power consumption is now the issue to overcome and hence an evaluation of how well BLIS ports to these architectures is merited.

The **ARM Cortex-A9** processor is a low-power RISC processor that implements a dual-issue superscalar, out-of-order pipeline. Its features depend on the specific implementation of the architecture; in our case, the Texas Instruments OMAP4-4430 selected for evaluation incorporates two Cortex-A9 cores.

The micro-kernel developed for the ARM Cortex-A9 processor was written exclusively using plain C code and the GNU toolchain. As no NEON extensions for DP arithmetic exist, no vector intrinsics were used in the micro-kernel to get our DP performance results. Common optimization techniques like loop unrolling, a proper instruction reordering to hide memory latency, and cache prefetching are incorporated in the micro-kernel.

Performance is reported in Figure 7. As the only tuned BLAS implementation for ARM is ATLAS [3] as of today, we only compare with it. In general, for all tested routines and matrix dimensions, BLIS outperforms ATLAS; of special interest is the gap in performance for small problem sizes of most level-3 operations (with the exception of DGEMM, see right plot).

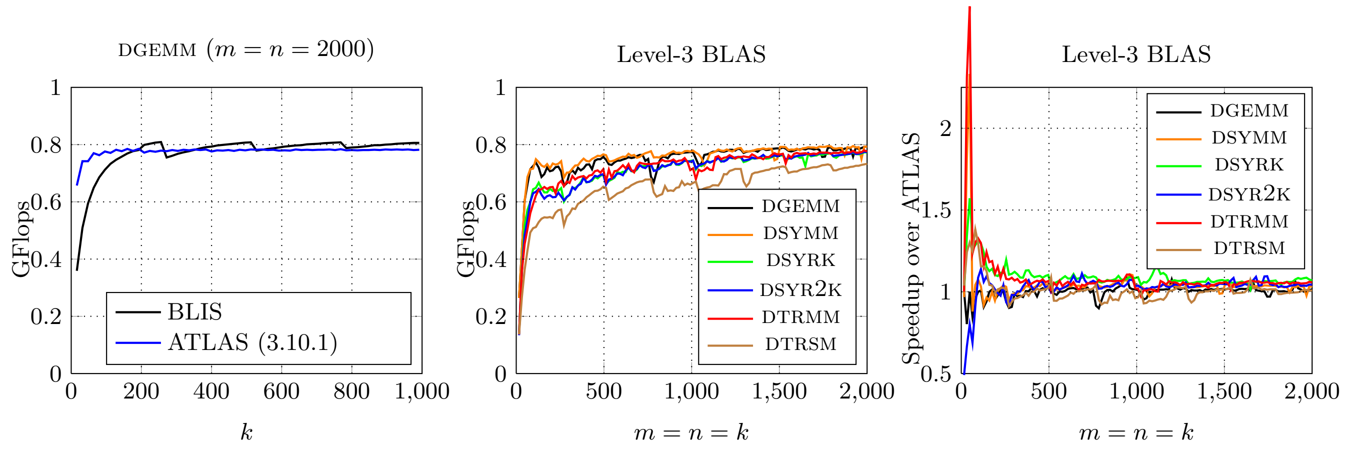


Figure 7: Performance on an ARM Cortex A-9 processor (one core).

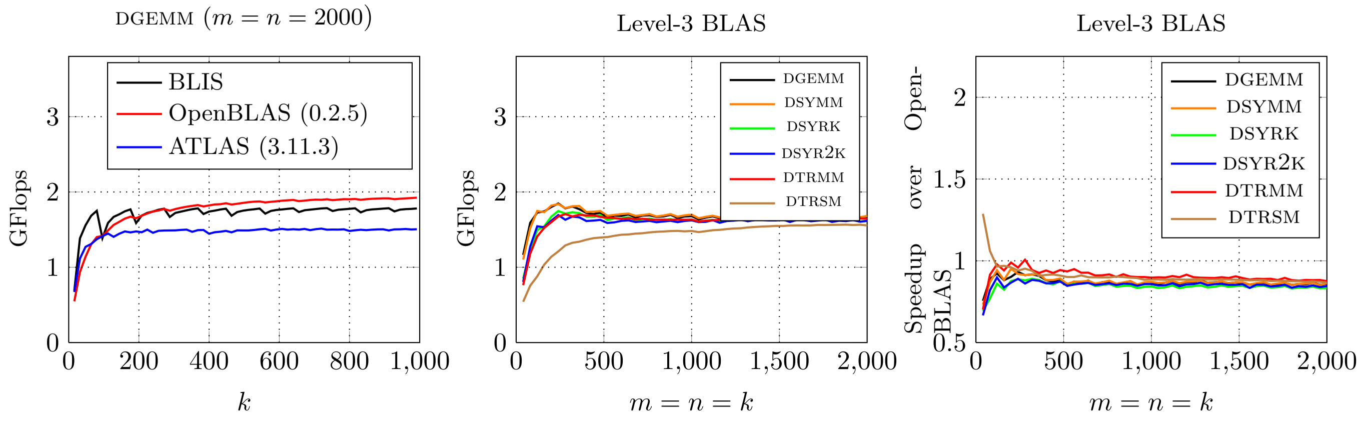


Figure 8: Performance on the Loongson 3A processor (one core).

The **Loongson 3A** CPU is a general-purpose 64-bit MIPS64 quad-core processor, developed by the Institute of Computing Technology, Chinese Academy of Sciences [18]. Each core supports 4-issue superscalar and out-of-order execution, and includes five pipelined execution units, two arithmetic logic units (ALU), two floating-point units (FPU) and one address generation unit (AGU). Every FPU is capable of executing single and double precision fused multiply-add instructions.

For this architecture, we optimized the BLIS DGEMM micro-kernel by using exclusively assembly code. Similar to our DGEMM optimization for the OpenBLAS [28], we adopted loop unrolling and instruction reordering, software prefetching, and the Loongson 3A specific 128-bit memory accessing extension instructions to optimize the BLIS micro-kernel. We performed a limited search for the best blocksizes m_c, k_c, and n_c.

Performance is reported in Figure 8. In this case, only open source libraries are available for the Loongson processor. Focusing on DGEMM, BLIS outperforms ATLAS by a wide margin, but this initial port of BLIS still does not perform quite at the same level as the OpenBLAS. Interestingly, by choosing the parameters for BLIS slightly different from those used for the OpenBLAS the performance of BLIS improved somewhat. This obviously calls for further investigation and performance tuning.

The **IBM Blue Gene/Q PowerPC A2** processor is composed of 16 application cores, one operating system core, and a redundant (spare) core. All 18 of the 64-bit PowerPC A2 cores are identical and designed to be both reliable and energy-efficient [14]. Mathematical acceleration for the kinds of kernels under study in this paper is achieved through the use of the 4-way double-precision SIMD QPX instructions that allow each core to execute up to 8 floating point operations in a single cycle [9]. Efficiency stems from the use of multiple (up to 4) symmetric hardware threads, each with their own register file. Multiple thread use enables dual issue capabilities (a floating point and a load/store operation in a single cycle), latency tolerance, and the reduction of bandwidth required. Other relevant details are

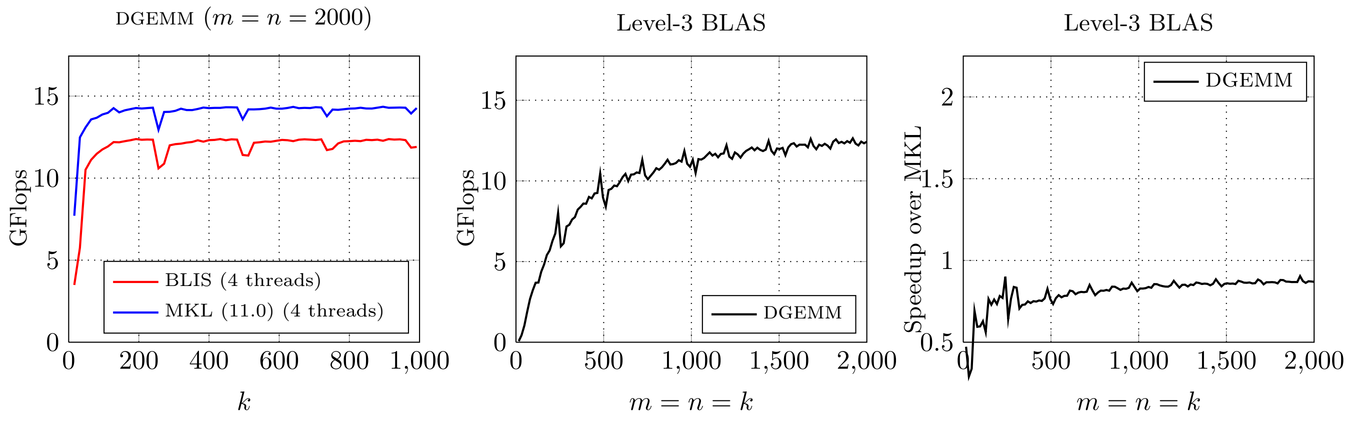


Figure 9: Performance on the Intel Xeon Phi processor (one core executing four threads).

catalogued in Figures 2 and 3.

The micro-kernel used for BLIS on Blue Gene/Q was written in GNU extended inline assembly and consists almost entirely of QPX instructions. Our experiments were carried out on a Blue Gene/Q node card wherein the compute nodes run IBM's CNK operating system. While a node card consists of 32 1.6 GHz processors (nodes), all results required the use of a single node (with 16 cores since one core is for the operating system and another is a "spare").

We do not present single core results since these would not be meaningful unless four threads were used. Instead, parallel performance with all sixteen cores is presented in the next section.

4.3 Special purpose architectures.

Finally, we examine a coprocessor and a processor initially intended for Digital Signal Processing (DSP).

The **Intel Xeon Phi coprocessor (KNC)** is the first production coprocessor in the Intel Xeon Phi product family. It features many in-order cores on a single die; each core has 4-way hyper-threading support to help hide memory and multi-cycle instruction latency. To maximize area and power efficiency, these cores are less aggressive, i.e., they have lower single-threaded instruction throughput than CPU cores and run at a lower frequency. However, each core has 32 vector registers, 512 bits wide, and its vector unit can sustatin a full 16-wide (8-wide) single (double) precision vector instructions in a clock cycle, and load 512 bits of data from the L1 data cache per cycle. Note that vector instructions can be paired with scalar instructions and data prefetches. Each core further has two levels of cache: a single-cycle access 32 KB first level data cache (L1) and a larger 512 KB second level cache (L2), which is globally coherent via directory-based MESI coherence protocol. The Intel Xeon Phi is physically mounted on a PCIe slot and has 8GB of dedicated GDDR5 memory.

In our experimentation we used Xeon Phi SE10P co-processor which has 61 cores running at 1.1 GHz, offering 17.1 GFLOPS of peak double-precision performance per core. It runs MPSS version 2.1.4346-16.

Our micro-kernel incorporates many of the insights in the paper by Heineke et al. [12]. Both the m_c by k_c block of $\hat{A}$ and the k_c by n_r block of $\hat{B}$ are streamed from the L2 cache. Because of this, and because prefetch instructions can be co-issued with floating-point operations, we aggressivly prefetch $\hat{A}$, $\hat{B}$, and $\hat{C}$ in the microkernel.

Each thread can only issue instructions every other clockcycle, thus it is necessary to use at least two threads per core to achieve maximum performance. In our implementation, we use four. These four threads share the block of A, so periodic thread synchronization is used to ensure data reuse of $\hat{A}$ within their shared L1 cache.

Performance for DGEMM is reported in Figure 9. We do not report performance for the rest of the level-3 operations, as we do not yet have implementations that utilize more than one thread per core. We see that our initial port of BLIS is only within 15% from highly tuned Intel MKL performance.

The **Texas Instruments C6678 DSP** incorporates the C66x DSP core from Texas Instruments [23], a Very Long Instruction Word (VLIW) architecture with eight different functional units in two independent sides, with connected but separate register files per side. This core can issue eight instructions in parallel per cycle [22]. The C66x instruction set includes SIMD instructions operating on 128-bit vector registers. Ideally, each core can perform up

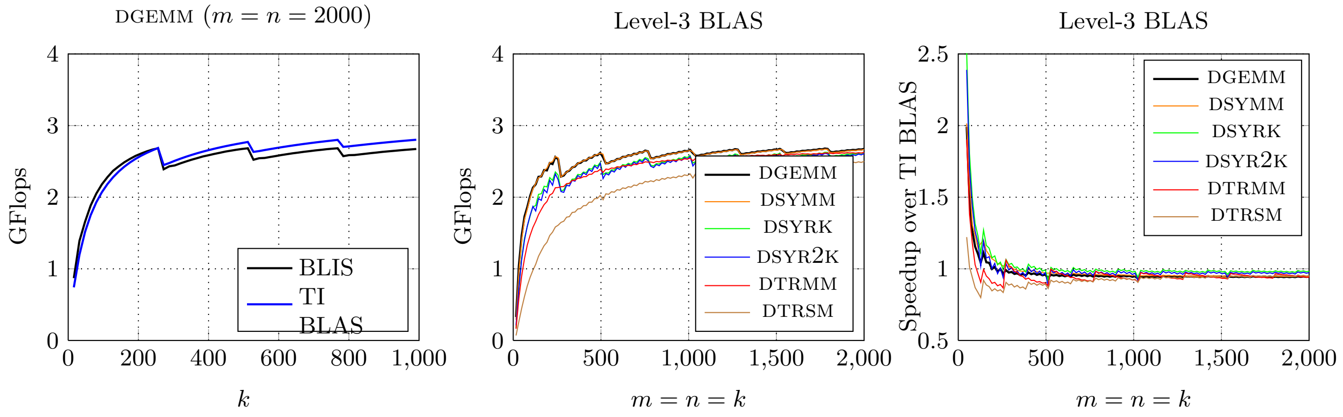
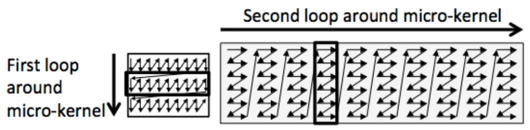


Figure 10: Performance on the Texas Instruments C66x DSP (one core).



Second loop around micro-kernel
First loop around micro-kernel

Figure 11: Illustration of parallelism opportunities within the "Goto inner kernel". The equivalent of that inner kernel is implemented as two loops around the BLIS micro-kernel. This exposes two extra opportunities for introducing loop parallelism.

to 8 single precision multiply-add (MADD) operations per cycle. In double precision, this number is reduced to 2 MADD operations per cycle. The C6678 DSP incorporates eight C66x cores, with a peak power consumption of 10W. Each level of the cache hierarchy can be configured either as software-managed RAM, cache, or part RAM/part cache. DMA can be used to transfer data between off-chip and on-chip memory without CPU participation.

The C66x architecture poses a number of challenges for BLIS; it is a completely different architecture (VLIW), and the software infrastructure for the TI DSP is dramatically different from the rest of our target architectures: the TI DSP runs a native real-time OS (SYS/BIOS), and an independent compiler (`cl6x`) is used to generate code. Despite that, the reference implementation of BLIS compiled and ran "out-of-the-box" with no further modifications.

Figure 10 reports BLIS performance compared with the only optimized BLAS implementation available as of today: the native TI BLAS implementation [1], which makes intensive use of DMA to overlap data movement between memory layers with computation and an explicit management of scratchpad memory buffers at the different levels of the memory hierarchy [15]. While this support is on the BLIS roadmap, it is still not supported; hence, no DMA support is implemented in our macro-kernel yet. Given the layered design of BLIS, this feature is likely to be easily integrated into the framework, and may be applicable to other architectures supporting DMA as well. Given the small gap in performance between BLIS and TI BLAS, we expect BLIS to be highly competitive when DMA capabilities are integrated in the framework.

5 Targeting multicore architectures

We now discuss basic techniques for introducing multithreaded parallelism into the BLIS framework.

The GotoBLAS are implemented in terms of the inner kernel discussed in Section 3 and illustrated in Figure 1. If that inner kernel is taken as a basic unit of computation, then parallelism is most easily extracted by parallelizing one or more of the loops around the inner kernel. By contrast, the BLIS framework makes the micro-kernel the fundamental unit of computation and implements Goto's inner kernels as two loops around the micro-kernel (Figure 11).

A complete study of how parallelism can be introduced by parallelizing one or more of the loops is beyond the scope of this paper. Instead, we simply parallelized the second loop around the micro-kernel using OpenMP [20] `pragma` directives, as well as the routines that pack a row panel of B and the block of A. Thus, individual threads

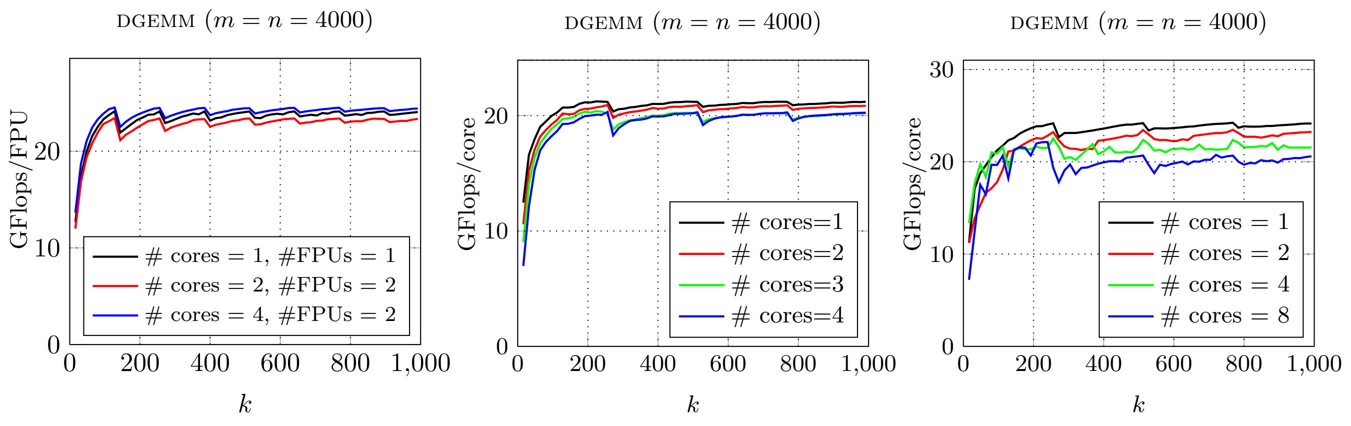


Figure 12: Parallel performance. Left: AMD A10. Middle: Intel Sandy Bridge (Xeon E3-1220). Right: IBM Power7.

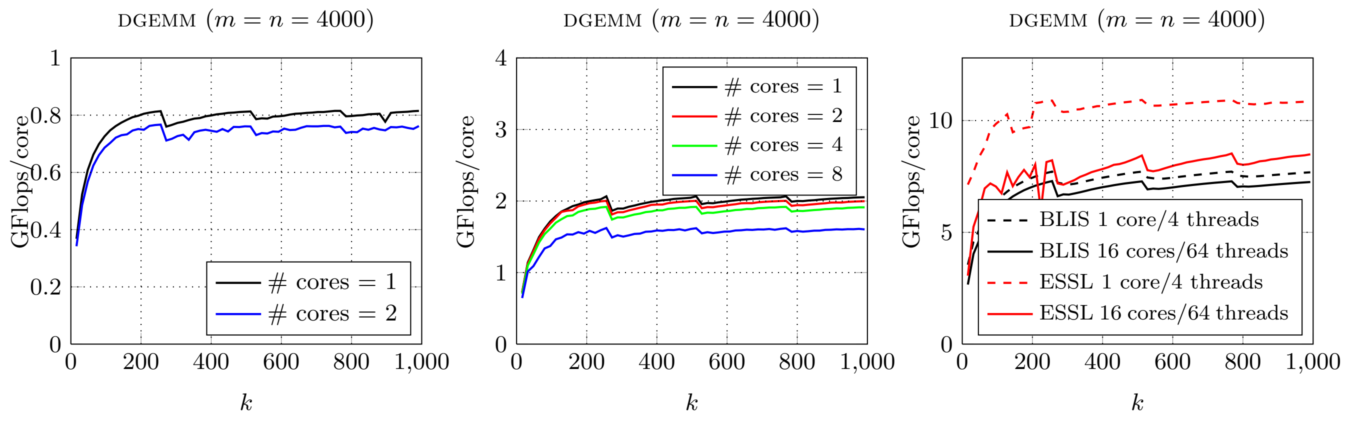


Figure 13: Parallel performance. Left: ARM Cortex A-9. Middle: Texas Instruments C6678 DSP. (Performance for a single core drops from Figure 10 since TI's OpenMP implementation reduces the available L2 cache size to 128 Kbytes.) Right: IBM Blue Gene/Q PowerPC A2.

work on multiplying the block $\tilde{A}$ times a $k_c \times n_r$ panel of $\tilde{B}$ in a round-robin fashion. The benefit of this approach is that the granularity of the computation is small.

We choose to report how fast performance ramps up when $m = n = 4000$ and k is varied. As discussed before, being able to attain high performance for small k is important for algorithms that cast most computation in terms of a rank-k update, as is the case for many algorithms incorporated in LAPACK [2] and `libflame` [24].

For architectures with a moderate number of cores, this approach turns out to be remarkably effective, as illustrated in Figures 12-13. In each of the graphs, we show the performance attained when using one thread per core, for as many cores as are available on the target architecture, and we scale the graphs so that the top coincides with the theoretical peak. Since we report GFLOPS/core, we expect performance (per core) to drop slightly as more cores are utilized.

The IBM Blue Gene/Q PowerPC A2 architecture is designed to execute four hardware threads per core for a total of 64 threads for the 16 cores. In our initial experiments, parallelism for this architecture was again attained by parallelizing the second loop around the micro-kernel and the packing routines. Preliminary results with 64 threads on 16 cores are reported in Figure 13 (right).

Our experiment of porting BLIS to the Intel Xeon Phi, which is designed to execute up to 240 threads on 60 cores, allowed us to evaluate whether BLIS will have a role to play as multicore becomes many-core. For this architecture, we parallelized the second loop around the micro-kernel in order to utilize four threads per core *and* the first loop around the inner kernel (the third loop around the micro-kernel) in order to increase the granularity of computation when utilizing the 60 cores.

Consirably more effort was dedicated to the port to the Intel Xeon Phi. Some relatively minor but important

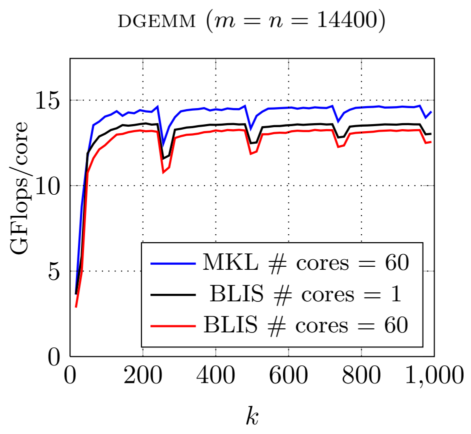


Figure 14: Parallel performance on the Intel Xeon Phi. The peak observed performance for BLIS with 60 cores (240 threads) was 796 GFlops (76% of peak).

changes were made to BLIS in order to help hide the considerable latency to memory on this architecture. The effort expended on the other architectures was minimal by comparison. Still, only OpenMP `pragma` directives were needed to achieve the reported parallelism. The resulting performance is reported in Figure 14. We see our multi-threaded BLIS implementation scales almost linearly to all 60 cores and is only within 10% from highly tuned Intel MKL BLAS performance, which is also shown in the same figure.

We did not similarly introduce parallelism into the loops around the micro-kernel for the other level-3 operations. However, we do not anticipate any technical difficulties in doing so.

What we learn from the experiments with multithreading is that the BLIS framework appears to naturally support parallelization on such architectures via OpenMP.

6 Library footprint

A concern with BLAS implementations can be their footprint. For example, on embedded systems the size of the executable can be a problem. BLIS is highly layered and hence very compact. This is in sharp contrast with, for example, ATLAS, which is autogenerated. As an example, on one of the tested architectures, a simple driver routine that only called DGEMM compiled to an executable of size around 107 Kbytes when linked to OpenBLAS 0.1.1, 180 Kbytes when linked to BLIS, and 2.38 Mbytes when linked to ATLAS 3.9.74. (When the call to DGEMM was commented out, the executable was 82 Kbytes in size.)

Thus, BLIS may be a better choice than ATLAS when the footprint of the source code is an issue. This insight should be tempered by the fact that all implementations require substantial work buffers.

7 Conclusion, contributions and future directions

The purpose of this paper was to evaluate the portability of the BLIS framework. Focusing on level-3 BLAS, one way to view BLIS is that it pushes the observations that all level-3 BLAS can be implemented in terms of matrix-matrix multiplication [16] to the limit. At the bottom of the food chain is now the micro-kernel, which implements a matrix-matrix multiplication with what we believe are, from a practical viewpoint, the smallest submatrices. We believe that the presented experiments allow us to be cautiously optimistic that BLIS will provide a highly maintainable and competitive open source software solution.

The results are preliminary. The BLIS infrastructure seems to deliver as advertised for the studied architectures for single threaded execution. For that case, implementing high-performance micro-kernels (one per floating-point datatype) brings all level-3 BLAS functionality online, achieving performance consistent with that of GEMM. We pushed beyond this by examining how easily BLIS will support multithreaded parallelism. The performance experiments show impressive speedup as the number of cores is increased, even for architectures with a very large number of cores (by current standards).

A valid question is, how much effort did we put forth to realize our results. On some architectures, only a few hours were invested in the port. On other architectures, those same hours yielded decent performance, but considerably more was invested in the micro-kernel to achieve performance more closely rivaling that of vendors and/or the OpenBLAS. What we do know is that the experts involved (who were not part of our FLAME project and who were asked to try BLIS) enthusiastically embraced the challenge, and we detected no reduction in that enthusiasm as they became more familiar with BLIS.

The BLIS framework opens up a myriad of new research and development possibilities. Should automatic fine-tuning of parameters be incorporated? Will the framework port to GPUs? (We believe it will.) How easily can funtionality be added? Will it enable new research, such as how to add algorithmic fault-tolerance [10, 13] to BLAS-like libraries? Will it be useful when DLA operations are used to evaluate future architectures with simulators (where auto-tuning may be infeasible due to time constraints)? It is our hope to investigate these and other questions in the future.

Availability

BLIS is available under the "new" ("3-clause") BSD license at `http://code.google.com/p/blis/`. The version used in this paper, including micro-kernels and test drivers, will be made available so that others can verify results and examine performance for other matrix sizes.

Acknowledgments

We thank the rest of the FLAME team for their support. This work was partially sponsored by the NSF (Grant OCI-1148125), Spanish Projects CICYT-TIN 2008/508 and TIN 2012-32180, the National High-tech R&D Program of China (Grant 2012AA010903), and the National Natural Science Foundation of China (Grant 61272136).

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

We thank TACC for granting access to the Stampede cluster, AMD and Texas Instruments for the donation of equipment used in our experiments, and Ted Barragy and Tim Mattson for their encouragement.

References

[1] M. Ali, E. Stotzer, F. D. Igual, and R. A. van de Geijn. Level-3 BLAS on the TI C6678 multi-core DSP. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 179 –186, oct. 2012.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] `http://www.vesperix.com/arm/atlas-arm/index.html`, 2013.

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

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

[6] Freescale Semiconductor. AltiVec Technology Programming Interface Manual. Available at `http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf`, 1999.

[7] K. Goto and R. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12:1–12:25, May 2008.

[8] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft.*, 35(1):1–14, July 2008.

[9] M. Gschwind. Blue Gene/Q: design for sustained multi-petaflop computing. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 245–246, New York, NY, USA, 2012. ACM.

[10] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[11] J. A. Gunnels, R. A. van de Geijn, D. S. Katz, and E. S. Quintana-Orti. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 47–56, Washington, DC, USA, 2001. IEEE Computer Society.

[12] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel(r) xeon phi(tm) coprocessor. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013)*, 2013. To appear.

[13] K. Huang and J. Abraham. Algorithm–based fault tolerance for matrix operations. *IEEE Trans. on Computers*, 33(6):518–528, 1984.

[14] IBM Blue Gene team. Design of the IBM Blue Gene/Q compute chip. *IBM Journal of Research and Development*, 57(1/2):1:1–1:13, 2013.

[15] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. A. van de Geijn. Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. In *SC'12. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, page 26:1–26:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press, IEEE Computer Society Press.

[16] B. Kågström, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[17] 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.

[18] Loongson Technology Corp. Ltd. *Loongson 3A processor manual*, 2009.

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

[20] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[21] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, 2011.

[22] TMS320C66x DSP CPU and Instruction Set Reference Guide. `http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf`, November 2010. Texas Instruments Literature Number: SPRUGH7.

[23] TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor. `http://www.ti.com/lit/ds/sprs691c/sprs691c.pdf`, February 2012. Texas Instruments Literature Number: SPRS691C.

[24] F. G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2012.

[25] F. G. Van Zee, E. Chan, R. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[26] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for generating BLAS-like libraries. FLAME Working Note #66. Technical Report UTCS TR-12-30, UT-Austin, November 2012.

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

[28] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[29] K. Yotov, X. Li, M. J. Garzarán, D. Padua, K. Pingali, and P. 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.