The BLIS Framework: Experiments in Portability

FIELD G. VAN ZEE, TYLER SMITH, BRYAN MARKER, TZE MENG LOW, ROBERT A. VAN DE GEIJN,
The University of Texas at Austin

FRANCISCO D. IGUAL,
Univ. Complutense de Madrid

MIKHAIL SMELYANSKIY,
Intel Corporation

XIANYI ZHANG,
Chinese Academy of Sciences

MICHAEL KISTLER, VERNON AUSTEL, 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 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.

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

General Terms: Algorithms, Performance

Authors' addresses: Field G. Van Zee and Robert A. van de Geijn, Department of Computer Science and Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712, {field,rvdg}@cs.utexas.edu. Tyler Smith, Bryan Marker, and Tze Meng Low, Department of Computer Science, The University of Texas at Austin, Austin, TX 78712, {tms,bamarker,ltm}@cs.utexas.edu. Francisco D. Igual, Departmento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, 28040, Madrid (Spain), figual@fdi.ucm.es. Mikhail Smelyanskiy, Parallel Computing Lab, Intel Corporation, Santa Clara, CA 95054, mikhail.smelyanskiy@intel.com. Xianyi Zhang, Institute of Software and Graduate University, Chinese Academy of Sciences, Beijing 100190 (China), xianyi@iscas.ac.cn. Michael Kistler, IBM's Austin Research Laboratory, Austin, TX 78758, mkistler@us.ibm.com. Vernon Austel and John Gunnels, IBM's Thomas J. Watson Research Center, Yorktown Heights, NY 10598, {austel,gunnels}@us.ibm.com. Lee Killough, Cray Inc., Seattle, WA 98164, killough@cray.com.

DOI:http://dx.doi.org/10.1145/0000000.0000000

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

ACM Reference Format:
ACM Trans. Math. Softw. 0, 0, Article 0 (0000), 21 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

This paper discusses early results for BLIS (BLAS-like Library Instantiation Software), a new framework for instantiating Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 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 futures architectures. An overview of the framework is given in [Van Zee and van de Geijn 2013].

This companion paper to [Van Zee and van de Geijn 2013] 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 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 micro-kernel 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 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 micro-kernels, 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 coding the micro-kernel, 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 micro-kernel 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. 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 re-targeted 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 related to algorithmic fault-tolerance.

Prior to BLIS, there were two open source options for high-performance BLAS: ATLAS [Whaley and Dongarra 1998] and OpenBLAS [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 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, and, to our knowledge, on nearly all architectures the GotoBLAS outperforms ATLAS, often by a wide margin. This out-performance is rooted in the fact that GotoBLAS and BLIS implement a fundamentally better approach—one which 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 auto-generator 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 auto-tuning is not possible because the build system (the system which generates the object code) differs from the host system (the system which 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.

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 so-called "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 code reuse via careful layering. We now describe how the GotoBLAS

[4]The original GotoBLAS software is no longer supported by its author, Kazushige Goto.

[5]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.

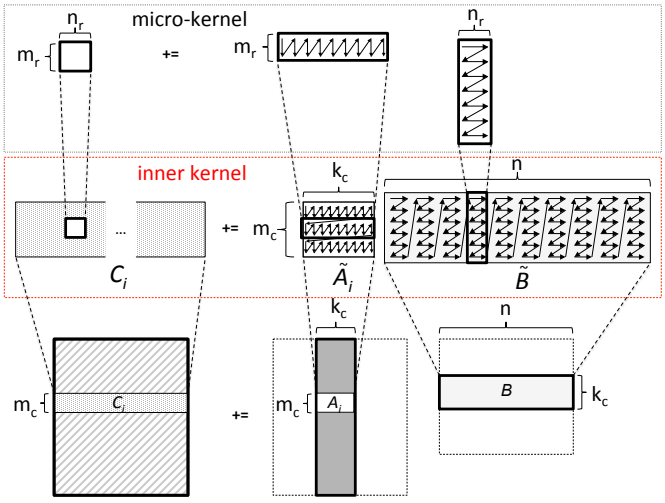

Fig. 1. Illustration from [Van Zee and van de Geijn 2012] of the various levels of blocking and related packing when implementing GEMM in the style of [Goto and van de Geijn 2008a]. The bottom layer shows the general GEMM and exposes the special case where $k = k_c$ (known as a rank-k update, with $k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

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.

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 [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 $m_c \times n$, while A (which is $m \times k_c$) 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 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 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 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 micro-kernel, which is illustrated in the top layer of Figure 1. It is only this micro-kernel, which contains 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. However, a key difference is that BLIS exposes the micro-kernel 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.

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. 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 micro-kernel is that it makes the porting of other matrix-matrix operations (level-3 BLAS) 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 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 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 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.

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
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.

Architecture	Compiler (version)	Compiler optimizations and architecture-specific flags	Micro-kernel implementation
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
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 micro-kernel implementation details.

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 (not to mention enhance or extend) 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"), each of which builds upon the same micro-kernel 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 instruction cache at run-time.

4. TARGETING A SINGLE CORE

The first BLIS paper [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 thread performance. Multithreaded, multicore performance is discussed in the next section.

For all but two architectures, *only* the micro-kernel (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 (though for space reasons, we omit discussion of *how* they were cho-

sen). 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 in order to examine modifications that may 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 ($C := AB+C$, where A is stored in lower triangle); DSYRK and DSYR2K ($C := AB^T + BA^T + C$, where C is stored in lower triangle); 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.

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 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 top graph reports the performance of the different DGEMM implementations, the middle graph the performance of BLIS for various level-3 BLAS operations, and the bottom 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 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]. Thus, the implementation should (ideally) ramp up quickly to its asymptote as k increases. The middle and bottom graphs report performance for square matrices, (also in increments of 32) up to problem sizes of 2000.

4.1. General-purpose architectures

The **AMD A10** processor implements the Trinity micro-architecture, an APU (Accelerated Processing Unit) that combines a reduced 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 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: FMA3 and FMA4. We found the FMA3 instructions to be the faster of the two, hence we used FMA3 in our micro-kernel.

Initial performance is reported in Figure 4. We compare against ACML 5.3.0 (with 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

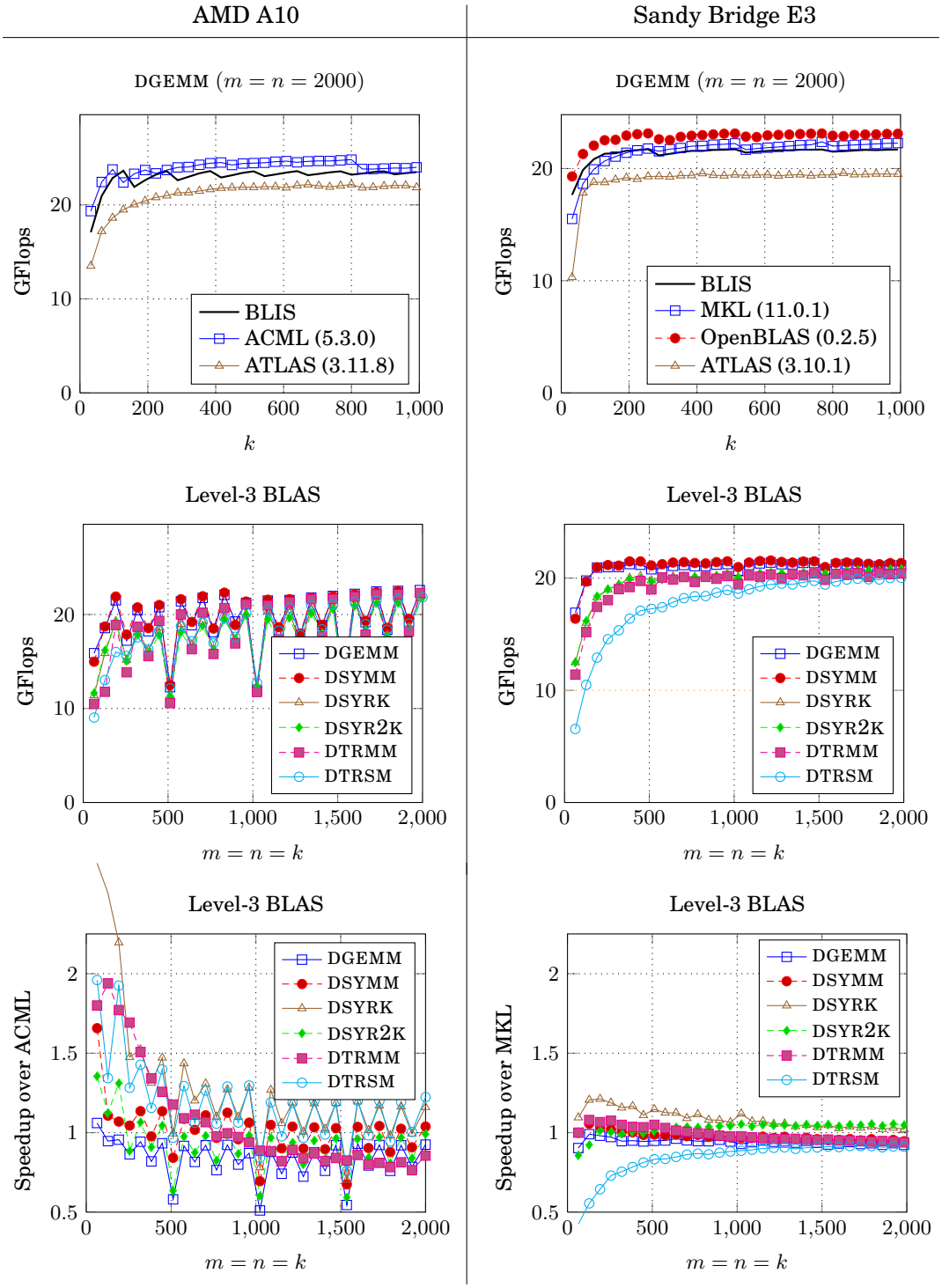

Fig. 4. Performance on one core of the AMD A10 (left) and the Sandy Bridge E3 (right) processors.

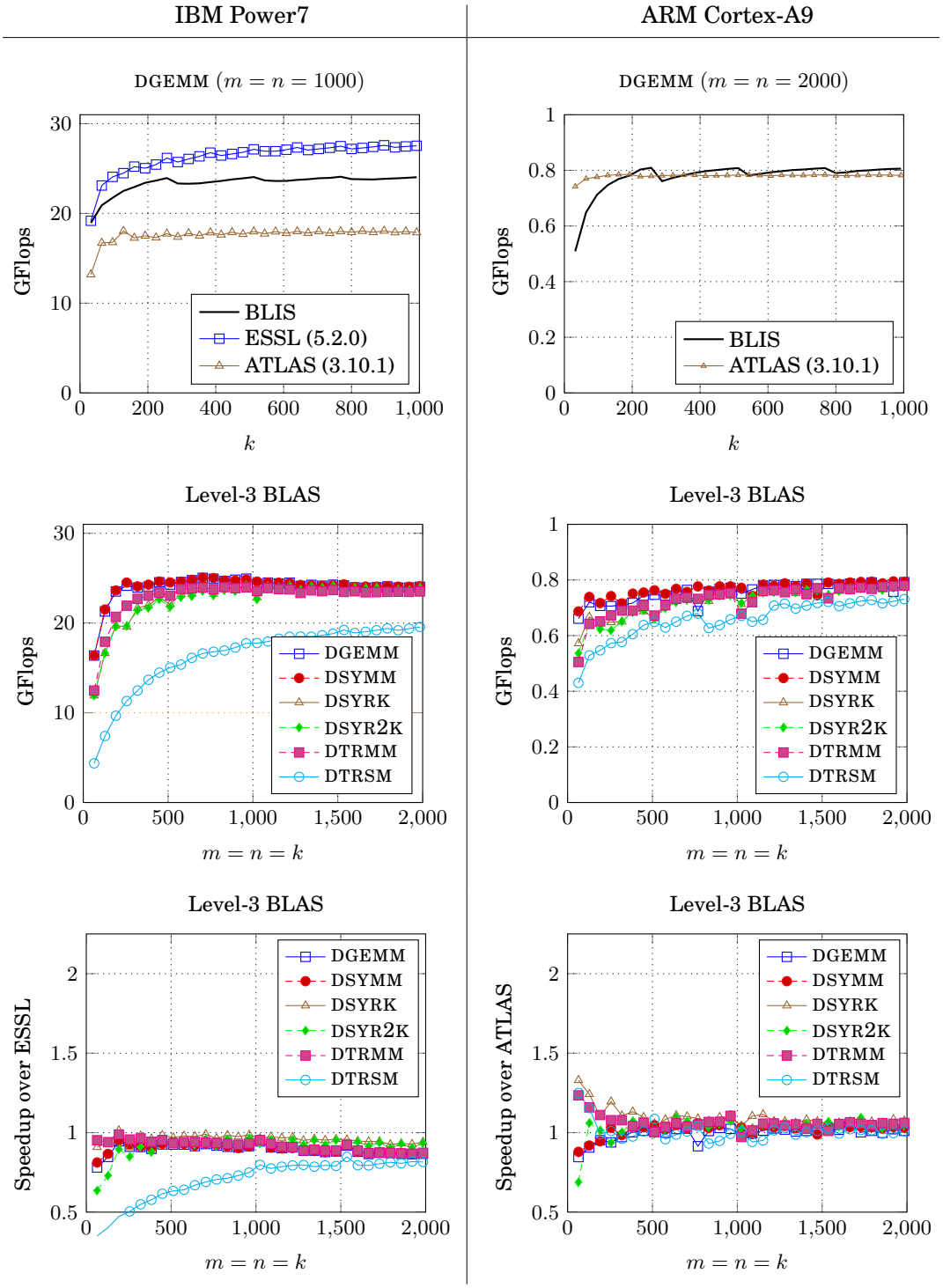

Fig. 5. Performance on one core of the IBM Power7 (left) and the ARM Cortex-A9 (right) processors.

problem sizes that are multiples of 128 can be fixed by adjusting the leading dimension of C. (We will investigate how to more generally fix this problem as part of our broader effort to improve 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-based 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 entirely in plain C, using only 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 4. The top graph shows that the BLIS DGEMM implementation clearly outperforms that of ATLAS for rank-k update. OpenBLAS yields the best performance, exceeding that of BLIS by roughly 10% for large problem sizes, and attaining similar relative performance for small values of k. Intel MKL DGEMM yields an intermediate performance and outperforms BLIS by a small margin for values of $k > 256$. (However, as shown in the bottom graph, MKL's DGEMM is superior for square problem sizes.) The performance for the remaining level-3 BLAS routines provided by BLIS is comparable with that of its DGEMM. On this architecture, the implementation of DTRSM can clearly benefit from implementing the optional TRSM micro-kernel supported by the BLIS framework [Van Zee and van de Geijn 2013].

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, 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 [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 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 same Advance Toolchain compiler. 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%, but 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 performance lags that of the other level-3 operations because a significant portion of its computations do not employ the micro-kernel.

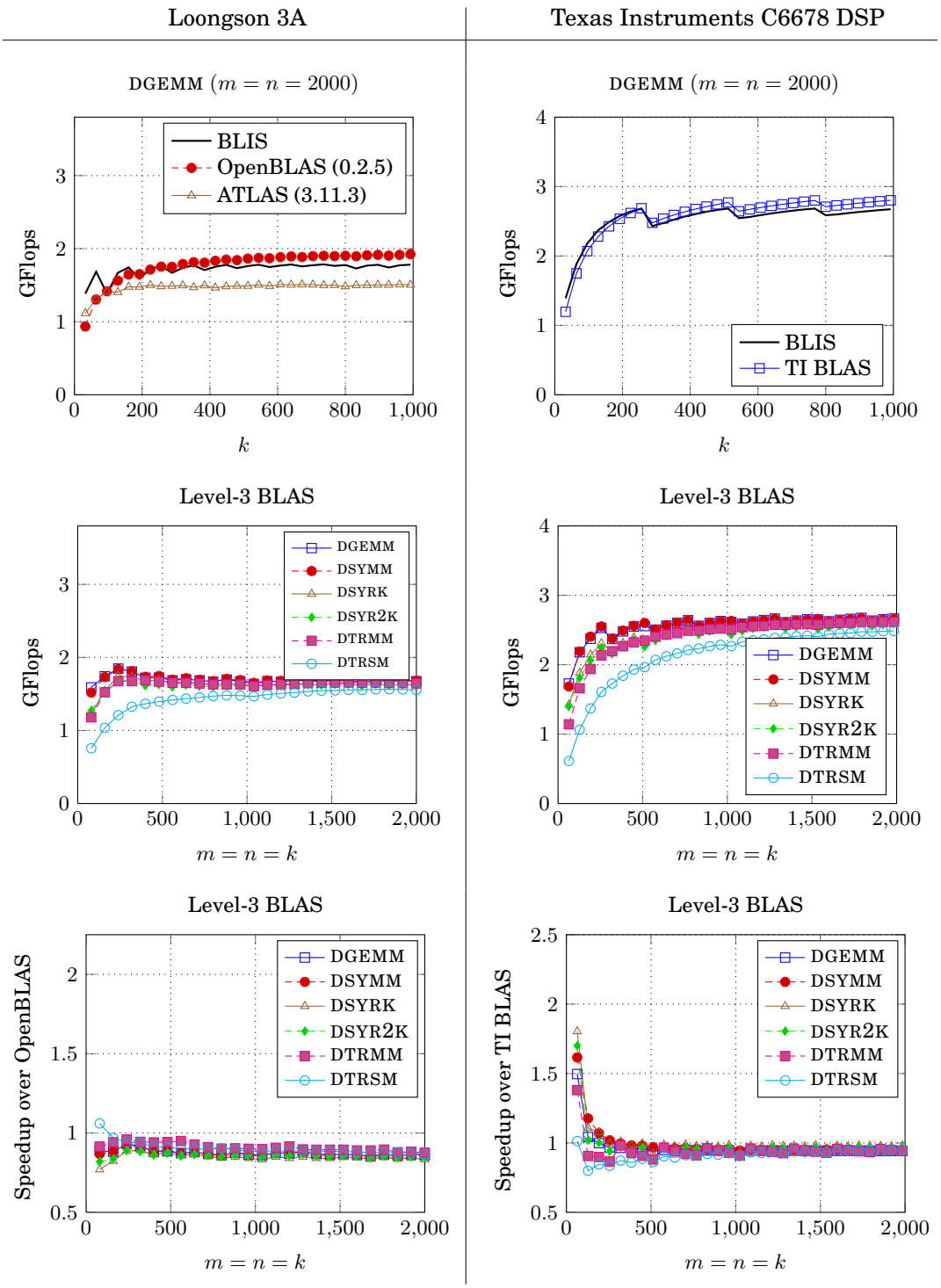

Fig. 6. Performance on one core of the Loongson 3A (left) and the Texas Instruments C6678 DSP (right) processors.

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. 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 5. As of this writing, the only tuned BLAS implementation for the ARM processor is ATLAS [ATLAS for ARM home page 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 Corp. Ltd 2009]. Each core supports four-way 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 [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 micro-kernel. We performed a limited search for the best blocksizes m_c, k_c, and n_c.

Performance is reported in Figure 6. 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 parameters for BLIS 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 [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 multiply-add (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.

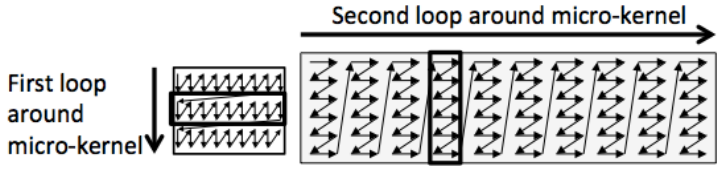

Fig. 7. Illustration of parallelism opportunities within the portion of the computation corresponding to the GotoBLAS "inner kernel." The equivalent of that inner kernel is implemented in BLIS as a portable macro-kernel, written in C, consisting of two loops around the micro-kernel. This exposes two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

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 6 reports BLIS performance compared with 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]. 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 8).

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 used OpenMP [OpenMP Architecture Review Board 2008] `pragma` directives to parallelize the second loop around the micro-kernel, as well as the routines that pack a block of A and a row panel of B. Within the macro-kernel, individual threads work on multiplying the block $\tilde{A}$ times a $k_c \times n_r$ 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.

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 [Anderson et al. 1999] and `libflame` [Van Zee 2012].

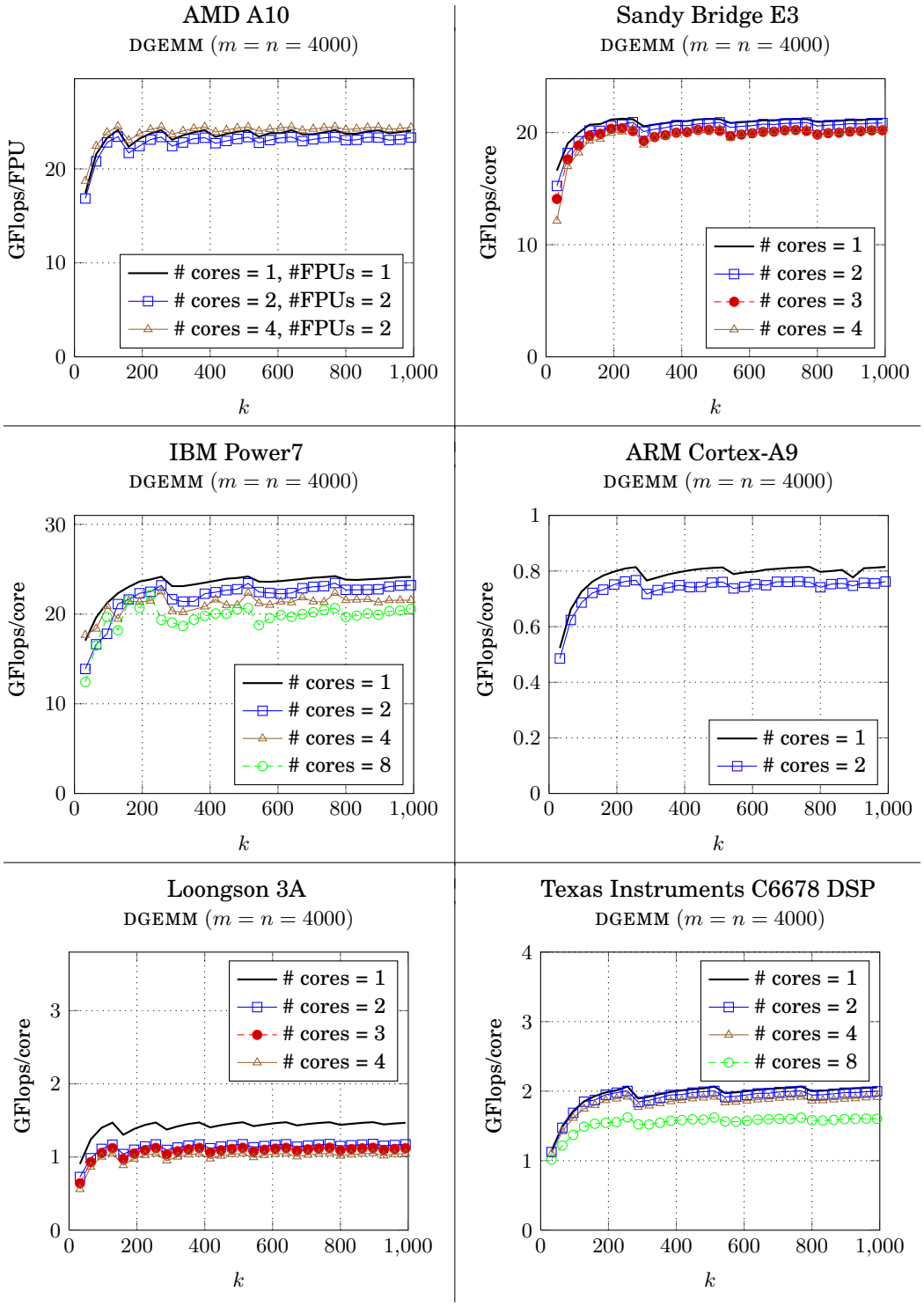

Fig. 8. Parallel performance. Performance of the Texas Instruments C6678 DSP on a single core drops from Figure 6 because TI's OpenMP implementation reduces the available L2 cache size to 128 Kbytes.

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.

5.1. Many-core architectures

We now examine the Blue Gene/Q PowerPC A2 and Intel Xeon Phi, two architectures which 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 paper 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. Multiple hardware 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 catalogued in Figures 2 and 3.

The micro-kernel used for BLIS on Blue Gene/Q was written in C with QPX vector intrinsics. 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 employed the use of a single node.

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—the first and second loops around the micro-kernel, respectively. (This 2×2 thread parallelism was encoded into the micro-kernel itself.) Thus, each core multiplies a pair of adjacent row panels of $\hat{A}$ by a pair of adjacent column panels of $\hat{B}$ to update the corresponding four adjacent blocks of C. Additionally, when utilizing all 16 cores, we parallelized in the n dimension with a degree of 16, with round-robin assignment of data. Thus, each core iterates over the same block of $\hat{A}$ while streaming in different pairs of column panels of $\hat{B}$.

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 $k > 256$, with performance asymptoting at 10-15% above the vendor library for large values. On 16 cores, the difference is even more pronounced, with BLIS DGEMM exceeding 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 quite high, indicating very good 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 hyper-threading support to help hide memory and multi-cycle 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 its vector unit can sustain 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

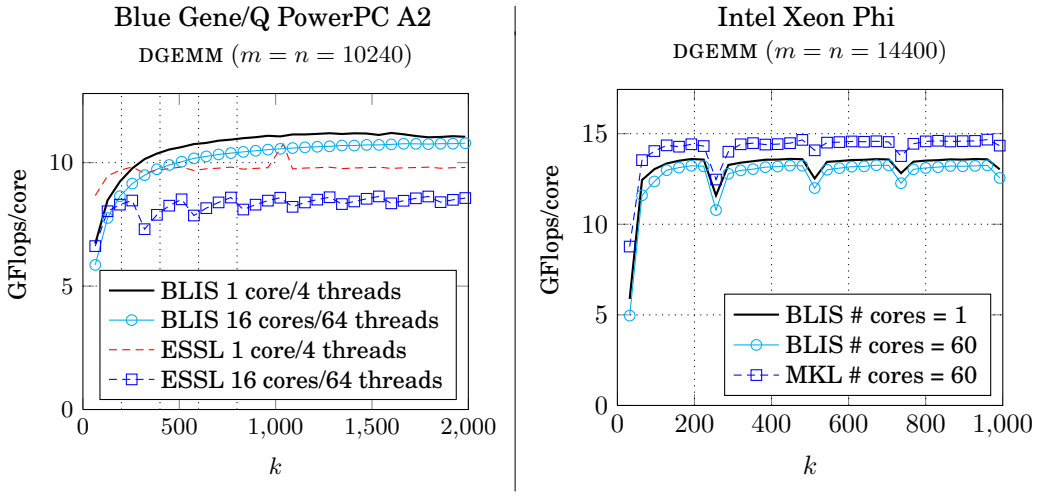

Fig. 9. Parallel performance on many-core architectures.

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 a 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.

The micro-kernel 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 $\hat{A}$ and the $k_c \times n_r$ 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 aggressively prefetch $\hat{A}$, $\hat{B}$, and C in the micro-kernel.

Each thread can only issue instructions every other clock cycle, 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 $\hat{A}$, so periodic thread synchronization is used to ensure data reuse of $\hat{A}$ within their shared L1 cache. 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.

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, as well as BLIS on a single core. While the raw performance of our initial multithreaded BLIS implementation falls short of the highly tuned MKL library (by about 15%), we once again see that, when moving from one to 60 cores, BLIS facilitates impressive scalability.

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.

Considerably more effort was dedicated to the port to the Blue Gene/Q and Intel Xeon Phi. Also, some relatively minor but important changes were made to BLIS in order to help hide the considerable latency to memory on the Intel Xeon Phi architec-

Object of interest	Byte footprint			
	BLIS	OpenBLAS	ATLAS	MKL
Executable that calls DGEMM	287K	32K	2.11M	2.80M
... and also DSYMM	292K	41K	2.13M	2.94M
... and also DSYRK	313K	56K	2.13M	3.16M
... and also DSYR2K	320K	72K	2.14M	3.22M
... and also DTRMM	364K	141K	2.17M	4.95M
... and also DTRSM	412K	210K	2.20M	5.57M
... and also ZGEMM	412K	253K	3.11M	7.98M
... and also ZHEMM	412K	262K	3.13M	8.03M
... and also ZHERK	412K	279K	3.15M	8.12M
... and also ZHER2K	412K	297K	3.16M	8.19M
... and also ZTRMM	412K	401K	3.20M	9.72M
... and also ZTRSM	412K	506K	3.25M	10.86M
Library archive	1.98M	6.18M	11.72M	???[6]
Total memory at run-time for executable that calls DGEMM ($m = n = k = 100$)	25.7M	43.1M	13.2M	13.9M
Total memory at run-time for executable that calls DGEMM ($m = n = k = 4000$)	391M	409M	415M	388M

Fig. 10. Various manifestations of library footprints when statically linked to each of: BLIS, OpenBLAS 0.2.6, ATLAS 3.10.1, and MKL 11.2. Here, "K" and "M" indicate 1024 and 1048576 bytes, respectively.

ture. The effort expended on the other architectures was minimal by comparison. Still, only OpenMP `pragma` directives were needed to achieve the reported parallelism. And 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 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 byte footprint. For example, on embedded systems where memory is particularly limited, a large executable can be problematic. BLIS is highly layered and designed to reuse code whenever possible, leaving it relatively compact in size. We provide evidence of this in Figure 10, which summarizes byte footprints of various executables and libraries on an Intel x86-64 architecture when statically linking to BLIS, OpenBLAS 0.2.6[7], ATLAS 3.10.1, and MKL 11.0 Update 4.

We can see that when linking BLIS to a simple test driver that calls only DGEMM, the resulting executable is 287Kbytes in size. Executables linked to OpenBLAS, ATLAS,

[6]MKL's BLAS are accessed by linking to three libraries, which together occupy 303Mbytes. However, these same libraries also provide other DLA functionality (such as LAPACK and ScaLAPACK) as well as signal processing functions (such as FFT). Thus, it would be difficult to estimate the size of only the object code needed by the BLAS.

[7]Here, OpenBLAS was configured as a sequential library, with CBLAS interfaces and built-in LAPACK functionality both disabled.

and MKL are 32Kbytes, 2.11Mbytes, and 2.80Mbytes, respectively. Thus, while BLIS does not yield the absolute smallest executable, it is still an order of magnitude smaller than a similar program linked to ATLAS or MKL.

The next observation we make is that adding calls to additional BLAS routines causes relatively moderate increases in BLIS-linked executable size. Adding calls to the other five double-precision real level-3 operations (DSYMM, DSYRK, DSYR2K, DTRMM, and DTRSM) results in 125Kbytes of additional object code when linking to BLIS, 178Kbytes when linking to OpenBLAS, 97Kbytes when linking to ATLAS, and 2.78Mbytes when linking to MKL.

However, some applications may need both real and complex domain flavors of the same operations. Adding calls to the double-precision complex analogues of the aforementioned level-3 routines causes OpenBLAS-, ATLAS-, and MKL-linked executables to swell in size by 296Kbytes, 1.05Mbytes, and 5.29Mbytes, respectively. On the other hand, adding these complex routine calls to a BLIS-linked executable causes *no* increase in executable size. This is possible because the real-only executables linked to BLIS *already* include all of the supporting infrastructure needed for computing in the complex domain. This is a consequence of BLIS's design, which defers the differentiation of domain (real versus complex) and precision (single versus double) until runtime.

Looking only at the library archives themselves, we see that BLIS[8] is smallest at 1.98Mbytes, with OpenBLAS and and ATLAS consuming 6.18Mbytes and 11.7Mbytes, respectively. ATLAS likely suffers, in general, from the fact that it is auto-generated. Also, ATLAS's design requires the compilation of many optimized "edge-case" kernels—enough to handle *any* possible edge case size (that is, any size less than the cache block size), which results in a very large kernel footprint. Similarly, OpenBLAS contains significant non-kernel code duplication and redundancy; however, this duplication also allows OpenBLAS-linked executables to stay exceptionally small when only a few BLAS routines are called, as each BLAS routine is more self-contained.

In some situations, executable size may not matter nearly as much as the total amount of memory allocated at run-time. All BLAS implementations require substantial workspace buffers, usually for creating packed copies of the matrix operands. The last two rows of Figure 10 list the total memory footprint of running processes when executing DGEMM for square problem sizes of 100 and 4000. For small problem sizes, the largest contributing factor to the runtime footprints of BLIS and OpenBLAS are packing buffers, which are statically sized at compile-time (as a function of the cache block sizes m_c, k_c, and n_c). ATLAS and MKL have similar workspace buffers. For larger problem sizes, the memory associated with the input matrices begins to dwarf any difference in workspace requirements.

Thus, BLIS may be a better choice than ATLAS or MKL when the footprint of the executable is an issue. Similarly, BLIS may be preferred when calling multiple BLAS (or BLAS-like) routines.

7. CONCLUSION, CONTRIBUTIONS AND FUTURE DIRECTIONS

The purpose of this paper 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

[8]This particular BLIS library was compiled with only optimized kernels for double-precision real computation. We estimate that the other kernels, were they to be written and included, would increase the size of the BLIS library by approximately 70Kbytes, resulting in a total library size of approximately 2.05Mbytes. This would also increase (by approximately the same amount) the sizes of the BLIS-linked executables listed in the first 12 rows of Figure 10.

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 micro-kernel, 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 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. 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. 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 a myriad of new research and development possibilities. Can high-performance micro-kernels 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 auto-tuning may be infeasible 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://code.google.com/p/blis/` .

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

ALI, M., STOTZER, E., IGUAL, F. D., AND VAN DE GEIJN, R. A. 2012. 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*. 179 –186.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., GREENBAUM, A., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

ATLAS for ARM home page 2013. `http://www.vesperix.com/arm/atlas-arm/index.html`.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16,* 1, 1–17.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14,* 1, 1–17.

FREESCALE SEMICONDUCTOR. 1999. AltiVec Technology Programming Interface Manual. Available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

GOTO, K. AND VAN DE GEIJN, R. 2008a. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft. 34,* 3, 12:1–12:25.

GOTO, K. AND VAN DE GEIJN, R. 2008b. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Soft. 35,* 1, 1–14.

GSCHWIND, M. 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, USA, 245–246.

GUNNELS, J. A., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001a. A family of high-performance matrix multiplication algorithms. In *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science 2073. Springer-Verlag, 51–60.

GUNNELS, J. A., VAN DE GEIJN, R. A., KATZ, D. S., AND QUINTANA-ORTI, E. S. 2001b. 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)*. IEEE Computer Society, Washington, DC, USA, 47–56.

HEINECKE, A., VAIDYANATHAN, K., SMELYANSKIY, M., KOBOTOV, A., DUBTSOV, R., HENRY, G., SHET, A. G., CHRYSOS, G., AND DUBEY, P. 2013. 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)*. To appear.

HUANG, K. AND ABRAHAM, J. 1984. Algorithm–based fault tolerance for matrix operations. *IEEE Trans. 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.

IGUAL, F. D., ALI, M., FRIEDMANN, A., STOTZER, E., WENTZ, T., AND VAN DE GEIJN, R. A. 2012. 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. IEEE Computer Society Press, IEEE Computer Society Press, Los Alamitos, CA, USA, 26:1–26:11.

KÅGSTRÖM, B., LING, P., AND LOAN, C. V. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft. 24,* 3, 268–302.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5,* 3, 308–323.

Loongson Technology Corp. Ltd 2009. *Loongson 3A processor manual*. Loongson Technology Corp. Ltd.

OpenBLAS 2012. http://xianyi.github.com/OpenBLAS/.

OPENMP ARCHITECTURE REVIEW BOARD. 2008. OpenMP application program interface version 3.0.

SINHAROY, B., KALLA, R., STARKE, W. J., LE, H. Q., CARGNONI, R., VAN NORSTRAND, J. A., RONCHETTI, B. J., STUECHELI, J., LEENSTRA, J., GUTHRIE, G. L., NGUYEN, D. Q., BLANER, B., MARINO, C. F., RETTER, E., AND WILLIAMS, P. 2011. IBM POWER7 multicore server processor. *IBM Journal of Research and Development 55,* 3, 1:1–1:29.

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

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

VAN ZEE, F. G. 2012. `libflame`*: The Complete Reference*. `www.lulu.com`.

VAN ZEE, F. G., CHAN, E., VAN DE GEIJN, R., QUINTANA-ORTÍ, E. S., AND QUINTANA-ORTÍ, G. 2009. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering 11,* 6, 56–62.

VAN ZEE, F. G. AND VAN DE GEIJN, R. A. 2012. BLIS: A framework for generating BLAS-like libraries. FLAME Working Note #66. Technical Report UTCS TR-12-30, UT-Austin. November.

VAN ZEE, F. G. AND VAN DE GEIJN, R. A. 2013. Blis: A framework for rapid instantiation of blas functionality. *ACM Trans. Math. Soft.*. In review.

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98*.

XIANYI, Z., QIAN, W., AND YUNQUAN, Z. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*.

YOTOV, K., LI, X., GARZARÁN, M. J., PADUA, D., PINGALI, K., AND STODGHILL, P. 2005. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93,* 2.