The BLIS Framework: Experiments in Portability

FIELD G. VAN ZEE, TYLER M. SMITH, BRYAN MARKER, TZE MENG LOW, and ROBERT A. VAN DE GEIJN, University of Texas at Austin
FRANCISCO D. IGUAL, Complutense University of Madrid
MIKHAIL SMELYANSKIY, Intel Corporation
XIANYI ZHANG, Chinese Academy of Sciences
MICHAEL KISTLER, VERNON AUSTEL, and JOHN A. GUNNELS, IBM Corporation
LEE KILLOUGH, Cray Inc.

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 many-core architectures. We show, with very little effort, how 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. Although most of this article focuses on single-core implementation, we also provide compelling results that suggest the framework's leverage extends to the multithreaded domain.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]—*Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Linear algebra, libraries, high performance, matrix, multiplication, BLAS

This work was partially sponsored by the National Science Foundation (NSF; grant ACI-1148125/1340293), 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 authors and do not necessarily reflect the views of the NSF. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.
Authors' addresses: F. G. Van Zee and R. A. van de Geijn, Department of Computer Science and Institute for Computational Engineering and Sciences, The University of Texas at Austin, 2317 Speedway, Stop D9500, Austin, TX 78712; emails: {field, rvdg}@cs.utexas.edu; T. M. Smith, B. Marker, and T. M. Low, Department of Computer Science, The University of Texas at Austin, 2317 Speedway, Stop D9500, Austin, TX 78712; emails: {tms, bamarker, ltm}@cs.utexas.edu; F. D. Igual, Departmento de Arquitectura de Computadores y Automatica, Universidad Complutense de Madrid, 28040, Madrid, Spain; email: figual@fdi.ucm.es; M. Smelyanskiy, Parallel Computing Lab, Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054; email: mikhail.smelyanskiy@intel.com; X. Zhang, Institute of Software and Graduate University, Chinese Academy of Sciences, Beijing, China 100190; email: xianyi@iscas.ac.cn; M. Kistler, IBM Corp, Mail Stop 903/3L008, 11501 Burnet Road, Austin, TX 78758; email: mkistler@us.ibm.com; V. Austel and J. A. Gunnels, IBM's Thomas J. Watson Research Center, 1101 Kitchawan Road, Yorktown Heights, NY 10598; emails: {austel, gunnels}@us.ibm.com; L. Killough, Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle, WA 98164; email: killough@leekillough.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 0098-3500/2016/06-ART12 $15.00
DOI: http://dx.doi.org/10.1145/2755561

ACM Reference Format:
Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. 2016. The BLIS framework: Experiments in portability. ACM Trans. Math. Softw. 42, 2, Article 12 (June 2016), 19 pages.
DOI: http://dx.doi.org/10.1145/2755561

1. INTRODUCTION

This article discusses early results for BLAS-like Library Instantiation Software (BLIS), a new framework for instantiating Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988, 1990] libraries. BLIS provides a novel infrastructure that refactors, modularizes, and expands existing BLAS implementations [Goto and van de Geijn 2008a, 2008b], thereby accelerating portability of dense linear algebra (DLA) solutions across the wide range of current and future architectures. An overview of the framework is given in Van Zee and van de Geijn [2015].

This companion article to Van Zee and van de Geijn [2015] 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 microkernel, in terms of which all level-3 functionality is expressed and implemented. Only this microkernel needs to be customized for a given architecture to achieve high performance; portable routines for packing and loops that block through matrices (to improve data locality) are provided as part of the framework. Aside from coding the microkernel, the developer needs only to choose appropriate cache block sizes to instantiate highly tuned DLA implementations on a given architecture, with virtually no additional work.[1]

A team of BLAS developers, most with no previous exposure to BLIS, was asked to evaluate the framework. They wrote only the microkernel for double-precision real general matrix multiplication (DGEMM) for architectures of their choosing. The architectures on which we report include general-purpose multicore processors (AMD A10, Intel® Xeon™ E3-1220 Sandy Bridge E3, and IBM Power7), low-power processors[2] (ARM Cortex-A9, Loongson 3A, and Texas Instruments C6678 DSP), and many-core architectures (IBM Blue Gene/Q PowerPC A2 and Intel Xeon Phi). We believe that this selection of architectures is illustrative of the main solutions available in the HPC arena, with the exception of GPUs (which we will investigate in the future). With moderate effort, not only were full implementations of DGEMM created for all of these architectures, but the implementations attained performance that rivals that of the best available BLAS libraries. Furthermore, the same microkernels, without modification, are shown to support high performance for the remaining level-3 BLAS.[3] Finally, by introducing simple OpenMP [OpenMP Architecture Review Board 2008] `pragma` directives, multithreaded parallelism was extracted from DGEMM.

[1] In the process of writing the microkernel, the developer also chooses a set of *register* block sizes, which subsequently do not change since their values are typically reflected in the degree of assembly-level loop unrolling within the microkernel implementation.
[2] Actually, these low-power processors also feature multiple cores, but for grouping purposes we have chosen to distinguish them from the other processors because of their relatively low power requirements.
[3] These other level-3 BLAS operations are currently supported within BLIS for single-threaded execution only. However, given that we report on parallelized DGEMM, multithreaded extensions for the remaining level-3 operations are, by our assessment, entirely feasible and planned for a future version of the framework.

2. WHY A NEW OPEN SOURCE LIBRARY?

Open source libraries for BLAS-like functionality provide obvious benefits to the computational science community. Although vendor libraries like ESSL from IBM, MKL from Intel, and ACML 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 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 [Huang and Abraham 1984] into BLAS libraries is one possible solution [Gunnels et al. 2001b]. Thus, a well-structured open source implementation would facilitate research that would otherwise be limited by a closed source library.

Prior to BLIS, there were two open source options for high-performance BLAS: ATLAS [Whaley and Dongarra 1998] and OpenBLAS [2012], the latter of which is a fork of the widely used GotoBLAS[4] [Goto and van de Geijn 2008a, 2008b] implementation. The problem is that neither is easy to maintain or modify, and we would argue that neither will easily facilitate such research.[5]

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 Yotov et al. [2005], it is shown that parameters can be determined analytically). Although BLIS also requires a kernel to be optimized, there are many important differences. BLIS mimics the algorithmic blocking performed in the GotoBLAS, and to our knowledge, the GotoBLAS outperforms ATLAS on nearly all architectures, often by a wide margin. This out-performance is rooted in the fact that GotoBLAS and BLIS implement a fundamentally superior approach—one that is based on theoretical insights [Gunnels et al. 2001a]. Another key difference is that we believe BLIS is 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, in which case autotuning is not possible because the build system (the system that generates the object code) differs from the host system (the system that executes the object code). ATLAS is also impractical for situations where execution occurs within a virtual (rather than physical) machine, such as when simulators are used to design future processors.

BLIS mimics the algorithms that Goto developed for the GotoBLAS (and thus OpenBLAS); however, we consider his implementations difficult to understand, maintain, and extend. Thus, they will be harder to adapt to future architectures and more cumbersome when pursuing new research directions. In addition, as we will discuss later, BLIS casts Goto's so-called inner kernel in terms of a smaller microkernel 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—opportunities that were previously unavailable.

[4]The original GotoBLAS software is no longer supported by its author, Kazushige Goto.
[5]We acknowledge that in this article we present no evidence that BLIS is any easier to decipher, maintain, or extend. The reader will have to investigate the source code itself.

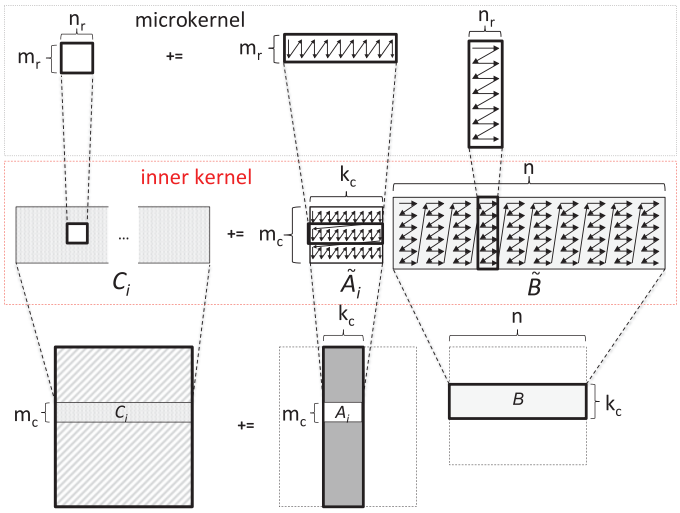

Fig. 1. Illustration from Van Zee and van de Geijn [2012] of the various levels of blocking and related packing when implementing GEMM in the style of Goto and van de Geijn [2008a]. The bottom layer shows the general GEMM and exposes the special case where $k = k_c$ (known as a rank-k update, with $k = k_c$). The top layer shows the microkernel 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 macrokernel. (This middle layer corresponds to the "inner kernel" in the GotoBLAS.) Similarly, m_r and n_r serve as register block sizes for the microkernel 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.

3. A LAYERED IMPLEMENTATION

In many ways, the BLIS framework is a reimplementation of the GotoBLAS software that increases code reuse via careful layering. We now describe how the GotoBLAS approach layers the implementation of matrix-matrix multiplication. Then, we will discuss ways in which BLIS employs this approach as well as how BLIS differs.

The Goto 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 [Goto and van de Geijn 2008a], 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.

To implement this special case ($k = k_c$), matrix C is partitioned into row panels, C_i, that are $m_c \times n$, whereas A is partitioned into $m_c \times k_c$ blocks, A_i. Thus, the problem reduces to subproblems of the form of $C_i := A_i B + C_i$. 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 an "inner kernel," which an expert codes in assembly language for a specific architecture. In this 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 refactors the inner kernel into a double loop over what we call the *microkernel*. The outer of these two loops was described earlier: 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, which now reside within the refactored macrokernel, and all loops required to block down to this point are coded portably in C99. It is then the multiplication of the current row panel of $\tilde{A}_i$ times the current column panel of $\tilde{B}$ that updates the corresponding $m_r \times n_r$ 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 microkernel, which is illustrated in the top layer of Figure 1. It is only this microkernel, which contains a single loop over the k dimension (usually limited to k_c), that needs to be highly optimized for a new architecture. Notice that this microkernel is implicitly present within the inner kernel of GotoBLAS. However, a key difference is that BLIS exposes the microkernel explicitly as the only routine that needs to be highly optimized for high-performance level-3 functionality. All loops implementing layers above the microkernel are written in C99 and therefore are fully portable to other architectures. This microkernel is also optimized in hardware as a linear algebra accelerator core [Pedram et al. 2012a, 2012b].

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 and mappings to memory beyond column- or row-major storage, in any combination. Other level-3 operations introduce further dimensions of variation, such as lower/upper storage for symmetric, Hermitian, and triangular matrices, as well as multiplication of such matrices from the left or right. The framework handles this burgeoning space of possible cases in part by exploiting the fact that submatrices of A and B must always be packed to facilitate high performance. BLIS uses this opportunity to intelligently cast an arbitrary special case into a common "base case" for which a high-performance implementation is provided. Further flexibility and simplification is achieved by specifying both row and column strides for each matrix (rather than a single "leading dimension"). Details can be found in Van Zee and van de Geijn [2012].

Other matrix-matrix operations. A key feature of the layering of BLIS in terms of the microkernel is that it makes the development of other matrix-matrix (level-BLAS 3) operations simple. In Kågström et al. [1998], 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 [Goto and van de Geijn 2008b]. (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 practical limit: the BLIS microkernel.

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 plain GEMM kernel cannot be used when updating panels of matrix C that intersect the diagonal, because such a kernel would illegally update parts of the strictly 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

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	–	8×3	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
TI C6678	1	1	8	4	4	32	32	512	4096	4×4	128×256	4096
IBM BG/Q A2	1.6	8	16	1[ii]	12.8	204.8	16	32K	–	8×8	1008×2016	4096
Intel Xeon Phi	1.09	16	60	1	17.44	1046.4	32	512	–	30×8	120×240	–

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

Fig. 2. Architecture summary.

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 innermost loop corresponding to the microkernel. 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 (not to mention enhance or extend) the implementation. By contrast, since BLIS already compartmentalizes all architecture-sensitive code within the microkernel, the framework naturally allows us to provide generic and portable instances of these specialized inner kernels (which we call *macrokernels*), each of which builds upon the same microkernel in a slightly different way. Thus, BLIS simultaneously reduces the *size* of the assembly kernel required as well as the *number* of kernels required. This also has the side effect of improving the performance of the hardware instruction cache.

4. TARGETING A SINGLE CORE

The first BLIS work [Van Zee and van de Geijn 2012] 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-threaded performance. Multithreaded, multicore performance is discussed in the next section.

For all but two architectures, *only* the microkernel (which 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 each chosen based on the target architecture; we refer the reader to Low et al. [2014], in which we demonstrate that an analytical study suffices to obtain optimal values for a specific architecture, taking into account exclusively cache-related parameters. Figure 2 summarizes the main BLIS parameters used. On two architectures, the Blue Gene/Q PowerPC A2 and Intel Xeon Phi, a more extensive implementation was attempted to examine modifications that may be required to support many-core architectures.

The performance experiments examined the following representative set of double-precision level-3 operations: DGEMM ($C := AB + C$); DSYMM ($C := AB + C$, where A is stored in lower triangle); DSYRK ($C := AA^T + C$, where C is stored in lower triangle) and DSYR2K ($C := AB^T + BA^T + C$, where C is stored in lower triangle); and DTRMM ($C := AB$, where A is lower triangular) and DTRSM ($C := A^{-1}B$, where A is lower triangular). In all cases, the matrices were stored in column-major order.

Architecture	Compiler (version)	Compiler optimizations and architecture-specific flags	Microkernel implementation
AMD A10 5800K	gcc (4.8)	`-O3 -mavx -mfma -march=native`	C code + inline assembly code
Sandy Bridge E3	gcc (4.8)	`-O3 -mavx -mfpmath=sse`	C code + inline assembly code
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
TI C6678	cl6x (7.4.1)	`-O2 -mv6600 --abi=eabi`	C 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

Fig. 3. Summary of compiler and microkernel implementation details.

The following architecture-specific sections describe details germane to the instantiation of BLIS on that particular system. 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. 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 and the vendor library, ATLAS, and/or OpenBLAS. In each set of graphs, the top graph reports the performance of the different DGEMM implementations, the middle graph reports the performance of BLIS for various level-3 BLAS operations, and the bottom graph reports 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 top and middle graphs, the uppermost point along the y-axis represents the peak performance of the architecture. The top graph reports how quickly the performance of DGEMM ramps up when $m = n = 1000$ and the k dimension is varied in increments of 32. This matrix shape is important because a relatively large matrix-matrix multiply with a relatively small k (known as a rank-k update) is often at the heart of important higher-level operations that have been cast in terms of DGEMM, such as the Cholesky, LU, and QR factorizations [Anderson et al. 1999; Van Zee et al. 2009]. Ideally, the implementation performance should ramp up quickly to its asymptote as k increases. The middle and bottom graphs report performance using square matrices (also in increments of 32) up to problem sizes of 2,000.

4.1. General-Purpose Architectures

The *AMD A10* processor implements the Trinity microarchitecture, an accelerated processing unit (APU) that combines a small number of CPU cores (between two and four) and a large number of GPU cores. The chosen processor (A10 5800K) incorporates four CPU cores and 384 GPU cores. Our implementation targets the 64-bit, x86-based CPU 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).

The microkernel 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 microkernel. The Trinity microarchitecture supports two formats of fused multiply-add (MADD) instructions: FMA3 and FMA4. We found that the FMA3 instructions facilitated higher performance, and so we employed FMA3 in our microkernel.

Initial performance is reported in Figure 4. We compare against ACML 6.1.0 (with FMA3 instructions enabled) and ATLAS. Our DGEMM implementation matches and, especially for small values of k, improves the performance of the proprietary ACML library. Furthermore, it clearly outperforms the latest stable version of ATLAS. Although the improvement in performance of our BLIS DGEMM is decent, the performance benefits for the rest of the level-3 BLAS routines are more evident, attaining improvements ranging between 25% and 50%.

The *Sandy Bridge E3* processor is a 64-bit, x86-based superscalar, out-of-order microarchitecture. 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 FPUs. Vector integer and FPUs support multiply, add, register shuffling, and blending operations on up to 256-bit registers.

The microkernel developed for the Sandy Bridge E3 processor was written by means of extended inline x86 assembly code, using AVX vector instructions and the GNU toolchain. Common optimization techniques like loop unrolling, instruction reordering, and cache prefetching were incorporated in the microkernel.

Performance results for Sandy Bridge E3 are reported in Figure 4. The top graph shows that the BLIS DGEMM implementation clearly outperforms that of ATLAS for rank-k update. Intel MKL yields the best performance, exceeding that of BLIS by roughly 5% for large problem sizes and attaining similar relative performance for small values of k. OpenBLAS DGEMM yields an intermediate performance and outperforms BLIS by a small margin for values of $k > 256$. The performance for the remaining level-3 BLAS routines provided by BLIS is comparable to that of its DGEMM and highly competitive with the proprietary implementation from Intel, especially for small problem sizes. On this architecture, the implementation of DTRSM can clearly benefit from implementing the optional TRSM microkernel supported by the BLIS framework [Van Zee and van de Geijn 2015].

The *IBM Power7* processor is an eight-core server processor designed for both commercial and scientific applications [Sinharoy et al. 2011]. 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, where each register can hold two double-precision floating-point values. Each Power7 thread has 64 architected VSX registers; however, 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 microkernel developed for Power7 was written almost entirely in plain C with AltiVec$^{\text{TM}}$ intrinsics for vector operations [Freescale Semiconductor 1999]. We compiled BLIS with the GNU C compiler provided by version 6.0 of the Advance Toolchain, an open source implementation that includes support for the latest features of IBM's Power architecture. The microkernel 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 3.864GHz Power7 processors. We configured ATLAS version 3.10.1 to use the architectural defaults for Power7 and built with the same Advance Toolchain compiler; for ESSL, we used version 5.2.0. All executions are performed on one core with large (16Mbyte) pages enabled.

Figure 5 presents BLIS performance on one core (executing one thread) of the IBM Power7 processor. BLIS DGEMM performance falls short of ESSL by about 10%; however, with further optimization, this gap could likely be narrowed considerably. The level-3 BLAS performance, presented in the middle and bottom graphs, shows that all of the level-3 operations except DTRSM achieve consistently high performance. TRSM

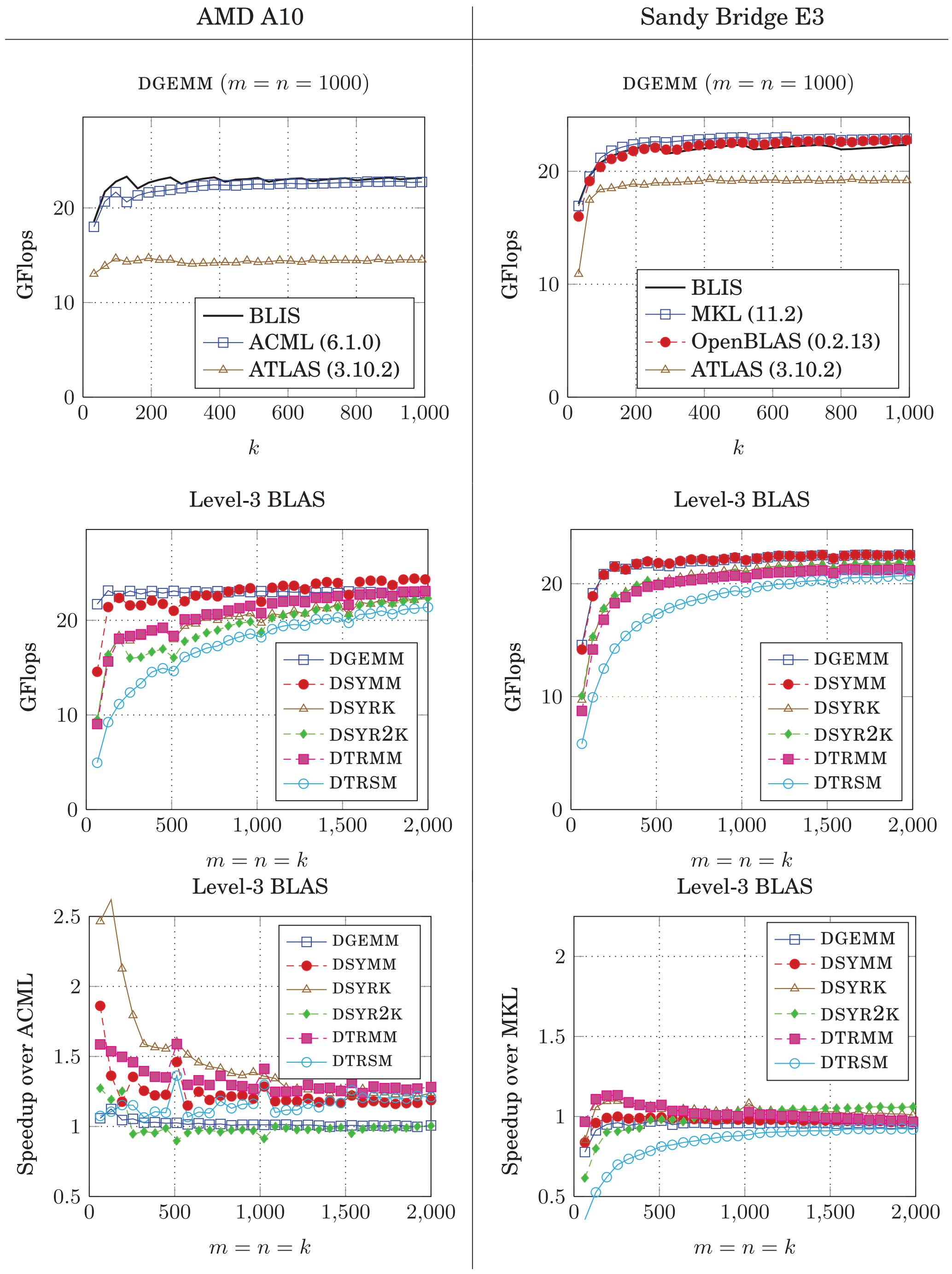

Fig. 4. Performance on one core of the AMD A10 (left) and the Sandy Bridge E3 (right) processors.

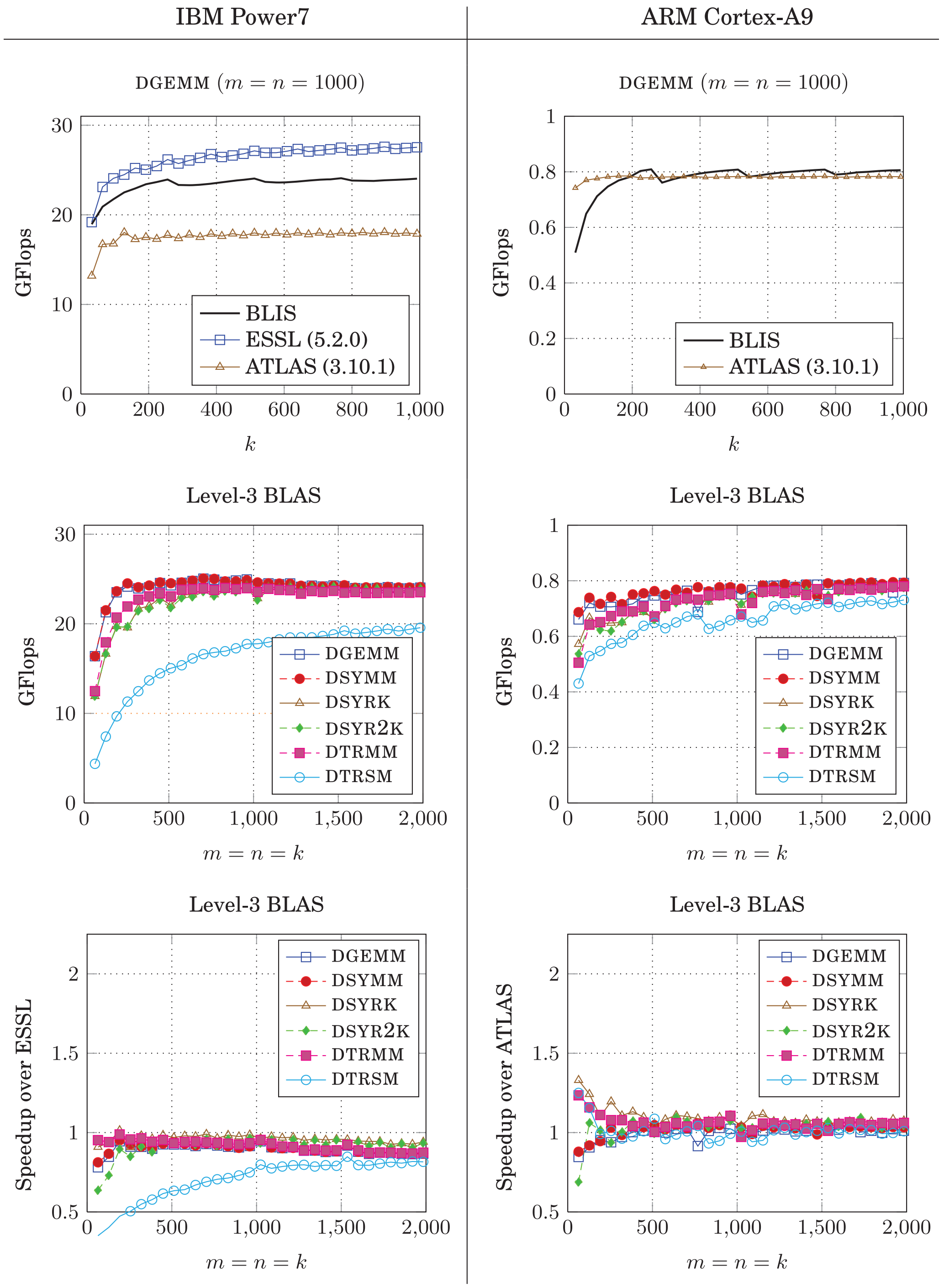

Fig. 5. Performance on one core of the IBM Power7 (left) and the ARM Cortex-A9 (right) processors.

performance lags that of the other level-3 operations because a significant portion of its computations do not employ the microkernel.

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 of utmost concern, 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 microkernel developed for the ARM Cortex-A9 processor was written exclusively using plain C code and the GNU toolchain. Given that NEON extensions for double-precision arithmetic do not exist, no vector intrinsics were used in the microkernel. Common optimization techniques like loop unrolling, a proper instruction reordering to hide memory latency, and cache prefetching are incorporated in the microkernel.

Performance is reported in Figure 5. As of this writing, the only tuned BLAS implementation for the ARM processor is ATLAS [2013], and so the top graph includes performance curves for only BLIS and ATLAS. 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, as shown in the top graph).

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 [Loongson Technology 2009]. Each core supports four-way superscalar and out-of-order execution, and includes five pipelined execution units, two arithmetic logic units, two FPUs, and one address generation unit. Every FPU is capable of executing single- and double-precision fused MADD instructions.

For this architecture, we optimized the BLIS DGEMM microkernel by coding exclusively in assembly code. Similar to the DGEMM optimization for the OpenBLAS [Xianyi et al. 2012], we adopted loop unrolling and instruction reordering, software prefetching, and the Loongson 3A-specific 128-bit memory accessing extension instructions to optimize the BLIS microkernel. We performed a limited search for the best block sizes m_c, k_c, and n_c.

Performance is reported in Figure 4. In the case of the Loongson 3A processor, only open source libraries are available. 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 block size parameters for BLIS to be slightly different from those used for the OpenBLAS, the performance of BLIS improved somewhat. Clearly, this calls for further investigation and performance tuning.

The *Texas Instruments C6678 DSP* incorporates the C66x DSP core from Texas Instruments [2012], 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 [Texas Instruments 2010]. The C66x instruction set includes SIMD instructions operating on 128-bit vector registers. Ideally, each core can perform up to eight single-precision MADD operations per cycle. In double precision, this number is reduced to two 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. In our experiments, we configured the on-chip memory as a classical L1-L2 cache hierarchy, delegating the memory movements to and from main memory to the cache subsystem.

Figure 6 reports BLIS performance compared to the only optimized BLAS implementation available as of today: the native TI BLAS implementation [Ali et al. 2012], 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 [Igual et al. 2012]. Although this support is on the BLIS road map, it is still not supported; hence, no DMA support is implemented in our macrokernel 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 microkernel the fundamental unit of computation and implements Goto's inner kernels as two loops around the microkernel, as illustrated in Figure 7.

A complete study of how parallelism can be introduced by parallelizing one or more of the loops is beyond the scope of this article and is studied in detail for some of the parallel architectures in Smith et al. [2014]. Instead, we simply used OpenMP [OpenMP Architecture Review Board 2008] `pragma` directives to parallelize the second loop around the microkernel, as well as the routines that pack a block of A and a row panel of B. Within the macrokernel, individual threads work on multiplying the $m_c \times k_c$ block $\tilde{A}$ by the current $k_c \times n_r$ panel of $\tilde{B}$, with the latter assigned to threads in a round-robin fashion. The benefit of this approach is that the granularity of the computation is quite small, which promotes load balance among threads.

We choose to report how fast performance ramps up when $m = n = 4{,}000$ 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 [Anderson et al. 1999] and `libflame` [Van Zee 2012].

For most of the architectures discussed in Section 4—that is, architectures with a moderate number of cores—this approach turns out to be remarkably effective, as illustrated in Figure 8. In each of the graphs, we show the performance attained when using one thread per core, employing as many cores as are available, and we scale the graphs so that the uppermost y-axis value coincides with the theoretical peak. Since we report GFLOPS/core, we expect performance (per core) to drop slightly as more cores are utilized.

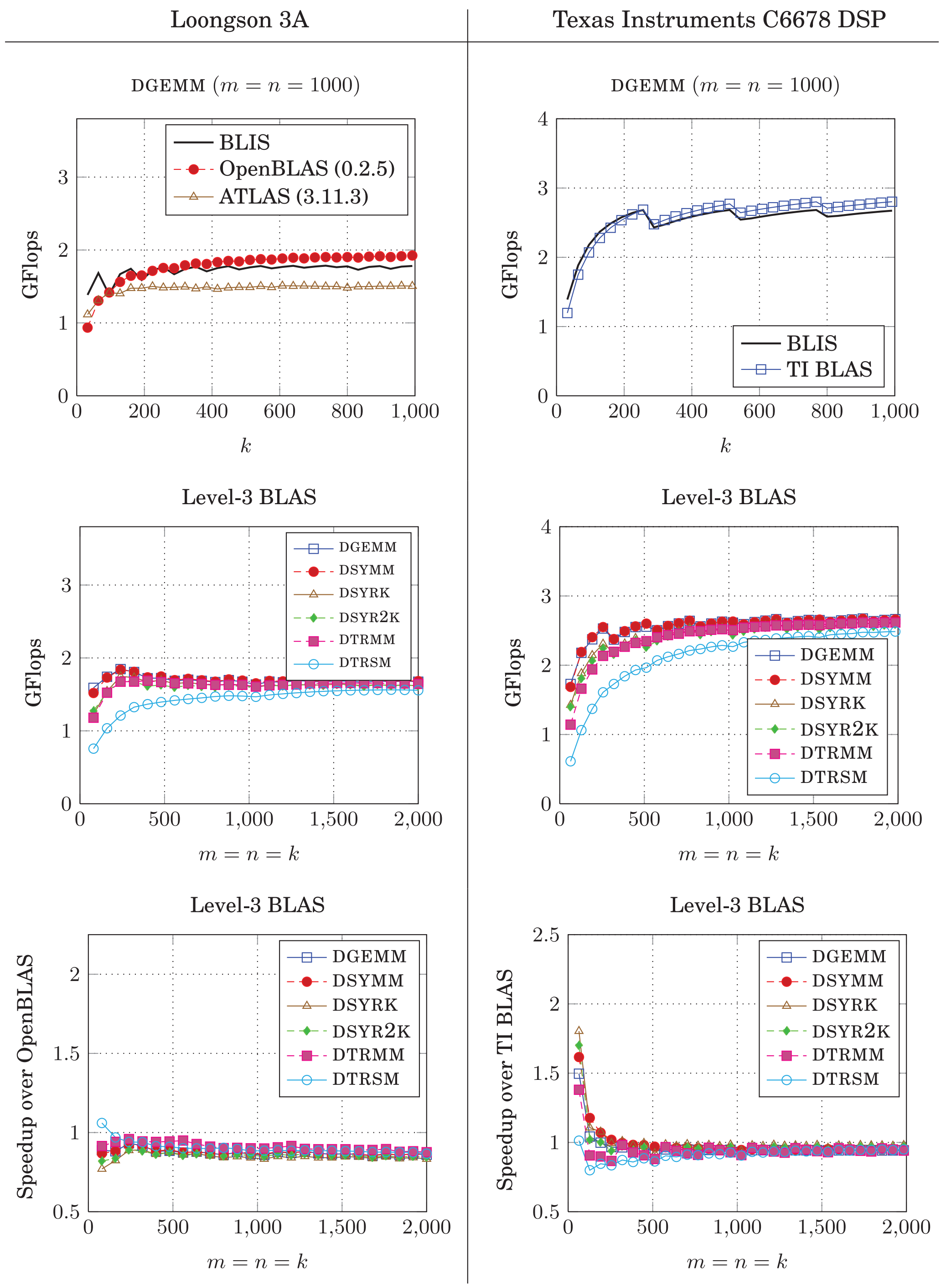

Fig. 6. Performance on one core of the Loongson 3A (left) and the Texas Instruments C6678 DSP (right) processors.

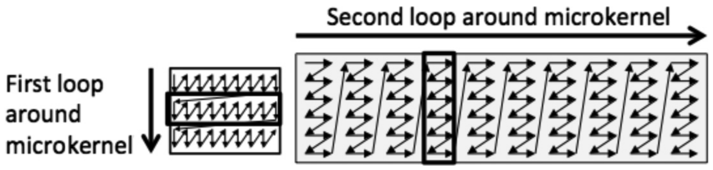

Fig. 7. Illustration of parallelism opportunities within the portion of the computation corresponding to the GotoBLAS inner kernel. The equivalent of that inner kernel is implemented in BLIS as a portable macro-kernel, written in C99, consisting of two loops around the microkernel. This exposes two extra opportunities for introducing loop parallelism.

5.1. Many-Core Architectures

We now examine the Blue Gene/Q PowerPC A2 and Intel Xeon Phi, two architectures that provide the potential for high levels of parallelism on a single chip.

The *IBM Blue Gene / Q PowerPC A2* processor is composed of 16 application cores, one operating system core, and one redundant (spare) core. All 18 of the 64-bit PowerPC A2 cores are identical and designed to be both reliable and energy-efficient [IBM Blue Gene Team 2013]. Mathematical acceleration for the kinds of kernels under study in this article is achieved through the use of the four-way double-precision quad processing extension (QPX) SIMD instructions that allow each core to execute up to eight floating-point operations in a single cycle [Gschwind 2012]. Efficiency stems from the use of multiple (up to four) symmetric hardware threads, each with their own register file. The use of multiple hardware threads enables dual-issue capabilities (i.e., a floating-point instruction and a load/store instruction may begin execution in a single cycle), alleviates pressures due to floating-point latency, and reduces the bandwidth required to maintain high performance. Other relevant details are catalogued in Figures 2 and 3.

The microkernel used for BLIS on Blue Gene/Q was written in C with QPX vector intrinsics.

Parallelism was achieved within a single core as well as across cores. On a single core, the four hardware threads were used to obtain two degrees of parallelism in both the m and n dimensions via the first and second loops around the microkernel, respectively. (This 2×2 thread parallelism was encoded into the microkernel itself.) Thus, each core multiplies a pair of adjacent row panels of $\tilde{A}$ by a pair of adjacent column panels of $\tilde{B}$ to update the corresponding four adjacent blocks of C. Additionally, when utilizing all 16 cores, we parallelized in the n dimension (once again within the second loop, but this time outside the microkernel) with a degree of 16, with round-robin assignment of data. Thus, each core iterates over the same block of $\tilde{A}$ while streaming in different pairs of column panels of $\tilde{B}$.

Figure 9 reports multithreaded performance on a single core as well as on 16 cores, with each core executing four hardware threads. We compare against the DGEMM implementation provided by IBM's ESSL library. Our single-core BLIS implementation outperforms that of ESSL for $k > 512$, with performance asymptoting at 2% or 3% above the vendor library for large values. On 16 cores, the difference is much more pronounced, with BLIS DGEMM outperforming that of ESSL by about 25% for most values of k tested. Remarkably, the ratio of performance per core between the 16 and single-core results is exceptionally high, indicating near-perfect scalability.

The *Intel Xeon Phi* co-processor (Knights Corner) is the first production co-processor in the Intel Xeon Phi product family. It features many in-order cores on a single die; each core has four-way hyperthreading support to help hide memory and multicycle instruction latency. To maximize area and power efficiency, these cores are less aggressive: they have lower single-threaded instruction throughput than CPU cores and run at a lower frequency. However, each core has 32 vector registers, each 512 bits wide, and

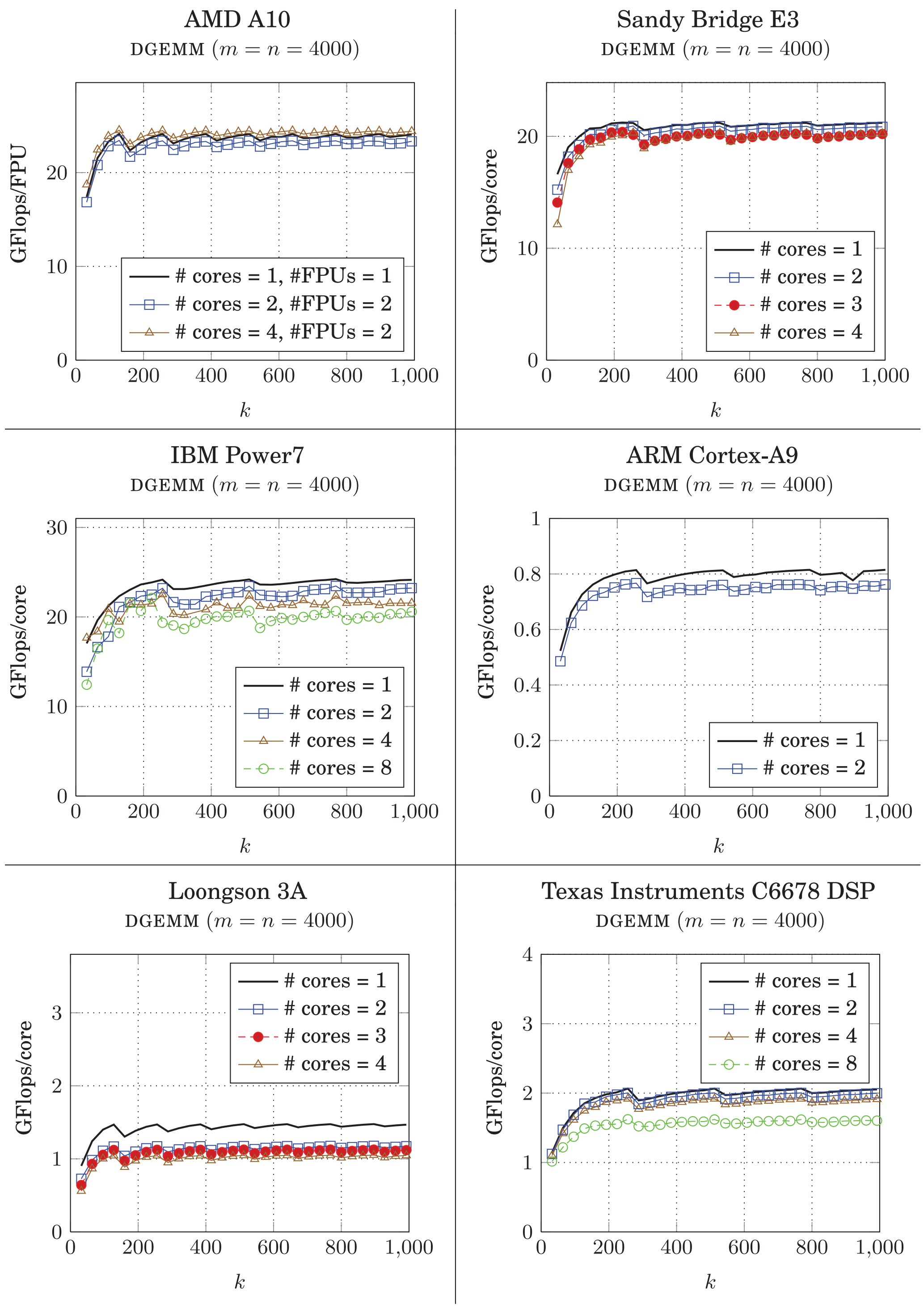

Fig. 8. Parallel performance. Performance of the Texas Instruments C6678 DSP on a single core drops from that shown in Figure 6 because Texas Instrument's OpenMP implementation reduces the available L2 cache size to 128 Kbytes.

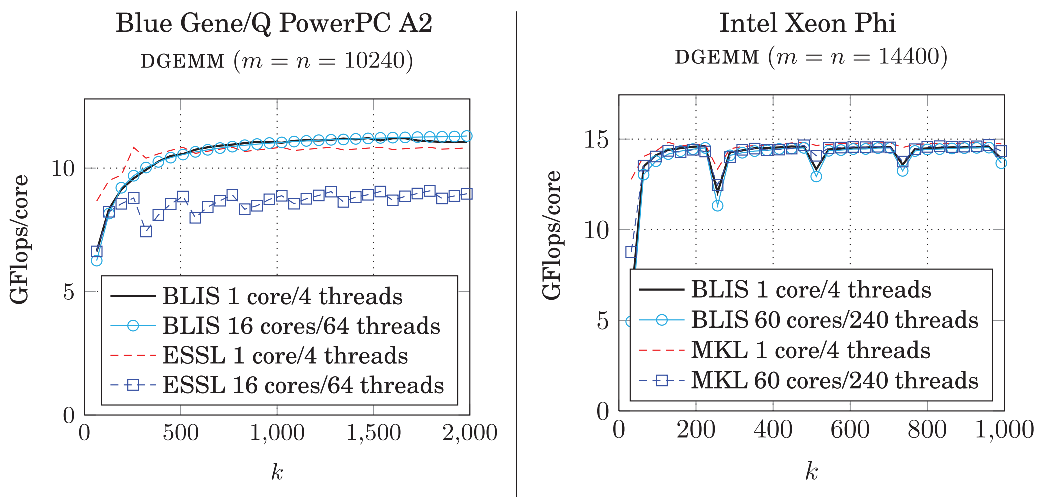

Fig. 9. Parallel performance on many-core architectures.

its vector unit can sustain 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 32KB L1 data cache and a larger 512KB L2 cache, 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 a Xeon Phi SE10P co-processor, which has 61 cores running at 1.1GHz, offering 17.1 GFLOPS of peak double-precision performance per core. It runs MPSS version 2.1.4346-16.

The microkernel was written in C using GNU extended inline assembly and incorporates many of the insights in Heinecke et al. [2013]. Both the $m_c \times k_c$ block of $\tilde{A}$ and the $k_c \times n_r$ panel of $\tilde{B}$ are streamed from the L2 cache. Because of this, and because prefetch instructions can be co-issued with floating-point operations, we aggressively prefetch $\tilde{A}$, $\tilde{B}$, and C in the microkernel.

Each thread can only issue instructions every other clock cycle, and 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 same block of $\tilde{A}$, so periodic thread synchronization is used to ensure data reuse of $\tilde{A}$ within their shared L1 cache. For this architecture, we parallelized the second loop around the microkernel to utilize four threads per core *and* the first loop around the macrokernel (i.e., the third loop around the microkernel) to increase the granularity of computation when utilizing the 60 cores.

Performance for DGEMM is reported in Figure 9. This graph includes results for execution of 240 threads on 60 cores for both BLIS and MKL version 11.0.1, as well as BLIS on a single core. Our multithreaded BLIS implementation of DGEMM performs on par with that of the highly tuned MKL library. In addition, when comparing results on a single core with those gathered on 60 cores, we see that BLIS facilitates impressive scalability.

Considerably more effort was dedicated to the port to the Blue Gene/Q and Intel Xeon Phi. Additionally, some relatively minor but important changes were made to BLIS to help hide the considerable latency to memory on the Intel Xeon Phi architecture. The effort expended on the other architectures was minimal by comparison, as a simple parallelization of the second loop around the microkernel was enough to attain excellent performance. We refer the reader to Smith et al. [2014] to observe the impact of different approaches toward parallel BLIS on multicore architectures. Still, only OpenMP `pragma` directives were needed to achieve the reported parallelism. In addition, Figure 9 shows that our multithreaded implementations for the Blue Gene/Q and Intel Xeon Phi scale almost linearly when all 16 and 60 cores, respectively, are utilized.

For these many-core architectures, we did not similarly introduce parallelism into the loops around the microkernel for the other level-3 operations. However, we do not anticipate any technical difficulties in doing so.

Our experiment of porting BLIS to the Blue Gene/Q PowerPC A2 and Intel Xeon Phi architectures allowed us to evaluate whether BLIS will have a role to play as multicore becomes many-core. What we learn from these experiments with multithreading is that the BLIS framework appears to naturally support parallelization on such architectures via OpenMP.[6]

6. CONCLUSION, CONTRIBUTIONS, AND FUTURE DIRECTIONS

The purpose of this article was to evaluate the portability of the BLIS framework. One way to view BLIS is that it starts from the observation that all level-3 BLAS can be implemented in terms of matrix-matrix multiplication [Kågström et al. 1998] and pushes this observation to its practical limit. At the bottom of the food chain is now the microkernel, which implements a matrix-matrix multiplication with what we believe are the smallest submatrices that still allow high performance. We believe that the presented experiments merit cautious optimism 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 microkernels (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. On other architectures, those same hours yielded decent performance, but considerably more was invested in the microkernel to achieve performance more closely rivaling that of vendors and/or the OpenBLAS. We can also report 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 myriad new research and development possibilities. Can high-performance microkernels be derived analytically from fundamental properties of the hardware? And/or, should automatic fine tuning of parameters be incorporated? Will the framework port to GPUs? How easily can functionality be added? Can the encoding of multithreading into the framework be generalized such that it supports arbitrary levels of (possibly asymmetric) parallelism? Will it enable new research, such as how to add algorithmic fault tolerance [Gunnels et al. 2001a; Huang and Abraham 1984] to BLAS-like libraries? Will it be useful when DLA operations are used to evaluate future architectures with simulators (where autotuning may be infeasible

[6]Supporting POSIX threads, although syntactically different from OpenMP, should also be feasible and relatively straightforward.

due to time constraints)? It is our hope to investigate these and other questions in the future.

Availability

The BLIS framework source code is available under the “new” (also known as the “modified” or “3-clause”) BSD license at http://github.com/flame/blis/.

ACKNOWLEDGMENTS

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

REFERENCES

Murtaza Ali, Eric Stotzer, Francisco D. Igual, and Robert A. van de Geijn. 2012. Level-3 BLAS on the TI C6678 multi-core DSP. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12)*. 179–186. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2012.26

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. 1999. *LAPACK Users' Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

ATLAS. 2013. ATLAS 3.8.4 ARM. Retrieved April 4, 2016, from http://www.vesperix.com/arm/atlas-arm/index.html.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 16, 1, 1–17.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 14, 1, 1–17.

Freescale Semiconductor. 1999. AltiVec Technology Programming Interface Manual. Retrieved April 4, 2016, from, http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

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

Kazushige Goto and Robert van de Geijn. 2008b. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software* 35, 1, 1–14.

Michael Gschwind. 2012. Blue Gene/Q: Design for sustained multi-petaflop computing. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 245–246. DOI:http://dx.doi.org/10.1145/2304576.2304609

John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. 2001a. A family of high-performance matrix multiplication algorithms. In *Computational Science—ICCS 2001*. Lecture Notes in Computer Science, Vol. 2073. Springer, 51–60.

John A. Gunnels, Robert A. van de Geijn, Daniel S. Katz, and Enrique S. Quintana-Orti. 2001b. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*. IEEE, Los Alamitos, CA, 47–56.

Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, George Chrysos, and Pradeep Dubey. 2013. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel® Xeon Phi™ coprocessor. In *Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS'13)*.

K. Huang and J. A. Abraham. 1984. Algorithm–based fault tolerance for matrix operations. *IEEE Transactions on Computers* 33, 6, 518–528.

IBM Blue Gene Team. 2013. Design of the IBM Blue Gene/Q compute chip. *IBM Journal of Research and Development* 57, 1/2, 1:1–1:13. DOI:http://dx.doi.org/10.1147/JRD.2012.2222991

Francisco D. Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert A. van de Geijn. 2012. Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. IEEE, Los Alamitos, CA, 26:1–26:11. http://dl.acm.org/citation.cfm?id=2388996.2389032

B. Kågström, P. Ling, and C. Van Loan. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software* 24, 3, 268–302.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 5, 3, 308–323.

Loongson Technology. 2009. *Loongson 3A Processor Manual*. Loongson Technology Corp. Ltd.

Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. 2014. *Analytical Modeling Is Enough for High Performance BLIS*. Technical Report. Department of Computer Sciences, University of Texas at Austin.

OpenBLAS. 2012. OpenBLAS Home Page. Retrieved April 4, 2016, from http://xianyi.github.com/OpenBLAS/.

OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. Retrieved April 4, 2016, from http://www.openmp.org/mp-documents/spec30.pdf.

Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn. 2012a. On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. In *Proceedings of the 2012 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12).*

Ardavan Pedram, Robert A. van de Geijn, and Andreas Gerstlauer. 2012b. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers* 61, 1724–1736. DOI:http://dx.doi.org/10.1109/TC.2012.132

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. 2011. IBM POWER7 multicore server processor. *IBM Journal of Research and Development* 55, 3, 1:1–1:29.

Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS'14)*.

Texas Instruments. 2010. TMS320C66x DSP CPU and Instruction Set Reference Guide. Retrieved April 4, 2016, from http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf.

Texas Instruments. 2012. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor. Retrieved April 4, 2016, from http://www.ti.com.cn/cn/lit/ds/symlink/tms320c6678.pdf.

Field G. Van Zee. 2012. `Libflame` *: The Complete Reference*. `www.lulu.com`.

Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering* 11, 6, 56–62.

Field G. Van Zee and Robert A. van de Geijn. 2012. *BLIS: A Framework for Generating BLAS-Like Libraries*. FLAME Working Note #66. Technical Report UTCS TR-12-30. Department of Computer Sciences, University of Texas at Austin.

Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Software* 41, 3, Article No. 14.

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC'98)*. 1–27.

Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS'12)*.

Kamen Yotov, Xiaoming Li, María Jesús Garzarán, David Padua, Keshav Pingali, and Paul Stodghill. 2005. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE* 93, 2, 358–386.

Received August 2013; revised February 2015; accepted April 2015