BLIS: A Framework for Rapidly Instantiating BLAS Functionality

FIELD G. VAN ZEE and ROBERT A. VAN DE GEIJN,
The University of Texas at Austin

The BLAS-like Library Instantiation Software (BLIS) framework is a new infrastructure for rapidly instantiating Basic Linear Algebra Subprograms (BLAS) functionality. Its fundamental innovation is that virtually all computation within level-2 (matrix-vector) and level-3 (matrix-matrix) BLAS operations can be expressed and optimized in terms of very simple kernels. While others have had similar insights, BLIS reduces the necessary kernels to what we believe is the simplest set that still supports the high performance that the computational science community demands. Higher-level framework code is generalized and implemented in ISO C99 so that it can be reused and/or re-parameterized for different operations (and different architectures) with little to no modification. Inserting high-performance kernels into the framework facilitates the immediate optimization of any BLAS-like operations which are cast in terms of these kernels, and thus the framework acts as a productivity multiplier. Users of BLAS-dependent applications are given a choice of using the traditional Fortran-77 BLAS interface, a generalized C interface, or any other higher level interface that builds upon this latter API. Preliminary performance of level-2 and level-3 operations is observed to be competitive with two mature open source libraries (OpenBLAS and ATLAS) as well as an established commercial product (Intel MKL).

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

General Terms: Algorithms, Performance

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

ACM Reference Format:
ACM Trans. Math. Softw. 0, 0, Article 0 (0000), 33 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The introduction of the Basic Linear Algebra Subprograms (BLAS) in the 1970s started a tradition of portable high performance for numerical software [Lawson et al. 1979; Dongarra et al. 1988; Dongarra et al. 1990]. The level-1 BLAS were proposed in the 1970s and targeted the vector-vector operations that supported high performance on the vector supercomputers of that era. When cache-based architectures arrived in the 1980s, performance required better amortization of data movement between memory, caches, and registers. The BLAS interface was extended to include matrix-vector (level-2) and matrix-matrix (level-3) operations. These last operations can achieve high performance by amortizing $\mathcal{O}(n^2)$ memory operations over $\mathcal{O}(n^3)$ floating-point operations. By casting computation in terms of these routines, high-performance could be attained on a wide range of architectures on which high-performance implementations were made available.

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.

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

The BLAS-like Library Instantiation Software (BLIS) is a software *framework* which allows experts and vendors[1] to rapidly instantiate high-performance libraries with BLAS functionality. It constitutes a major redesign of how BLAS are instantiated to be portable yet high-performing.

BLIS is part of a larger effort to overhaul the dense linear algebra software stack as part of the FLAME project [Van Zee et al. 2009]. As such, it provides a new companion C interface that is BLAS-like and, to us, more convenient, while fixing known problems with the BLAS interface and extending functionality. Because BLIS is a framework, it also supports the rapid implementation of new functionality that was, for example, identified as important by the BLAST Forum [BLAST 2002] yet was never widely-supported, presumably because no reference implementations were made available. We are fully aware that there is a "Don't Mess with the BLAS" attitude among many who depend on them and therefore we also make a traditional Fortran BLAS compatibility layer available.

This paper makes the following contributions:

— It discusses and summarizes our evaluation of the strengths and shortcomings of the original BLAS and explains why each shortcoming is worth addressing. This is important, because it defines the scope of what a new framework should be able to support.
— It proposes a new dense linear algebra framework that allows a developer of BLAS-like libraries to (1) rapidly instantiate an entire library on a new architecture with relatively little effort, and (2) implement new BLAS-like operations that, whenever possible, leverage existing components of the framework.
— It observes that level-2 and level-3 operations can be implemented in such a way that virtually *all* key differences can be factored out to generic, architecture-agnostic code, which in turn helps us isolate essential kernels that are easy to understand, implement, and reuse. This observation is key because it allows us to drastically reduce the scope of code that must be manually optimized for a given architecture, allowing the BLIS framework to act as a productivity multiplier.
— It demonstrates level-2 and level-3 performance that is highly competitive with (and, in some cases, superior to) existing BLAS implementations such as OpenBLAS, ATLAS, and Intel's Math Kernel Library (MKL) [Intel 2012].
— It makes the BLIS framework (including an optional BLAS compatibility layer) available to the community under an open source software license.[2]

These contributions, captured within the BLIS framework, lay the foundation for building modern dense linear algebra software that is flexible, portable, backwards compatible, and capable of high performance.

While we believe this paper makes significant contributions to the field, we readily admit that *this paper* does not address certain application needs such as multithreaded parallelism and environments that rely upon Graphics Processing Units (GPUs). We include a few comments regarding how BLIS will support this in the future, with pointers to early evidence, but intend to write a more complete paper on this in the near future. The subject matter is of a great enough scope that multiple papers will be needed to fully describe it.

[1] Henceforth, we will use the terms "experts," "vendors," and "[library] developers" interchangeably to refer generally to the same group of people—those who would use BLIS to instantiate a BLAS library for a particular (potentially new) architecture.
[2] We make the BLIS framework available under the so-called "new" or "modified" or "3-clause" BSD license.

1.1. Notation

Throughout this paper, we will use relatively well-established notation when describing linear algebra objects. Specifically, we will use uppercase Roman letters (e.g. A, B, and C) to refer to matrices, lowercase Roman letters (e.g. x, y, and z) to refer to vectors, and lowercase Greek letters (e.g. χ, ψ, and ζ) to refer to scalars.

2. REFLECTION

It is important to understand where our work fits into the dense linear algebra (DLA) software stack. At the bottom of the stack, there are the traditional BLAS, which provide basic vector-vector, matrix-vector, and matrix-matrix functionality. Above this is functionality that traditionally has been provided by the Linear Algebra Package (LAPACK) [Anderson et al. 1999]. This includes commonly-used solvers for linear systems, eigenvalue problems, and singular value problems, as well as more sophisticated operations. With the advent of SMP and multicore architectures, multithreading was added to the BLAS via which LAPACK then attained the benefit of multiple processing cores. More recently, other approaches to parallelism have been explored by the developers of LAPACK, including the MAGMA project [Agullo et al. 2009], which explores how to unleash GPUs for parallel computing, the PLASMA project [Agullo et al. 2009], which explores parallelism via task scheduling of what they call "tiled" algorithms, and Quark [Agullo et al. 2010], which more recently added the ability to execute tasks out-of-order. Prior to that, forming the top of the DLA software stack, was the introduction of a distributed memory library, ScaLAPACK [Choi et al. 1992], that mirrored LAPACK and was layered upon (distributed memory) Parallel BLAS (PBLAS), which themselves were layered upon the BLAS and the Basic Linear Algebra Communication Subprograms (BLACS) [Dongarra et al. 1993].

More recently, as part of the FLAME project, we have started to revisit the DLA software stack, funded by a sequence of National Science Foundation (NSF) grants that have culminated in a Software Infrastructure for Sustained Innovation (SI2) grant to vertically integrate DLA libraries. This effort builds on our experience while developing the `libflame` library [Van Zee et al. 2009; Van Zee 2012] (which targets essentially the same level of the stack as does LAPACK), the SuperMatrix runtime system [Chan et al. 2007; Chan et al. 2008; Quintana-Ortí et al. 2009] (which extracts parallelism in a way similar to that of MAGMA, PLASMA, and Quark, but within the `libflame` library), and the Elemental DLA library for distributed memory architectures [Poulson et al. 2013] (which extends many ideas from PLAPACK [van de Geijn 1997]). Funded by NSF, we are now integrating these libraries.

Since their earliest versions, `libflame` and Elemental have relied upon the traditional BLAS, exporting higher level interfaces that hide details of indexing. While developing and maintaining `libflame`'s external wrapper interfaces (i.e., object-based functions that directly call BLAS routines), we benefited from the strengths of the BLAS while also observing first-hand several shortcomings of the BLAS interface and its underlying functionality, as did the participants in the BLAST forums [BLAST 2002].

It is the traditional BLAS functionality for which the BLIS framework provides a rapid instantiation mechanism.

We will now visit each of the shortcomings of the BLAS since they set the stage for what challenges the framework will need to overcome.

— **Inflexible storage.** BLAS requires that matrices be stored in column-major order [Dongarra et al. 1988; Dongarra et al. 1990]. A C interface to the BLAS, known as CBLAS, supports both row- and column-major storage [BLAST 2002; BLAS 2012]. However, when row-major storage is used, CBLAS requires that *all* matrix operands

Level	Name	Operations
1	AXPY	$y := y + \alpha\overline{x}$
	COPY	$y := \overline{x}$
2	GEMV	$y := \beta y + \alpha\overline{A}x$ $y := \beta y + \alpha A\overline{x}$ $y := \beta y + \alpha\overline{Ax}$
	GER	$A := A + \alpha\overline{x}y^T$ $A := A + \alpha\overline{xy^T}$
	HER	$A := A + \alpha\overline{xx^H}$
	TRMV	$x := \overline{A}x$
	TRSV	$x := \overline{A^{-1}}x$
3	GEMM	$C := \beta C + \alpha\overline{A}B$ $C := \beta C + \alpha A\overline{B}$ $C := \beta C + \alpha\overline{AB}$
	HERK	$C := \beta C + \alpha\overline{AA^H}$ $C := \beta C + \alpha\overline{A^HA}$
	TRMM	$B := \alpha\overline{A}B$
	TRSM	$B := \alpha\overline{A^{-1}}B$

Fig. 1. A representative sample of variations of complex domain BLAS operations which are not implementable via the BLAS interface.

Level		Name	Operation	Description
1	V	SCAL2V	$y := \alpha\text{CJ}(x)$	Non-destructive vector scale.
		SETV	$\psi_i := \alpha, \quad \forall\psi_i \in y$	Set all vector elements to α.
		INVERTV	$\psi_i := \psi_i^{-1}, \quad \forall\psi_i \in y$	Invert all vector elements.
	M	AXPYM	$B := B + \alpha\text{CT}(A)$	Element-wise AXPY on matrices.
		COPYM	$B := \text{CT}(A)$	Element-wise COPY on matrices.
		SCALM	$B := \alpha\text{CJ}(B)$	Element-wise SCAL on matrices.
		SCAL2M	$B := \alpha\text{CT}(A)$	Element-wise SCAL2V on matrices.
		SETM	$\beta_{ij} := \alpha, \quad \forall\beta_{ij} \in B$	Set all matrix elements to α.
2		TRMV3	$y := \beta y + \alpha Ax$	Non-destructive TRMV.
		TRSV3	$y := \beta y + \alpha A^{-1}x$	Non-destructive TRSV.
3		TRMM3	$C := \beta C + \alpha AB$	Non-destructive TRMM.
		TRSM3	$C := \beta C + \alpha A^{-1}B$	Non-destructive TRSM.

Fig. 2. A sample of BLAS-like operations which were omitted from the BLAS. The functions CJ() and CT() denote optional conjugation and optional conjugation and/or transposition, respectively.

be stored in row-major order; thus, mixing row- and column-stored matrices within the same operation invocation is not allowed. To be fair, *some* combinations of row- and column-stored operands can be indirectly induced with BLAS that only support column (or row) storage by adjusting parameters, particularly the `trans`, `uplo`, and `side` parameters. However, this technique only results in partial support and tends to obstruct code readability. For some storage combinations, the operation is possible only by making temporary copies of certain matrices with explicit transpositions (and/or conjugations). Furthermore, no implementation of the BLAS allows general striding, whereby neither rows nor columns are contiguous in memory. Support for this advanced storage scheme is needed for dense matrix computation on tensors [Solomonik et al. 2012; Schatz et al. 2012].

Level	Name	Operations		
1M	AXPYM	$B_C := B_C + \alpha_C A_R$	$B_C := B_C + \alpha_R A_R$	
	COPYM	$B_C := A_R$		
	SCAL2M	$B_C := \alpha_C A_R$	$B_C := \alpha_R A_R$	
	SETM	$B_C := \alpha_R$		
2	GEMV	$y_C := y_C + A_C x_R$	$y_C := y_C + A_R x_C$	$y_C := y_C + A_R x_R$
	GER	$A_C := A_C + x_C y_R^T$	$A_C := A_C + x_R y_C^T$	$A_C := A_C + x_R y_R^T$
	HER	$A_C := A_C + x_R x_R^T$		
	TRMV	$x_C := A_R x_C$		
	TRSV	$x_C := A_R^{-1} x_C$		
3	GEMM	$C_C := C_C + A_C B_R$	$C_C := C_C + A_R B_C$	$C_C := C_C + A_R B_R$
	HERK	$C_C := C_C + A_R A_R^T$		
	TRMM	$B_C := A_R B_C$		
	TRSM	$B_C := A_R^{-1} B_C$		

Fig. 3. A representative sample of mixed domain operations which are not available via the BLAS interface. Subscripts $_R$ and $_C$ denote real and complex operands, respectively. Note that scalars, such as α and β, were omitted from the level-2 and level-3 operation expressions for clarity and typesetting reasons only. For the same reason we also omit the CJ() and CT() functions, which denote optional conjugation and optional conjugation and/or transposition, respectively.

— **Incomplete parameter support for the complex domain.** BLAS supports many operations on single- and double-precision complex data. However, for some of these operations, BLAS only implements the Hermitian case, leaving the complex symmetric case unsupported. Furthermore, the BLAS omits the "conjugate without transposition" option from all instances of the `trans` argument. Thus, many operations in the complex domain are not supported. Figure 1 contains a partial list of complex operations that are not supported by BLAS. One may argue that this is not a critical omission. After all, the programmer can simply conjugate an operand in-place, execute the desired BLAS operation, and then undo the conjugation. The problem with this technique—besides the obvious conjugation overhead incurred—is that it may not be thread-safe. In a multithreaded environment, another thread could attempt to read an operand in a temporarily conjugated state, which would result in a race condition. Alternatively, one could create a temporary conjugated copy of the operand. However, this still comes with potentially significant allocation, workspace, and memory copy costs.

— **Opaque API.** BLAS exports a "single-layer" API. That is, the application developer can access routines such as `dgemm`, but he or she cannot access the lower-level building blocks (the so-called "kernels") that facilitate the routine's high-performance implementation. Access to these building blocks is crucial for experts who wish to efficiently construct specialized routines that cannot be found in the BLAS. Without access to these lower-level APIs, an application developer who needs specialized functionality is forced to compose a solution strictly in terms of the exposed BLAS interfaces. This often results in sub-optimal implementations because intermediate data products cannot be re-used across successive BLAS calls and thus must be redundantly packed and/or computed. An example occurs, for example, in the application of a block Householder transformations [Joffrain et al. 2006; Bischof and Van Loan 1987; Schreiber and Van Loan 1989], a crucial component of several important operations found in LAPACK, including the QR factorization.

— **BLAS has not grown with community's needs.** The initial BLAS standard lacked many basic dense linear algebra operations such as, for example, element-wise operations on matrices. (Figure 2 lists some of these unsupported operations.) Furthermore, the set of operations supported today by the level-1, level-2, and level-3 BLAS has not changed since its inception in the 1980s. Indeed, the BLAST Forum attempted to standardize extensions to the BLAS that address some of the same issues discussed in the present paper. However, most of these extensions appear to be unsupported by some major commercial (MKL [Intel 2012], ACML [AMD 2012], ESSL [IBM 2012]) and open source (OpenBLAS [OpenBLAS 2012], ATLAS [Whaley and Dongarra 1998]) implementations[3]. Furthermore, the operations missing from the BLAS go beyond those identified during the BLAST forum. A recent paper showed that by providing an optimized routine for applying multiple sets of Givens rotations, high-performance eigenvalue (EVD) and singular value decomposition (SVD) can be achieved using a simple restructuring of the tridiagonal and bidiagonal QR algorithms [Van Zee et al. 2012], which, while numerically robust, were previously thought to be inherently low-performance algorithms. And prior to that, other efforts identified fused kernels that helped accelerate reduction to condensed form operations, which serve as preprocesses to some implementations of EVD and SVD [Howell et al. 2008; Van Zee et al. 2012].
— **Inability to compute with mixed domain operands.** With the exception of just a few level-1 routines, BLAS does not support operations where some operands are complex and some are real.[4] For example, the update of a complex vector by applying a real triangular matrix, via the `TRMV` operation, is not possible. One could work around this problem by copying the real triangular matrix to a temporary complex matrix of the same size (initializing the imaginary components to zero) and then computing with this complex matrix using `ztrmv`. However, this approach incurs considerable cost in workspace, memory operations, and code complexity, not to mention an unnecessary factor of two increase in floating-point operations. Figure 3 contains a partial list of potentially useful variations on existing complex BLAS routines where at least one of the operands resides entirely in the real domain.[5]
— **Very few portable high-performance frameworks.** There exist very few portable frameworks that allow a hardware vendor to easily port existing BLAS operations to new architectures and/or design new operations. (Indeed, we suspect one of the reasons the BLAST extensions were never widely adopted can be attributed to the lack of a complete reference implementation which vendors could use as a guide when optimizing their own versions.) This typically means that a vendor who wants to create a BLAS library for his or her new architecture must start "from scratch." Granted, some work would be needed regardless, as there would be small kernels which require optimization at a very low (non-portable) level. But, as we will discuss later in this paper, we have found that large parts of a BLAS implementation need not differ from architecture to architecture. Yet, existing open source solutions lack the flexibility and infrastructural leverage we envision.[6]

[3]While some extensions may be supported by some BLAS libraries, the extensions are not supported uniformly by all such libraries, and hence using them would tend to reduce portability.
[4]The BLAST Forum proposed mixed domain and mixed precision functionality [BLAST 2002], but these extensions were never adopted by the community, perhaps because they targeted a set of less frequently-encountered use cases.
[5]Not all combinations of real and complex operands make sense for all operations. However, there are enough combinations that *do* make sense to merit consideration.
[6]Among the most prominent open source BLAS implementations are netlib BLAS, OpenBLAS, and ATLAS. The netlib BLAS constitutes the original reference implementation and is quite portable, but makes very little reuse of internal code [BLAS 2012] and does not take advantage of the memory cache hierarchies

Any one of these deficiencies may not be fatal for a given application, but to some developers, including us, the original BLAS are simply inadequate. Perhaps most importantly, when taken together and multiplied over the hundreds or thousands of application developers who must spend time and effort to fashion nearly identical workarounds (which subsequently perform sub-optimally), these flaws in the BLAS constitute a considerable loss in human productivity.

3. FEATURE SET

Building on the observations in the previous sections, we propose the BLIS framework to ideally meet the following criteria:

— **Generalized storage.** The BLIS framework should support column-major, row-major, or general stride storage, and should do so for individual matrix operands, and should do so with minimal (if any) impact on performance. This means, for example, that the BLIS version of GEMM should support the operation $C = \beta C + \alpha AB$, where C is row-stored, A is column-stored, and B is stored with general stride (i.e., non-unit in both dimensions). In addition to this, we want BLIS to support basic element-wise operations on lower- or upper-trapezoidal matrices with arbitrary diagonal offsets.

— **Full support for the complex domain.** Operations should be primarily designed and developed in their most general form, which is typically in the complex domain. These formulations then simplify down to real domain analogues with relative ease (not to mention mathematical elegance). For mathematical "symmetry" between the real and complex domains, conjugation on real vectors and matrices should be allowed by the interface and treated as a *no-op*. Also, where applicable, both complex Hermitian and complex symmetric operations should be supported.

— **Multi-layered API with exposed kernels.** The framework should expose its implementations in various API layers. Low-level kernels will be available to experts who wish to design new linear algebra operations, or reconfigure existing ones to meet the needs of the application or hardware. Operation implementations would further consist of higher-level blocked algorithm codes as well as lower-level (but still portable) "macro-kernels," which interface directly with architecture-aware micro-kernels. Optimizations can occur at various levels, in part thanks to exposed packing and unpacking facilities, which by default are highly parameterized (and thus flexible).

— **Portability *and* high performance.** Portability should be a top priority. To achieve this, non-portable, architecture-specific codes should be compartmentalized into very small kernels.[7] As such, once these codes are provided, virtually all BLIS operations would be enabled at very little additional effort because all BLIS operations are expressed and implemented in terms of these kernels. Performance would be attained via careful implementation of the kernels, and thus would be inherited into multiple families of operations. For example, once an efficient matrix-matrix multiplication kernel is developed, nearly all level-3 operations should be enabled and attain high performance.

---

found on nearly all modern architectures. Thus, it tends to perform very poorly. OpenBLAS, a fork of the discontinued GotoBLAS, offers competitive performance on a range of architectures, but is implemented as a BLAS library rather than a framework for building such libraries. ATLAS provides a system similar to that of OpenBLAS, except that in certain instances block sizes are automatically fine-tuned for cache performance [Whaley and Dongarra 1998]. However, once again, while ATLAS is well-suited to generating BLAS libraries with a fixed set of functionality, it does not appear designed to facilitate the implementation of new BLAS-like operations. Thus, we are left without a true framework for building high performance BLAS-like libraries.

[7] By "small," we mean that these kernels are typically implemented as a single loop around highly-optimized code, typically expressed in an assembly or assembly-like language.

— **Ease of use.** The BLIS framework, and the library of routines it generates, should be easy to use for end users, experts, and vendors alike. A BLAS compatibility layer would be available to allow effortless linkage with existing BLAS-depending codes. Of course, this layer will only export the smaller subset of BLIS functionality that directly corresponds to the BLAS interface. Users who are willing to use BLIS's new BLAS-like interfaces will have access to the full set of BLIS operations and features. Motivated users and library developers would also have access to easy-to-use, object-based APIs.
— **Functionality that grows with the community's needs.** The BLIS framework should be, as its name suggests, a framework with which one can rapidly instantiate BLAS-like libraries rather than a library or static API alone. Furthermore, the framework should be extensible, allowing its developers (and others) to leverage existing components within the framework to support new operations as they are identified. As time goes on, it may be the case that new kernels are required in order to most efficiently support certain operations, such as the case with the application of multiple Givens rotations [Van Zee et al. 2012].
— **Support for mixed domain (and/or mixed precision) operations.** The BLIS framework should allow an expert to install additional kernels that enable various mixed domain operations. Similarly, operations that mix precision (and those that mix both domain and precision) should also be available to those who need them. However, neither should be mandatory. By making these features optional, we avoid unnecessarily burdening library developers with operations that their users do not need. But, importantly, the mixed domain/precision infrastructure should already be present, even when the corresponding kernels are left unoptimized. This way, a user can, at the very least, conveniently solve his or her mixed domain/precision problem, even if it requires the use of an unoptimized reference code.
— **Prioritize code re-use to minimize binary footprint.** A brute force or auto-generation approach to achieving the aforementioned goals tends to quickly lead to code bloat due to the multiple dimensions of variation that must be supported, namely: operation (i.e., GEMM, HEMM, TRMM, etc.); parameter (i.e., side, transposition, upper/lower structure, unit/non-unit diagonal); datatype (i.e., single-/double-precision real/complex); storage (i.e., row, column, general); and algorithm (i.e., partitioning path and kernel shape). The reason is simple: auto-generation techniques inherently tend to consider and optimize one specific case of one operation at a time. The BLIS framework should feature a holistic design that takes reasonable steps to limit the incursion of code bloat while still providing the targeted functionality. This may be primarily achieved by careful abstraction, layering, and re-use of generic codes, subject to the constraint that performance-penalizing design decisions and coding conventions should be minimized.

Realizing all of these goals simultaneously in one software framework constitutes a significant improvement over prior instantiations of the traditional BLAS. While this list may seem ambitious, our preliminary findings suggest that attaining these goals is entirely within reach on modern architectures.

4. SUPPORTED OPERATIONS

Before discussing *how* the previously mentioned features are realized within BLIS, and other implementation details, we first wish to provide high-level descriptions of the core operations supported by the framework. We do not give APIs for these operations since the BLIS interface is, for now, mostly for internal consumption. Furthermore, others may wish to layer their own custom APIs upon the interfaces exposed by the framework.

Name	Operation	Description
AXPYV	$y := y + \alpha \text{CJ}(x)$	Accumulate scaled vector.
COPYV	$y := \text{CJ}(x)$	Copy vector.
DOTV	$\rho := \text{CJ}(x)^T \text{CJ}(y)$	Dot product.
DOTXV	$\rho := \beta\rho + \alpha \text{CJ}(x)^T \text{CJ}(y)$	Generalized dot product.
INVERTV	$\psi_i := \psi_i^{-1}, \quad \forall \psi_i \in y$	Invert all vector elements.
SETV	$\psi_i := \alpha, \quad \forall \psi_i \in y$	Set all vector elements to α.
SCALV	$y := \alpha y$	In-place vector scale.
SCAL2V	$y := \alpha \text{CJ}(x)$	Non-destructive vector scale.

Fig. 4. A list of level-1v operations supported by the BLIS framework. The function CJ() denotes optional conjugation.

The BLIS framework organizes its operations into classes of operations, where each class contains related operations. As with the BLAS, we refer to these classes as "levels." Notice that these classes have as much to do with the kernels that need to be implemented as the functionality that they provide.

4.1. Level-1v: Vector-vector Operations

The level-1 BLAS specification contains operations on vectors. The BLIS framework supports similar operations, which we collectively refer to as level-1v BLIS.

Figure 4 contains a list of level-1v operations supported by BLIS. Note that in the mathematical expressions shown here, an instance of the CJ() function denotes the opportunity to optionally conjugate an operand.[8] We emphasize that this list is incomplete and is only meant to provide a representative sample of operations which can (and should) be supported by BLIS.

Experienced users will immediately notice a few differences between level-1 BLAS and level-1v BLIS, most notably with regards to expanded functionality. We briefly discuss the main differences:

— Level-1v BLIS operations are presented with a "V" appended to their names. This is done to differentiate them from similar operations that take matrix (instead of vector) arguments, which will be discussed in Section 4.2
— All operations allow the user to optionally conjugate vectors that are strictly input operands. For example, a developer could use AXPYV to compute with a vector x as if it were conjugated. Note that this optional conjugation does *not* affect the state of vector x during the course of the computation, thus maintaining thread safety.
— New operations are now available, such as DOTXV and SCAL2V. The former is useful when one wishes to scale and/or accumulate a dot product into an existing scalar, while the latter can be used to simultaneously copy and scale a vector. Other routines are provided for similar convenience, such as SETV and INVERTV.

The net effect of the changes above will be to simplify certain unblocked algorithms, including those for operations typically found within LAPACK [Anderson et al. 1999].

4.2. Level-1m: Vector-Vector Operations on Matrices

One of the more obvious omissions from the BLAS can be found in its lack of element-wise operations on matrix operands. Many of the operations one might want to perform

[8] As one might expect, this conjugation reduces to a *no-op* when the operand is stored in the real domain.

Name	Operation	Description
AXPYM	$B := B + \alpha \text{CT}(A)$	Accumulate scaled matrix.
COPYM	$B := \text{CT}(A)$	Copy matrix.
SETM	$\beta_{ij} := \alpha, \quad \forall \beta_{ij} \in B$	Set all matrix elements to α.
SCALM	$B := \alpha B$	In-place matrix scale.
SCAL2M	$B := \alpha \text{CT}(A)$	Non-destructive matrix scale.

Fig. 5. A list of level-1m operations supported by the BLIS framework. The functions CJ() and CT() denote optional conjugation and optional conjugation and/or transposition, respectively. Each of the operations listed above is supported on dense as well as lower or upper triangular/trapezoidal matrices with arbitrary diagonal offsets, such that only the specified region(s) are referenced and/or updated.

on vectors, such as AXPYV, COPYV, or SCALV, can also be meaningfully performed on matrices. One needs look no further than computational tools such as MATLAB to find this sort of functionality implemented and easily accessible [Moler et al. 1987].

As alluded to previously, there are workarounds for this situation that are not unreasonable for certain users. For example, if one wishes to perform $B := B + \alpha A$ (i.e., an AXPY operation on matrices), he or she can insert a loop over an AXPYV-like routine that scales and accumulates the matrices one column at a time. But what if the matrices are stored by rows? To achieve unit stride, the loop would need to accordingly be over rows rather than columns. Or what if the user wishes to perform a conjugation to the input matrix A, or a transposition (or both)? What if the matrices are triangular (or symmetric or Hermitian) in structure and thus only stored in the lower or upper triangle? Finally, what if the stored region is not strictly triangular, but rather dependent upon the precise location of the matrices' main diagonals? It is easy to see how implementing such functionality using only AXPYV could prove daunting for many BLAS users. Yet another problem with the aforementioned solution is that looping over a top-level AXPYV interface (such as via the BLAS) results in repeated error checking on matrix columns (or rows), which in this case one would preferably avoid in favor of a single set of checks up-front. Thus, leaving the application developer to implement a custom solution is not ideal.

The BLIS framework supports several level-1m operations on matrices. Of these, a representative list is shown in Figure 5. These operations are provided with enough parameterization and generality to handle any combination of the situations listed above, in addition to a few situations not mentioned such as general stride storage and implicit unit diagonals.

4.3. Level-1f: Fused Vector-vector Operations

It has been shown elsewhere that so-called "fused" kernels can be used to more efficiently implement certain operations that are rich in $\mathcal{O}(n^2)$ floating-point operations (flops) performed on $\mathcal{O}(n^2)$ data [Howell et al. 2008; Van Zee et al. 2012]. The performance of such operations is greatly hindered by two facts. First, on modern architectures, memory load and store instructions tend to be much more expensive than floating-point instructions. Second, since the ratio of flops to memory operations (memops) is typically small, the computation often stalls while waiting for new data to be fetched from (or stored back to) memory.

While these level-2 operations will never perform as well as their level-3 brethren (which inherently call for $\mathcal{O}(n^3)$ flops to be performed on $\mathcal{O}(n^2)$ data), non-trivial performance gains can still be achieved for many problem sizes by carefully reusing val-

Name	Operation	Description
AXPY2V	$\begin{cases} y := y + \alpha_0 \text{CJ}(x_0) \\ y := y + \alpha_1 \text{CJ}(x_1) \end{cases}$	Fused AXPYV pair with shared y.
DOTAXPYV	$\begin{cases} \rho := \text{CJ}(x)^T y \\ z := z + \alpha \text{CJ}(x) \end{cases}$	Fused DOTV-AXPYV with shared x.
AXPYF	$\begin{cases} y := y + \alpha \text{CJ}(\chi_0)\text{CJ}(a_0) \\ \vdots \\ y := y + \alpha \text{CJ}(\chi_{f-1})\text{CJ}(a_{f-1}) \end{cases}$	Fused sequence of f AXPYV with shared y, where f is implementation-dependent.
DOTXF	$\begin{cases} \psi_0 := \beta\psi_0 + \alpha \text{CJ}(a_0)^T \text{CJ}(x) \\ \vdots \\ \psi_{f-1} := \beta\psi_{f-1} + \alpha \text{CJ}(a_{f-1})^T \text{CJ}(x) \end{cases}$	Fused sequence of f DOTXV with shared x, where f is implementation-dependent.
DOTXAXPYF	$\begin{cases} y_0 := \beta y_0 + \alpha \text{CJ}(A^T)\text{CJ}(x_0) \\ y_1 := y_1 + \alpha \text{CJ}(A)\text{CJ}(x_1) \end{cases}$	Fused DOTXF-AXPYF with shared $(a_0, \cdots, a_{f-1}) = A$, where f is implementation-dependent.

Fig. 6. A list of level-1f operations supported by the BLIS framework. The function CJ() denotes optional conjugation. Note that above, f corresponds to the implementation-dependent fusing factor for the operation in question. These fusing factors may be queried from within the framework.

```
for i = 0 : m − 1
    LOAD y_i → ψ1
    ψ1 := βψ1
    STORE y_i ← ψ1
endfor
for j = 0 : n − 1
    LOAD x_j → χ1
    χ1 := αχ1
    for i = 0 : m − 1
        LOAD A_ij → α1
        LOAD y_i → ψ1
        ψ1 := ψ1 + α1χ1
        STORE y_i ← ψ1
    endfor
endfor
```

2mn + m + n flops
3mn + 2m + n memops

```
for i = 0 : m − 1
    LOAD y_i → ψ1
    ψ1 := βψ1
    STORE y_i ← ψ1
endfor
for j = 0 : n − 1 : 4
    LOAD x_{j:j+3} → χ_{1:4}
    χ_{1:4} := αχ_{1:4}
    for i = 0 : m − 1
        LOAD A_{i,j:j+3} → α_{1:4}
        LOAD y_i → ψ1
        ψ1 := ψ1 + α1χ1 + α2χ2
                 + α3χ3 + α4χ4
        STORE y_i ← ψ1
    endfor
endfor
```

} AXPYF(α, A1, x1, y) where A1 = A_{j:j+3}, x1 = x_{j:j+3}

2mn + m + n flops
$\frac{3}{2}mn + 2m + n$ memops

Fig. 7. Left: A column-based (i.e., AXPYV-based) algorithm for computing the GEMV operation $y := \beta y + \alpha Ax$. This algorithm requires $3mn + 2m + n$ memory operations (ignoring the loading of α and β). Center: By unrolling and fusing 4 iterations at a time, we are able to eliminate 3 out of every 4 loads and stores on vector y while still performing the same number of flops. The affected code is highlighted in grey and corresponds to an AXPYF kernel operation with a fusing factor of $f = 4$. Thus, by implementing GEMV in terms of this fused kernel, we reduce the total number of memory operations incurred by nearly a factor of two.

ues once they have been loaded into the processor core's registers. We refer to this technique as *register-level fusing* [Van Zee et al. 2012].[9]

[9]One may also use cache-level fusing, which, as its name suggests, targets reuse of data while it resides in cache memory. However, in most situations, fusing at the register level will almost always yield superior

Name	Operation(s)	Description
GEMV	$y := \beta y + \alpha \text{CT}(A)\text{CJ}(x)$	General matrix-vector multiply.
GER	$A := A + \alpha \text{CJ}(x)\text{CJ}(y)^T$	General rank-1 update.
HEMV	$y := \beta y + \alpha \text{CJ}(A)\text{CJ}(x)$	Hermitian matrix-vector multiply.
HER	$A := A + \alpha \text{CJ}(xx^H)$	Hermitian rank-1 update.
HER2	$A := A + \alpha \text{CJ}_x(x)\text{CJ}_y(y)^H + \bar{\alpha} \text{CJ}_y(y)\text{CJ}_x(x)^H$	Hermitian rank-2 update.
SYMV	$y := \beta y + \alpha \text{CJ}(A)\text{CJ}(x)$	Symmetric matrix-vector multiply.
SYR	$A := A + \alpha \text{CJ}(xx^T)$	Symmetric rank-1 update.
SYR2	$A := A + \alpha \text{CJ}_x(x)\text{CJ}_y(y)^T + \alpha \text{CJ}_y(y)\text{CJ}_x(x)^T$	Symmetric rank-2 update.
TRMV	$x := \alpha \text{CT}(A)x$	Triangular matrix-vector multiply.
TRSV	$x := \alpha \text{CT}(A)^{-1}x$	Triangular solve.

Fig. 8. A list of level-2 operations supported by the BLIS framework. The functions CJ() and CT() denote optional conjugation and optional conjugation and/or transposition, respectively. A subscript is used in conjuction with CJ() to indicate two instances of the same conjugation.

Figure 6 shows a list of operations supported by the BLIS framework which, if implemented with register-level fusing, would facilitate higher levels of performance for key subproblems found within various level-2 algorithms. For example, the operation GEMV may be cast as a series AXPYV (or DOTV) operations. If groups of f AXPYV subproblems are fused together, then certain operand elements, once loaded into registers, may be reused $f-1$ times. This reuse of data typically results in a higher performance, as less time is spent waiting for operand elements to arrive from memory. Naturally, the implementor would wish to maximize this reuse of data, and so he or she would aim to implement these level-1f operations with as large of a "fusing factor" f as possible, given the constraints of the floating-point datatype and the size of the architecture's register set. (While these fusing factors are implementation-dependent, they are made available to other framework components, as well as application code, via query macros.)

Figure 7 illustrates the previous example with pseudo-code that exposes individual load and store instructions. Here, it is easy to see that an AXPYF implemented with a fusing factor of $f = 4$ reduces the number of memops needed to compute GEMV by nearly a factor of two.

An alternative fusing may be achieved via the DOTXF kernel operation. Similarly, the other level-1f operations listed in Figure 6 may be used to optimize various other level-2 operations, which are discussed in the next section.

Note that AXPY2V is a special case of AXPYF where $f = 2$. However, as we will see in the next section, AXPY2V is still needed when optimizing the level-2 operations HER2 and SYR2, which provide the opportunity to fuse *only* and *exactly* two instances of AXPYV per row or column of the output matrix.

4.4. Level-2: Matrix-Vector Operations

The level-2 operations available in the BLIS framework are largely similar to the corresponding operations within the level-2 BLAS. Figure 8 lists these operations. The differences between the two are enumerated as follows:

— For all operations where a matrix or vector is strictly an input operand, that operand may be used as if it were conjugated. This option is in addition to the transposition or conjugate-transposition options already allowed by the BLAS.
— Unlike in the BLAS, the TRMV and TRSV operations scale their matrix-vector products by a scalar α before overwriting their results to vector x. This scaling establishes consistency with the operations' level-3 analogues, TRMM and TRSM, respectively.
— BLIS supports non-destructive TRMV and TRSV operations via TRMV3 and TRSV3, respectively.
— While not obvious from Figure 8, BLIS implements SYMV, SYR, and SYR2 for the complex domain, whereas the BLAS omits these operations.

Like `libflame`, the BLIS framework provides multiple algorithmic variants[10] for many of the operations it implements [Van Zee et al. 2009]. In the case of level-2 operations, it is sometimes advantageous to choose a different variant depending on how the matrix operand is stored. To achieve unit stride[11] through memory, matrices that are column-stored should be computed upon with variants that traverse individual columns. Similarly, row-based variants tend to be well-suited to matrices stored by rows. (Note that in either case, a transposition can cause this affinity to flip.) Thus, for example, a GEMV variant based on AXPYV is appropriate for column-stored matrices (or transposed row-stored matrices), while GEMV based on DOTXV should be preferred for row-stored matrices (or transposed column-stored matrices). The BLIS framework allows us to support both of these situations. (A general strided matrix is probably best computed upon via a DOTXV-based kernel because it exhibits an inherently lower ratio of memops to flops than one based on AXPYV, though there may be other considerations beyond the scope of this document that affect the optimal choice.)

Recall that in Section 4.3 we discussed various level-1f BLIS operations that use register-level fusing to avoid redundant memops within level-2 operations. Figure 9 shows which level-2 operations benefit from the optimization of various level-1v and level-1f operations. Importantly, our preliminary experience shows that optimizing these kernels is sufficient in order to achieve competitive level-2 performance on modern architectures. (Performance results for level-2 operations that employ these kernel operations may be found in Section 7.2.)

To put things concretely, Figure 9 reveals that optimizing *only* DOTXV and AXPYV is sufficient to enable accelerated implementations of *all* level-2 operations[12]. These implementations would be efficient regardless of whether the matrix it operates upon was stored by rows or columns. If one were to go further and implement architecture-tuned versions of AXPY2V, DOTXF, AXPYF, and DOTXAXPYF, in addition to AXPYV, one could use architecture-agnostic components of the BLIS framework to quickly instantiate a set of level-2 BLIS (and BLAS) that was even more fully optimized.

---

performance because it allows one to avoid certain memory accesses altogether, rather than simply reduce their cost.

[10] Henceforth, we abbreviate the term "algorithmic variant" to simply "variant."

[11] Unit stride through memory is usually, though not universally, desired on modern cache-based systems because it typically increases spatial locality of data in the cache hierarchy.

[12] The reader should exclude banded and packed level-2 operations when considering this assertion. Banded and packed level-2 operations require very special storage formats, and these storage formats naturally require that the kernels be parameterized differently. Thus, the best approach to these operations remains an open question and may be the topic of future research.

Level-2 Operation	Matrix storage	By optimizing this kernel, level-2 performance is...		
		Improved by	Further improved by	Optimized by
GEMV, TRMV, TRSV	column-stored	AXPYV		AXPYF
	row-stored	DOTXV		DOTXF
HEMV, SYMV	column-stored row-stored	DOTXV + AXPYV	DOTAXPYV, DOTXF + AXPYF	DOTXAXPYF
GER, HER, SYR	column-stored row-stored	AXPYV		
HER2, SYR2	column-stored row-stored	AXPYV		AXPY2V

Fig. 9. Here we present optimization dependencies for each of the level-2 operations supported in the BLIS framework. The table shows which level-1v and/or level-1f operations must be implemented and optimized in order to achieve higher performance for a given level-2 operation. Note that for general stride matrix storage, unit stride through memory cannot be achieved, and thus when given the choice (i.e., for GEMV, TRMV, and TRSV) either DOT-based or AXPYV-based kernels may be used, though the former may be preferred due to the inherently lower ratio of memops to flops associated with DOT-based level-2 algorithms.

Name	Operations(s)	Description
GEMM	$C := \beta C + \alpha \text{CT}(A)\text{CT}(B)$	General matrix-matrix multiply.
HEMM	$C := \beta C + \alpha \text{CJ}(A)\text{CT}(B)$, or $C := \beta C + \alpha \text{CT}(B)\text{CJ}(A)$	Hermitian matrix-matrix multiply.
HERK	$C := \beta C + \alpha \text{CT}_A(A)\text{CT}_A(A)^H$	Hermitian rank-k update.
HER2K	$C := \beta C + \alpha \text{CT}_A(A)\text{CT}_B(B)^H$ $+ \bar{\alpha} \text{CT}_B(B)\text{CT}_A(A)^H$	Hermitian rank-$2k$ update.
SYMM	$C := \beta C + \alpha \text{CJ}(A)\text{CT}(B)$, or $C := \beta C + \alpha \text{CT}(B)\text{CJ}(A)$	Symmetric matrix-matrix multiply.
SYRK	$C := \beta C + \alpha \text{CT}_A(A)\text{CT}_A(A)^T$	Symmetric rank-k update.
SYR2K	$C := \beta C + \alpha \text{CT}_A(A)\text{CT}_B(B)^T$ $+ \alpha \text{CT}_B(B)\text{CT}_A(A)^T$	Symmetric rank-$2k$ update.
TRMM	$B := \alpha \text{CT}(A)B$, or $B := \alpha B \text{CT}(A)$	Triangular matrix-matrix multiply.
TRMM3	$C := \beta C + \alpha \text{CT}(A)\text{CT}(B)$, or $C := \beta C + \alpha \text{CT}(B)\text{CT}(A)$	Non-destructive triangular matrix-matrix multiply.
TRSM	$B := \alpha \text{CT}(A)^{-1}B$, or $B := \alpha B \text{CT}(A)^{-1}$	Triangular solve with multiple right-hand sides.

Fig. 10. A list of level-3 operations supported by the BLIS framework. The functions CJ() and CT() denote optional conjugation and optional conjugation and/or transposition, respectively. A subscript is used in conjuction with CT() to indicate two instances of the same conjugation and/or transposition.

4.5. Level-3: Matrix-Matrix Operations

Figure 10 lists the core set of level-3 operations supported by the BLIS framework. These operations largely correspond to the level-3 operations provided by the BLAS. The most prominent differences mirror those found between the level-2 BLIS and BLAS: namely, the ability to conjugate individual input operands.

Just as the level-2 operations are built atop kernels which may be optimized to enable high performance, level-3 operations are likewise implemented in terms of smaller kernels. This idea is not new [Goto and van de Geijn 2008a; 2008b; Gunnels et al. 2001b; Whaley and Dongarra 1998]. Section 5 discusses how these level-3 operations

are implemented in the BLIS framework so that flexibility (i.e., generality), portability, and high performance are simultaneously achieved.

5. IMPLEMENTATION TECHNIQUES

Implementing the BLIS framework is fraught with challenges, especially if one is to avoid explosive code bloat. The last bullet at the end of Section 3 touches upon several "dimensions" of generality to be captured by BLIS. If the framework itself was to be manageable and maintainable going forward, we had to find ways of collapsing each of these dimensions as much as possible while maintaining the desired functionality. Here, we provide a detailed discussion of some of the implementation techniques and design decisions that allow us to achieve most of our functionality goals for BLIS without incurring significant increases in source (or object) code. We consider these software engineering details to be important because they facilitate the primary contributions of this paper.

For reasons that will seem clearer later on, we discuss our techniques for implementing the level-3 BLIS separately from that of the other levels.

5.1. Level-3 techniques

Level-3 operations make up perhaps the most-used portion of the BLAS.[13] These operations are also typically the most difficult to implement, in part because the space of potential solutions is much larger than that of level-2 and lower operations. It may, therefore, seem counterintuitive that the portion of the BLIS framework associated with level-3 operations is actually *simpler* (and smaller) than that of the remaining operations. We now walk through the methods used to implement this functionality.

5.1.1. Operation. Perhaps the most obvious framework dimension we wish to manage is that of the operations supported. Despite seemingly significant differences, all level-3 operations are inherently very similar. It was observed by [Kågström et al. 1998] that level-3 operations can be cast in terms of general rank-k update (GEMM). The authors of [Goto and van de Geijn 2008b] built on this idea but observed that packing facilities could be modified to induce all level-3 operations via the same low-level kernels that support the implementation of GEMM. Let us use Figure 11 to briefly describe this algorithm. A matrix multiplication with presumably large m, n, and k dimensions is first partitioned along k with a cache block size of k_c, creating rank-k subproblems (depicted at the bottom of Figure 11). At this stage, B is packed to contiguous memory (labeled as $\tilde{B}$) in a special format that facilitates unit stride at the lowest level of computation. Each rank-k subproblem is then partitioned along the m dimension with a cache block size of m_c, creating block-panel subproblems. The current $m_c \times k_c$ block A_i is then packed to $\tilde{A}_i$, which is once again stored in a special format. This block-panel subproblem that remains ($C_i := C_i + \tilde{A}_i \tilde{B}$) is then implemented as a highly optimized, assembly-coded kernel. This kernel will continue on to partition the matrices along the n, m_c, and finally k_c dimensions. Now, to extend this algorithm to other level-3 operations, the authors of [Goto and van de Geijn 2008b] point out that sometimes the only change required is in the packing facility. For example, for HEMM and SYMM, if $\tilde{A}_i$ is packed in such a way that blocks that intersect the diagonal are made dense (since only one triangle is stored), then the block-panel kernel can be used unmodified. However, HERK, SYRK, HER2K, and SYR2K require that the block-panel kernel be specialized because only the part of C_i that is stored is to be updated. Similar complications arise

[13] This is likely no coincidence, as the level-3 operations are typically capable of achieving a large fraction of peak performance on modern architectures. Thus, application developers have an incentive to express their computation in terms of level-3 operations whenever possible.

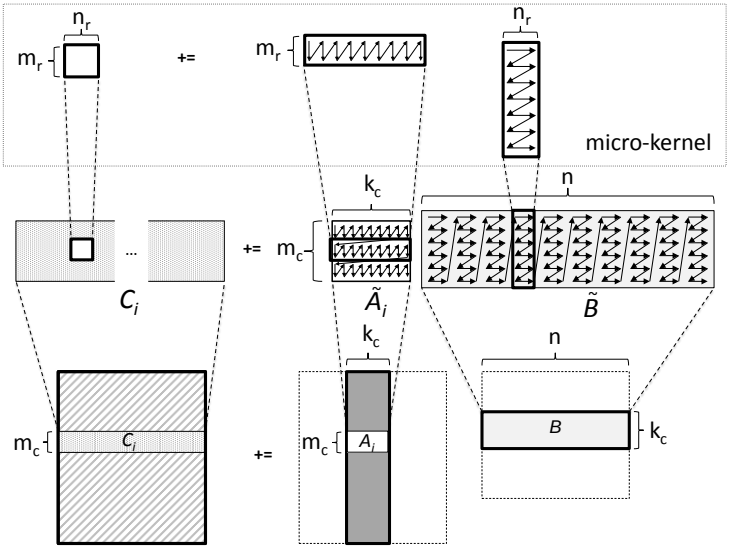

Fig. 11. Illustration of the various levels of blocking and related packing when implementing GEMM in the style of [Goto and van de Geijn 2008a]. Here, m_c and k_c serve as cache block sizes used by the higher-level blocked algorithms to partition the matrix problem down to a so-called "block-panel" subproblem (depicted in the middle of the diagram), implemented in BLIS as a portable macro-kernel. Similarly, m_r and n_r serve as register block sizes for the micro-kernel in the m and n dimensions, respectively, which also correspond to the length and width of the individual packed panels of matrices $\tilde{A}_i$ and $\tilde{B}$, respectively.

with TRMM and TRSM due to the potential triangular nature of A_i, which thus require additional specialization. To avoid this proliferation of kernels, the BLIS framework uses a somewhat different layering.

In BLIS, the fundamental kernel encompasses only the inner-most computation that iterates over the k_c dimension (depicted at the top of Figure 11). Because this kernel is a much smaller, more basic operation, we refer to it as a "micro-kernel."[14] Similarly, we use the term "macro-kernel" to refer to the two loops above this micro-kernel (i.e., those loops that partition along the n and m_c dimensions, which correspond to the outer and middle loops in a Goto-style kernel). This layering provides two key benefits:

— It allows virtually all of the architectural details to be confined to a much smaller, simpler kernel of computation; and
— It allows nearly all of the operation-specific nuances of the level-3 kernels for HERK, SYRK, HER2K, SYR2K, TRMM and TRSM to be factored out to different macro-kernels, each of which is a portable, pre-supplied component of the BLIS framework.[15]

An additional benefit of the macro-kernel/micro-kernel design used by BLIS is that it facilitates portable edge-case handling. That is, the edge case handling occurs entirely

[14] Our use of a micro-kernel as the basic kernel of computation, as well as its general implementation (as a series of rank-1 updates), is strikingly similar to the corresponding kernel of another effort, which investigates implementing a linear algebra co-processor in hardware [Pedram et al. 2012b; Pedram et al. 2012a]. Interestingly, the two projects isolated their respective kernels independently of one another, despite the kernels' similarities and the projects' collaborative proximity to one another.

[15] To maximize TRSM performance, a library developer would typically need to supply an additional micro-kernel that fuses a GEMM subproblem and TRSM subproblem in order to avoid redundant memory operations. In practice, we have observed that 80–90% of potential performance is attainable even when this fused micro-kernel is not supplied. We illustrate this performance gap on an Intel Xeon architecture in Section 7.3.

within the macro-kernel. When $m \bmod m_r \neq 0$ or $n \bmod n_r \neq 0$, the same micro-kernel is used, except the result is stored to a temporary $m_r \times n_r$ buffer. (This is always possible because packed matrix dimensions are inflated to a multiple of m_r or n_r, as necessary, so that the packing facility can perform zero-padding.) Then, only those elements of the local buffer corresponding to stored values are written to the output matrix. No additional micro-kernels are required to handle edge cases. This approach requires some extra macro-kernel logic, but results in only an additional $\mathcal{O}(n)$ memory operations, and thus the net impact on performance is, under modest assumptions, negligible.

Thus, the *only* code that must be optimized for a given architecture is that of the micro-kernel.[16]

Once the micro-kernel is written (which inherently involves choosing register block sizes), the library developer need only choose appropriate cache block sizes to achieve a full implementation of GEMM. From there, virtually no additional work is needed (on the part of the developer) to instantiate high-performance implementations of the remaining level-3 operations. This leveraging is immediate and automatic because the BLIS framework already encodes the differences in the various level-3 operations into (1) the portable macro-kernels, and (2) the framework's highly parameterized packing facility.

5.1.2. Parameter case. The dimension most vulnerable to code duplication is arguably that of operational parameters. Each BLAS operation requires that multiple parameter cases must be supported. For example, a full BLAS implementation of GEMM must allow nine parameter cases. This stems from the three accepted values for each of the `transA` and `transB` arguments that correspond to the two input matrices, A and B. In BLIS, GEMM must support 16 cases due to the extra conjugation-without-transposition option. The TRMM and TRSM operations must support 24 and 32 cases in BLAS and BLIS, respectively, due to the combinatorial impact of the `side`, `uplo`, `trans`, and `diag` parameters. It is easy to see how a BLAS-like library could quickly become unwieldy and difficult for developers to manage.

The BLIS framework allows level-3 operations to handle and implement most of their various parameter cases via the packing facility. But before discussing this further, we first broach the topic of how BLIS organizes its data. At higher levels of execution, the level-3 operands are expressed and tracked with "objects." This simple abstraction, implemented as C `structs`, allows the framework to express its algorithms at a high level.[17] These objects encapsulate nearly all properties of a matrix and are "tagged" in various ways depending on the operation's parameterization. For example, if a particular instance of GEMM requires that matrix A be used as if it were conjugated, the implementation toggles the conjugation bit of the object associated with the matrix. As one might expect, the actual elements of matrix A are never modified, and so the conjugation of the matrix object is implicit. Similarly, the object determines whether the matrix requires transposition[18], the location (offset) of its diagonal,

[16]While BLIS sequesters almost all non-portable code within the micro-kernels, these codes must still be implemented with care. How to optimally implement these micro-kernels is heavily dependent on features of the memory hierarchy and instruction set, and thus is well beyond the scope of this document.

[17]Unlike in `libflame`, the functions used to query and modify object properties are carefully implemented as C preprocessor macros. This technique avoids a great deal of potential function call overhead and also provides the compiler with more opportunities for optimization. Furthermore, when functions must be called, objects are passed by address rather than by value, thereby minimizing function call overhead. In this way, the BLIS framework affords its developers (and expert users) an object-based API that is lightweight and thus performance-friendly.

[18]Since BLIS supports both row and column strides, we may, alternatively, induce transpositions at any time before packing by swapping the row and column strides, swapping the length and width dimensions, and

whether that diagonal is implicitly unit, its structure (symmetric, Hermitian, triangular), and its storage: lower-stored, upper-stored, dense, or zero.

Now, when an object is about to be packed, its properties are inspected to determine exactly how the packing should unfold. Transposition, conjugation, and unit diagonalization can all be applied at virtually no cost during the packing (because the matrix data must be loaded from and/or stored to memory anyway). In general, upon completion, the packed matrix reflects *explicitly* what its corresponding unpacked object reflects *implicitly*. If the matrix is triangular and stored only in the upper or lower region, only that region will be packed. Similarly, if the matrix is Hermitian and "densification" is requested, the packed matrix will be packed so that the unstored region contains the conjugate-transpose of the stored region. (Packed objects also inherit diagonal offset information from their unpacked brethren, which greatly simplifies the handling of non-square blocks with structure.) The benefit of handling these parameter case combinations in the packing routine is that the lower-level codes, including the micro-kernel and, to some extent, the macro-kernels, are kept extraordinarily simple; that is, these lower-level codes need only support one or two very plain parameter cases (i.e., no transposition, no conjugation, non-unit diagonal, etc.).

We say "one or two" cases must be supported because, for some operations that assume operand structure, the BLIS framework explicitly implements both lower- and upper-stored cases (via separate variants and/or macro-kernels), either because a different region of the matrix is being updated (HERK, HER2K, SYRK, SYR2K) or because the algorithms promote movement through the operands in opposite directions (TRMM, TRSM). While this approach results in a modest increase in source code, these components of the framework are portable and would typically not require any modification when porting across architectures.

What remains? The `side` parameter, found in HEMM, SYMM, TRMM, and TRSM. The `side` parameter, since it describes the orientation of the matrices rather than the matrices themselves, is not embedded within the object, and so it remains a parameter of the operation throughout most layers of the BLIS framework. But, the `side` parameter need not be supported directly. Instead, in BLIS, one case is typically implemented (say, the left side case) and assumed by the mid- and lower-level algorithms. The right side case is handled early on at higher levels of the implementation; if the right side case of the operation is detected, it is induced by calling the left side case with transposed operands.[19]

5.1.3. Datatype. A prominent framework dimension we must manage is that of floating-point datatype. Specifically, we wish to minimize the amount of code that must reside within the framework (and the amount of code the expert must optimize) to support the floating-point datatypes expected by the community. BLAS supports operations on four standard datatypes: single-precision real, double-precision real, single-precision complex, and double-precision complex. This facet of the implementation is prominently encoded (usually) as the first character in the routine name: `s`, `d`, `c`, and `z`, respectively. For the level-3 BLAS, the implicit assumption is that the datatype of the operation corresponds to the datatype of all matrix operands. In BLIS, we instead associate an independent datatype property with each matrix object. So the question

---

if necessary, toggling the region stored and the diagonal offset. In some situations, we prefer this method because it allows us to further collapse the number of algorithms needed. TRMM serves as a good example of this, since some variants of this operation (those that partition matrix A) move in opposite directions for the lower- and upper-stored cases.

[19]In practice, a little more sophistication is sometimes employed than transposing every matrix operand. For example, with HEMM, the right side case is induced by toggling the conjugation of matrix A, which is mathematically equivalent to a transposition. (This holds in both the real and complex domains.)

becomes, how do we handle arbitrary combinations of domains and precisions among matrices?

Let us first handle the special case that merely mimics the datatype support of the BLAS (i.e., the case where all matrices contain elements of the same datatype). The sole noteworthy expansion of code that is proportional to the number of datatypes supported (in our case, four) is that of the GEMM micro-kernel. For each datatype supported, one micro-kernel must be provided (and optimized, if high performance is desired).[20] This is the *only* datatype-specific source code that is required of homogeneous-typed level-3 BLIS. All other level-3 codes, including high-level object-based algorithmic variants and mid-level macro-kernels, are datatype-agnostic.[21]

Now let us consider the cases where matrix datatypes are not all identical. We remind the reader that BLIS does not require the library developer to support these mixed-datatype cases. If this functionality is desired, however, very little effort is needed because mixed datatype support is largely handled by reusable framework code. When mixed-datatype operations require matrices to be typecast to a higher or lower precision, casting may be performed as needed when the matrices are packed. When domains must be mixed, the framework activates logic that utilizes existing same-datatype micro-kernels. to implement the desired mixed-domain operations in a way that avoids redundant flops (and memops).

5.1.4. Storage format. Recall that a central goal of BLIS is to support matrices stored not only in column-major order and row-major order, but also general stride.[22] Furthermore, we wish to allow operations to mix storage types among matrix operands. While this may appear to be an unmanageable number of cases to support, especially for three-operand operations such as GEMM, the impact on the level-3 BLIS framework is quite minimal. (In fact, there are situations where separate row and column strides actually simplifies the underlying implementation.) This expanded functionality is achieved by virtue of two aspects of the framework.

— Since input matrices are already universally packed to contiguous storage for performance reasons, we can easily design the packing facility to read these input operands according to their individual row and column strides.
— Generalized storage is supported for output operands because the BLIS micro-kernel interface is explicitly parameterized with row and column strides for the output matrix C (which need not be packed).

These two features combine to allow row storage, column storage, or general stride storage for any operand, as well as any combination across operands, for all supported operations.

5.1.5. Algorithm. As previously mentioned, BLIS provides multiple algorithmic variants for each operation supported. Each algorithmic variant is derived from a *loop invariant*, which, among other things, expresses the way matrix operands are partitioned and the direction(s) in which the variant moves through these matrices [Gunnels et al. 2001a; Gunnels and van de Geijn 2001; Bientinesi et al. 2005; van de Geijn and Quintana-Ortí 2008]. The variant set provided by BLIS for each level-3 operation can be thought of as "complete" in the sense that there exists at least one variant that partitions each dimension of the problem. For example, the framework provides

[20] If a fused GEMM-TRSM micro-kernel is desired, this code must also be instantiated for each datatype.
[21] Lower-level macro-kernels are kept datatype-agnostic in a manner similar to that of unblocked variants for level-1v, -1m, and -2 operations. This topic is discussed in Section 5.2.3
[22] Note that the former two storage layouts are simply special cases of general stride where the row stride is unit and the column stride is unit, respectively.

three basic blocked variants for GEMM: one for partitioning along each of the m, n, and k dimensions. (Typically, we omit from BLIS variants that traverse operands backwards when a forward-moving variant is available and applicable.) In a more sophisticated level-3 (or LAPACK-like) algorithm, one may have an operation with two or more subproblems, where each subproblem belongs to a different family of operations. This recursion may involve several levels of partitioning and blocking, usually to promote cache reuse and locality, but also sometimes for parallelism or out-of-core execution (or both). Some algorithms are more advantageous for some circumstances, and thus the ability to jump between them during recursive subproblems becomes key to enabling high performance. But being able to select a given compound (overall) algorithm requires the set of compound algorithms to exist (at runtime) in the first place. It comes as no surprise that coding, maintaining, and storing the various compound algorithms for each operation can become a chore.

BLIS solves this problem by borrowing a relatively novel construct unique to `libflame` which we call *control trees*. Control trees are based on the following fundamental observation: *If we supply a basic (but "complete") set of algorithmic variants for a given operation, as well as basic sets of variants for all operations on which it might depend, then one can specify the execution path of an arbitrarily-deep compound algorithm by using a tree.* This tree can be represented with a recursive data structure. Each node in this structure is typed according to the operation it represents, and encodes information (such as block size, algorithmic variant, and packing format) that parameterizes the implementation to be used. Interior nodes in the tree correspond to blocked algorithms that further partition the problem into smaller subproblems while leaf nodes correspond to unblocked codes that perform actual computation. Blocked algorithms, instead of calling a hard-coded implementation directly, call the operation's internal "back-end" (while passing in the corresponding sub-tree). This back-end function acts as a decoder that inspects the contents of the current control tree node and dispatches the prescribed blocked or unblocked implementation.[23] And since control trees are stateless objects, they are thread-safe. The tree nodes themselves can even be used to encode information about when to partition the problem for parallelism [Chan et al. 2008].

The key consequence of the control tree infrastructure is that it allows one to eliminate virtually *all* redundant algorithm codes from the framework. This is possible because the trees, which may be specified statically prior to compilation or built dynamically at run-time, allow the algorithm developer to compose entire algorithms from the basic building blocks of its subproblems. Another consequence of control trees is that once the basic variants are developed and coded, one can change the overall algorithm, block size (at any level), or packing format, by simply changing the control tree. Thus, in BLIS, blocked algorithmic variants capture only the essence of the *algorithm*, while the *implementation* of subproblems are determined by the tree and not realized until it is decoded at run-time by the back-end. If new blocked variants (or macro-kernels) are added to the framework, their support is added by making the internal back-end code aware of these new routines. They are then available for use by simply specifying their usage via a control tree.

[23] It would reasonable for the reader to harbor some skepticism that control trees would not incur significant overhead during execution. Specifically, for every blocked algorithm subproblem, a control tree-based solution incurs an additional function call to the subproblem operation's internal back-end routine, whereas a conventional algorithm would encode the desired subroutine call directly into the body of the algorithm implementation. In our experience, this overhead tends to be negligible since it is amortized over a large amount of computation. The level-3 performance results in Section 7 confirm that control tree overhead can be quite minimal, if not indistinguishable.

And so, by expressing the execution path (and key runtime parameters) of the overall level-3 algorithm with a tree structure, we allow arbitrary algorithms to be composed from a small set of variants, while simultaneously keeping the source code and object code footprints quite manageable.

5.2. Level-2, Level-1v, Level-1m, and Level-1f operations

We now discuss how each of the five dimensions of generality are achieved for the level-1v, level-1m, level-1f, and level-2 operations.

5.2.1. Operation. Recall that with level-3 operations, we were able to greatly simplify the implementations by expressing the operations in terms of a GEMM micro-kernel, with the remaining differences expressed via either the packing facility or the operation's macro-kernel, or a combination of the two. However, level-2 (and lower) operations cannot afford to be implemented with a pack stage if performance is to be maximized. Indeed, performance is already hindered by the fact that the ratio of flops to memops for these operations is effectively $\mathcal{O}(1)$. Thus, there is not nearly as much opportunity for reuse of algorithms between level-2 operations.[24] However, there is a silver lining. As mentioned in Sections 4.3 and 4.4, all level-2 operations can be realized in optimized form by simply optimizing the level-1f kernels shown in Figure 6, and similarly, a level-1m implementation can be accelerated by optimizing its corresponding level-1v operation. And so the amount of code that must be customized to achieve high performance for a given architecture, while greater than that of level-3 BLIS, remains relatively small.

It should be noted that, because BLIS exposes multiple layers, we can easily implement higher-level operations in terms of lower-level codes that forgo error checking, thereby avoiding redundant error checking. For example, level-1m BLIS operations are implemented in terms of level-1v kernels; however, these kernels are not accessed via the same APIs that an end-user would call. Instead, we use expert-level interfaces to access level-1v codes that skip error checking altogether. In this way, function parameters and other properties of the problem, such as operand dimensions, can be checked once up-front.

5.2.2. Parameter case. The number of parameter cases needed to fully support level-2 and lower operations is noticeably smaller than that of level-3 operations. For example, unlike their level-3 analogues, HEMV, TRMV, and TRSV have no `side` parameter. And while some operations still sometimes offer the option of transposing an operand or computing with only the lower- or upper-stored region of a matrix, both of these options can usually be handled quite naturally by manipulating the row and column strides of the matrix object. For example, a matrix transpose is typically induced by swapping the row and column strides (and sometimes making adjustments to dimensions, as with the case of GEMV). Furthermore, level-2 BLIS operations may be implemented to explicitly support only the lower-stored case, but still indirectly support the upper-stored case by internally inducing a matrix transposition prior to executing any underlying level-1v or -1f kernels. In the case of HEMV, HER, and HER2, switching the lower/upper storage parameter corresponds to toggling one or more conjugation parameters (since $\text{LOWERTRIANGLE}(H) = \text{UPPERTRIANGLE}(\bar{H})$, where H is Hermitian). Scalar-level conjugation is implemented directly within the underlying level-1v

[24]Nevertheless, careful coding allows us to consolidate some operation implementations. For example, we implement real SYR and complex HER as the same algorithm. This is possible since the Hermitian transpose in the case of complex HER reduces to a transpose in the real domain. Furthermore, a simple switch that disables the conjugation component of the Hermitian transpose allows us to implement complex SYR using the same algorithm codes. A similar consolidation may be performed for SYR2/HER2 and SYMV/HEMV.

or level-1f kernels, which allows algorithms to be implemented without the need for temporarily conjugated (and thread-unsafe) operands. Support for arbitrary diagonal offsets for lower- and upper-stored matrices is implemented uniformly across all level-1m operations, and in such a way that factors out the logic that adjusts loop bounds and index offsets to account for the location of the diagonal.

5.2.3. Datatype. By default, the BLIS framework provides portable, C-based implementations of all level-1v, level-1m, and Level-2 operations, each with full datatype support. BLAS achieves this by providing a separate code for each datatype supported, despite these codes differing in only slight ways. BLIS avoids this redundancy by using the C preprocessor to employ macro-based templates whenever unblocked codes are written (including the macro-kernels used by level-3 operations). Thus, the framework stores only one datatype-agnostic copy of each variant; the datatype-specific codes are then instantiated by the preprocessor at compile-time. This technique drastically reduces the source code (though not the object code) footprint of the framework, which in turn makes the framework much easier to modify and maintain.

These macro-based templates also allow one to capture mixed datatype algorithms by specifying the types of each operand separately. If and when the library developer configures BLIS to allow mixed domain and/or precision, these additional datatype combinations are instantiated automatically when the library is built. However, at first all implementations, whether of homogeneous or mixed datatypes, will be unoptimized. If the developer chooses to support only operands of homogeneous datatypes, then he or she need only optimize the level-1v and/or level-1f kernels for homogeneous datatypes (i.e., four datatype instances of each operation, assuming all four floating-point datatypes are to be supported). If the developer wishes to support datatype mixing, then he or she will need to provide additional level-1v and/or level-1f kernels that take operands of mixed types.[25]

5.2.4. Storage format. Generalized storage is supported for level-1m and -2 operations just as with level-3 operations. The mechanism, however, is different. Here, we rely upon underlying kernels to support general stride. Recall that the level-1m and -2 operations are implemented in terms of level-1v and -1f kernels; so, as long as the latter are implemented to support both contiguous (row- or column-stored) and general stride storage, the former will exhibit this support as well. Admittedly, *supporting* contiguous as well as general stride cases is distinct from *optimizing* those cases, as kernel developers often allow the assumption of contiguous storage to influence their kernel implementation (specifically, their choice of assembly instructions). Thus, the library developer will have to weigh whether including optimizations for both cases is important for his or her users.

5.2.5. Algorithm. The set of variants for level-2 and lower operations is much smaller due to the reduced number of possible matrix and vector partitionings. For a typical level-1m operation such as AXPYM, BLIS needs to provide only one variant, because the default runtime system can manipulate matrix objects (e.g. strides) to allow unit stride through memory whenever possible. For a typical level-2 operation, two or more variants are provided. This allows the runtime system to choose the best variant based on the exact nature of the problem, taking in account matrix storage as well as transposition. For example, GEMV invokes a DOTXV-based implementation if matrix A is

[25]Unfortunately, this aspect of the framework—whereby the developer must provide one level-1v/level-1f kernel for every datatype combination desired—is probably the weakest part of the framework. But, absent the developer (and his or her users) tolerating a potentially significant cost in terms of workspace and memory copy overhead incurred by a runtime typecast/packing facility, we see no other way of providing this functionality in optimized form.

Object of interest	Byte footprint			
	BLIS	OpenBLAS	ATLAS	MKL
Executable that calls DGEMM	337K	33K	2.11M	2.80M
... and also DSYMM	341K	42K	2.12M	2.94M
... and also DSYRK	367K	56K	2.12M	3.16M
... and also DSYR2K	373K	72K	2.13M	3.22M
... and also DTRMM	422K	142K	2.17M	4.95M
... and also DTRSM	475K	210K	2.20M	5.57M
... and also ZGEMM	475K	251K	3.12M	7.98M
... and also ZHEMM	475K	259K	3.14M	8.03M
... and also ZHERK	475K	276K	3.16M	8.12M
... and also ZHER2K	475K	297K	3.17M	8.19M
... and also ZTRMM	475K	394K	3.22M	9.72M
... and also ZTRSM	475K	497K	3.27M	10.86M
Library archive	2.17M	6.22M	11.81M	?[26]
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. 12. Various manifestations of library footprints when statically linked to each of: BLIS 0.1.0-20, OpenBLAS 0.2.6, ATLAS 3.10.1, and MKL 11.2. Here, "K" and "M" indicate 1024 and 1048576 bytes, respectively.

row-stored and not in need of a transpose, or if A is column-stored and in need of a transpose, while an AXPYV-based variant is preferred for the two remaining cases. For general stride cases, unit stride cannot be achieved, and thus by default the BLIS framework chooses a DOTXV-based variant, if one is available, since it incurs fewer memory operations while performing the same number of flops.

5.3. Summary remarks

We have now discussed a wide breadth of implementation techniques that we employ in BLIS which help us keep the framework as small, manageable, and flexible as possible. It should be noted that *some* increase in code is sustained even when employing the methods described above. But our primary goal here was to avoid the situation whereby increasing functionality along one dimension would *proportionally* inflate the resulting library's code size. The only dimension along which we clearly fail at this goal is that of datatype instantiation (in the case of mixed-domain and mixed-precision) for level-1v and level-1f kernels.

The BLIS framework employs other more mundane techniques that contribute generally to its usability and maintainability, which we choose not to document here for space reasons.

In the next section, we will quantify the object code size of BLIS and compare it to that of other BLAS libraries.

6. LIBRARY FOOTPRINT

[26] 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

A concern with BLAS implementations can be their byte footprints. 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 12, which summarizes byte footprints of various executables and libraries when statically linking to BLIS 0.1.0-20, OpenBLAS 0.2.6[27], ATLAS 3.10.1, and MKL 11.0 Update 4. The data in Figure 12 was gathered on the same architecture discussed in Section 7.1.

We can see that when linking BLIS to a simple test driver that calls only DGEMM, the resulting executable is 337Kbytes in size. Executables linked to OpenBLAS, ATLAS, and MKL are 33Kbytes, 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. Specifically, adding calls to the other five double-precision real level-3 operations (DSYMM, DSYRK, DSYR2K, DTRMM, and DTRSM) results in 138Kbytes of additional object code when linking to BLIS, 177Kbytes when linking to OpenBLAS, 106Kbytes 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 287Kbytes, 1.07Mbytes, 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[28] is smallest at 2.17Mbytes, with OpenBLAS and and ATLAS consuming 6.22Mbytes and 11.81Mbytes, 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 12 list the total memory footprint of running processes when executing DGEMM for square problem sizes of 100 and 4000. For small problem sizes,

---

processing functions (such as FFT). Thus, it would be difficult to estimate the size of only the object code needed by the BLAS.

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

[28]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 2.24Mbytes. This would also increase (by approximately the same amount) the sizes of the BLIS-linked executables listed in the first 12 rows of Figure 12.

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, in summary, 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. PERFORMANCE

In this section we present performance of various level-2 and level-3 BLIS operations. Comparisons are made with corresponding BLAS implementations of various libraries.

The platform chosen for the performance experiments was, admittedly, an old architecture. We approached the development of BLIS as a scientific experiment: Decades of research by various researchers on the implementation of BLAS libraries provided experimental and theoretical insights. From this, we conjectured how the framework should be designed. So as to leave the evaluation on modern architectures as the ultimate verification that we were on to something, we developed the framework on a relatively old architecture. The preliminary evaluation of the framework on that old architecture is given in this paper. Other papers, to be published in the future, will provide additional evidence.

7.1. Platform and implementation details

All experiments reported in this paper were performed on a single core of a Dell PowerEdge R900 server consisting of four six-core Intel Xeon "Dunnington" X7460 processors. Each core provides a peak performance of 10.64 GFLOPS. Performance experiments were gathered under the GNU/Linux 2.6.18 operating system. Source code was compiled by the GNU C compiler (`gcc`), version 4.8.2. All experiments were performed in double-precision floating-point arithmetic on randomized, column-stored real domain matrices.

The version of BLIS tested was 0.1.0-20.[29] We compare BLIS performance to three other single-threaded implementations: OpenBLAS 0.2.6, ATLAS 3.10.1 and MKL 11.0 Update 4. The level-3 BLIS implementations are based on a hand-coded micro-kernel similar to the corresponding inner-most loop of one of the kernels found within OpenBLAS. This micro-kernel was expressed in GNU extended inline assembly syntax and coded to leverage vector floating-point instructions. Level-1 and Level-1f BLIS kernels were coded at a somewhat higher level, using SSE vector intrinsics.

In the following subsections, we discuss the performance results of Figures 13 and 14. These graphs show performance, measured in GFLOPS (10^9 floating-point operations per second), as a function of matrix operand (problem) size, where each data point shows the best of three trials. For operations with matrix operands that have both m and n dimensions, we fix $m = n$, and for those that have a k dimension (i.e. GEMM, SYRK, and SYR2K) we fix $m = n = k$. All graphs are scaled such that the maximum y-axis value is equal to the peak performance of a single core, 10.64 GFLOPS.

The underlying micro-kernel utilized by level-3 operations reported in Figure 13 used register block sizes $m_r = n_r = 4$, while the higher-level blocked algorithms used cache block sizes $m_c = 384$ and $k_c = 384$. Similarly, the level-1f kernels AXPYF, DOTXF, and DOTXAXPYF used by level-2 operations reported in Figure 14 were implemented with fusing factors of $f = 4$.

[29] This version of BLIS may also be uniquely identified by the first 12 digits of its `git` "commit" (SHA1 hash) number: f60c8adc2f61.

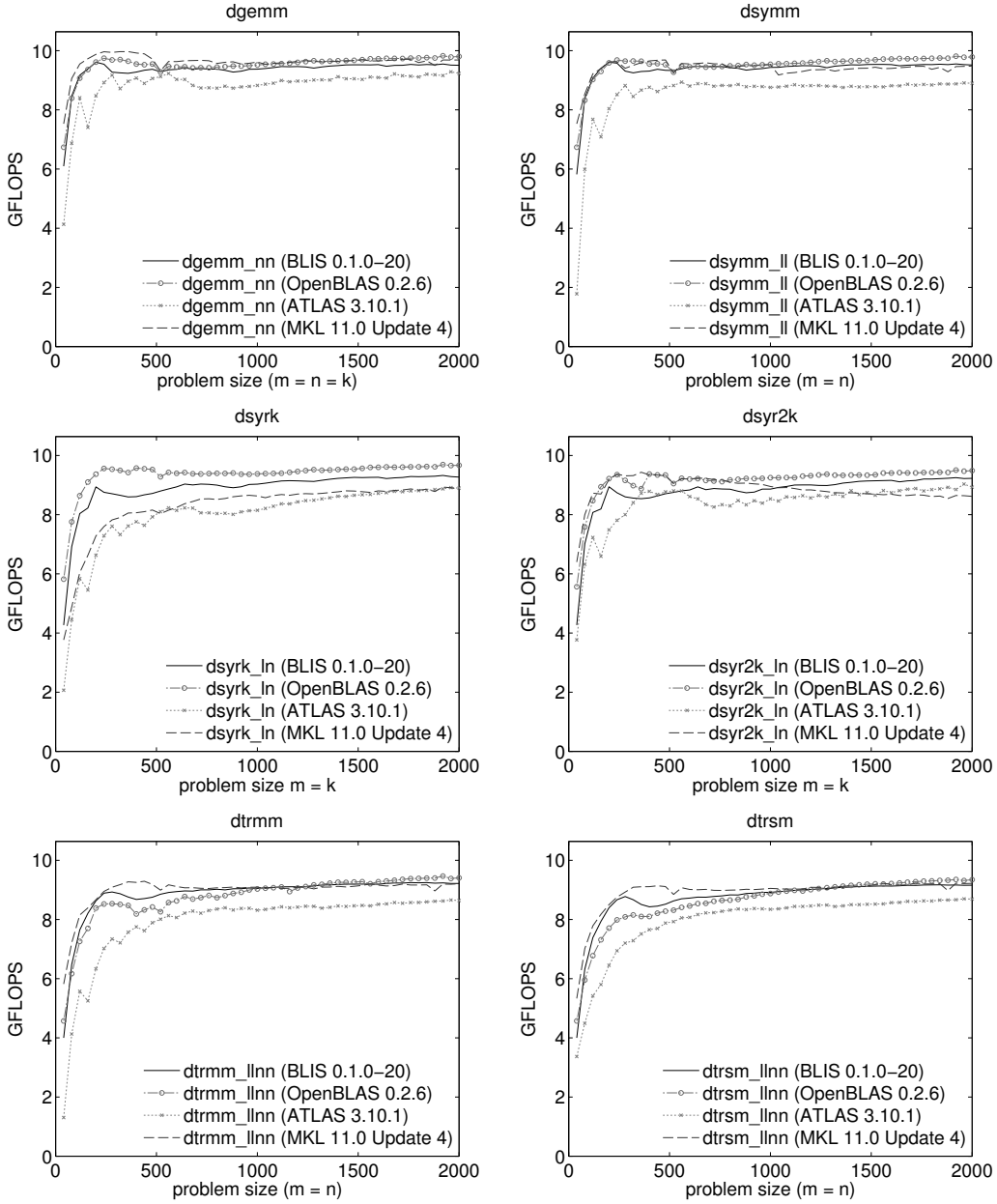

Fig. 13. Performance of various implementations of double-precision real GEMM (top-left), SYMM (top-right), SYRK (center-left), SYR2K (center-right), TRMM (bottom-left), and TRSM (bottom-right), on a single core of a 2.66GHz Intel Xeon 7400 system.

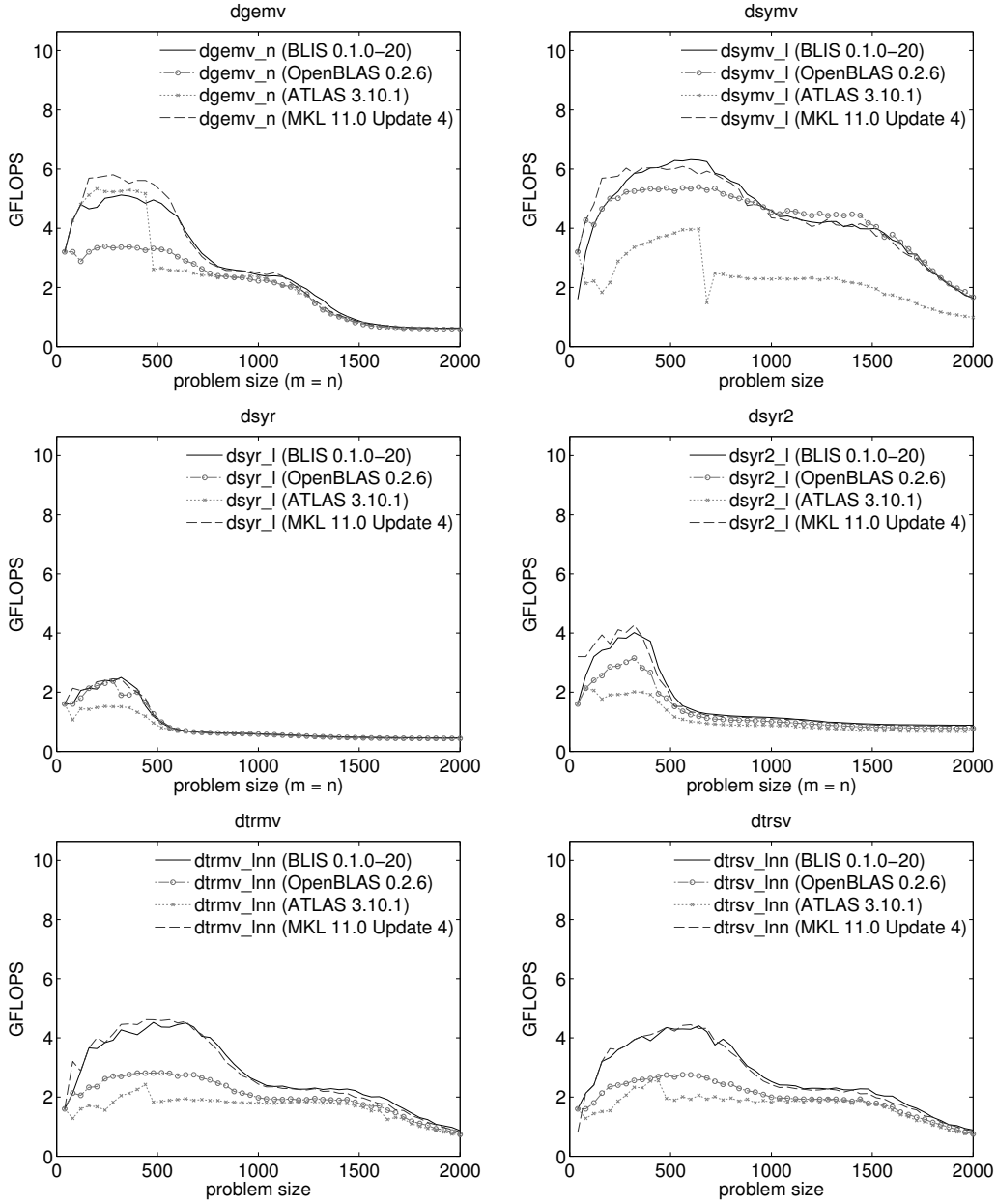

Fig. 14. Performance of various implementations of double-precision real GEMV (top-left), SYMV (top-right), SYR (center-left), SYR2 (center-right), TRMV (bottom-left), and TRSV (bottom-right), on a single core of a 2.66GHz Intel Xeon 7400 system.

The register block sizes m_r and n_r and cache block sizes m_c and k_c were chosen to be identical to the corresponding block sizes used by OpenBLAS. How these block sizes are optimally chosen is beyond the scope of this paper; however, we suspect that this topic is fertile ground for future research. For the DOTXAXPYF kernel, we chose f to be as large as possible such that loads into and stores out of registers are minimized (i.e., such that all values neccessary for a given iteration can be read and used from registers before being discarded or written back to main memory). We then used this value $f = 4$ for DOTXF and AXPYF as well, even though these kernels' simpler loop bodies would have allowed us to employ larger fusing factors.[30]

7.2. General results

Figure 13 shows results for the double-precision real instances of six level-3 operations: GEMM, SYMM, SYRK, SYR2K, TRMM, and TRSM. In all six graphs, the performance of the BLIS implementation can be seen to be highly competitive with existing implementations, including those provided by the Intel MKL library.

Similarly, Figure 14 shows performance for the double-precision real instances of six level-2 operations: GEMV, SYMV, SYR, SYR2, TRMV, and TRSV. (These are the level-2 analogues of the operations reported on in Figure 13.) Once again, observed performance of the implementations instantiated via the BLIS framework is comparable to that of codes available via OpenBLAS, ATLAS, and MKL.

These results suggest that the BLIS framework is capable of facilitating competitive performance for BLAS-like operations. Importantly, with the exception of TRSM, the BLIS performance results shown in Figure 13 were achieved by optimizing *only* the micro-kernel illustrated in Figure 11; all other aspects of the implementations employed portable component codes from the framework. Similarly, the optimizations that enabled the BLIS performance results in Figure 14 were limited to those of the level-1 and level-1f kernels shown in the right-hand column of Figure 9.

7.3. Benefit of fused GEMM-TRSM micro-kernel

As mentioned in Footnote 15, near-optimal TRSM performance requires more than just an optimized GEMM micro-kernel. TRSM is inherently dissimilar from the other level-3 operations in that it cannot be entirely expressed in terms of a simple matrix-matrix multiplication. While the bulk of its computation can be cast in terms of GEMM, it also requires a small triangular solve subproblem. While this TRSM subproblem constitutes a very small fraction of the total flops executed, it incurs extra memory operations that could otherwise be avoided altogether if the GEMM and TRSM subproblems were fused at the register level.

The BLIS implementation of TRSM reported on in Figure 13 uses a fused GEMM-TRSM micro-kernel, and thus is more fully-optimized than an implementation that leverages only the GEMM micro-kernel. How much TRSM performance is gained by crafting this specialized micro-kernel? To answer this question, we tested two additional implementations in addition to the one shown in Figure 13. These three implementations are described as follows:

— **BLIS with fused GEMM-TRSM.** This implementation fuses the GEMM micro-kernel with the small TRSM subproblem that follows it so that the current $m_r \times n_r$ submatrix of B is held in registers rather than being written out to memory and immediately

[30]It should be noted that the register set would not have allowed much of an increase in the fusing factors for these two operations—perhaps to $f = 8$ at most. And even then, the improvement, in terms of avoided memory operations, is relatively small and diminishes quickly for larger values of f.

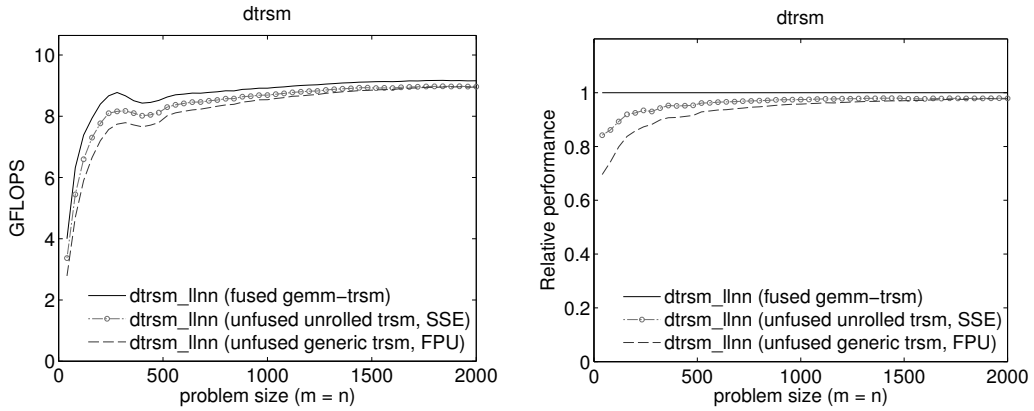

Fig. 15. Left: Performance of BLIS TRSM with four levels of optimization: fully optimized via a fused GEMM-TRSM micro-kernel (as shown in the bottom-right graph of Figure 13); partially optimized via an unfused and unrolled TRSM micro-kernel compiled with scalar SSE instructions (which are automatically emitted by the compiler via `-mfpmath=sse`); and unoptimized via an unfused and generic (loop-based) TRSM compiled with traditional FPU-based instructions. Right: Performance of the latter three implementations relative to that of the first.

read back in. This fused code is written with GNU extended inline assembly syntax and utilizes vector SSE instructions.

— **BLIS with unfused unrolled** TRSM **via SSE.** This implementation of TRSM separates the GEMM micro-kernel call from the smallest TRSM subproblem. This small TRSM subproblem is coded in C, manually unrolled, and then compiled with the `-mfpmath=sse` compiler option, which causes `gcc` to emit scalar SSE instructions.

— **BLIS with unfused generic** TRSM **via FPU.** This is similar to the above implementation, except that rather than being unrolled, the TRSM subproblem is expressed in terms of loops over the register block sizes. Additionally, the code is compiled with the `-mfpmath=387` compiler option, which causes `gcc` to emit instructions that utilize only the standard floating-point unit present on Intel x86 processors. Since this micro-kernel is not unrolled, it can be used unmodified with any register block sizes, making it completely generic.

We would expect the SSE-based unfused implementation to outperform the generic FPU-based code, with both underperforming the implementation which employs a fused GEMM-TRSM micro-kernel. Figure 15 reports performance consistent with this prediction. The key take-away from this graph is that while the fused implementation offers somewhat higher performance, the performance penalty incurred by forgoing this level of optimization is not prohibitive, especially when using compiler-generated scalar SSE instructions on manually unrolled C code, and/or when computing upon larger matrices.

7.4. Potential for automation

Auto-tuning of level-3 BLAS started with IBM's effort to provide BLAS for the Power2 architecture [Agarwal et al. 1994]. That effort showed that high-performance implementations could be achieved in a high-level language (Fortran). The PHiPAC project [Bilmes et al. 1997] picked up on this, but introduced the idea of generating highly efficient C code, including automatic tuning of block sizes and loop orderings. This was then pushed to the next level by the ATLAS project, which reduced the search space by focusing on automatically tuning inner kernels and block sizes. Inner kernels

are now hand-written in assembly code and catalogued within the system so that the appropriate kernel can be chosen later, when the library is configured and built.

The reader may notice that we do not discuss auto-tuning. Yotov et al. showed that architecture parameters and characteristics can be used to analytically derive the tuning parameters that ATLAS determines through exhaustive search [Yotov et al. 2005]. Further evidence that auto-tuning is not necessary comes from the fact that Goto did not auto-tune (and achieves better performance than ATLAS) for all architectures he targeted [Goto and van de Geijn 2008a; 2008b], nor did Volkov auto-tune when optimizing for GPUs [Volkov and Demmel 2008]. We believe that the BLIS framework structures the implementation effort in a way that exposes fundamental techniques and hence allows architectural parameters and insight to be used in lieu of auto-tuning. Nonetheless, one could at some point add auto-tuning to the framework.

A related topic of research is that of automatically generating linear algebra libraries from domain and hardware specifications. We believe that the BLIS framework isolates architecture specifics to the point where a tool like Spiral [Püschel et al. 2005] could be used to generate micro-kernels. Indeed, one of the motivations for creating the BLIS framework was to distill the dense linear algebra software stack into basic facilities (i.e., computational and packing kernels) so that Design by Transformation [Marker et al. 2012] can encode expert knowledge about the library operations and then generate highly optimized higher-level functionality. We intend to investigate this further in future work. Other groups, such as the Build to Order BLAS project [Siek et al. 2008; Belter et al. 2009], may also benefit from the kernel and/or layering that BLIS exposes.

8. CONCLUSIONS

In this paper, we have proposed a new framework for rapidly instantiating BLAS-like libraries. This framework addresses several shortcomings of the original BLAS interface design (while also providing backward compatibility), expands functionality, and most importantly, isolates most architecture-sensitive details to a few relatively small kernels, which, when optimized, immediately enable high-performance across virtually all supported operations. Experimental level-2 and level-3 performance was observed to be competitive with two highly-regarded open source libraries, as well as the famously optimized commercial solution from Intel. Evidence of BLIS's code reuse and compactness was also given by comparing library and executable byte sizes across these same libraries.

This work lays the foundation for future efforts:

— At the time that this paper was first written, adding multithreading to the BLIS framework was not yet explored. Focusing on level-3 operations, the thought was that BLIS exposes many loops and that there are therefore many opportunities for parallelism. In the evaluation [Van Zee et al. 2013] that ported BLIS to a multitude of then-current architectures, some experimentation of how to extract parallelism was explored, with great success. These techniques for many-core threading were further generalized and described in [Smith et al. 2014].
  Now one of our highest priorities is to add systematic support for multithreading (via either POSIX threads or OpenMP). We envision a multithreaded BLIS framework that exports *simultaneous* access to sequential and parallel implementations, where the number of threads for the latter can be changed by the user at runtime. This would allow applications to control precisely how much parallelism is achieved within BLIS at any given time.
— While the BLIS framework has been ported to several architectures, we wish to gain exposure to even more types of hardware, including GPUs. These experiences will

help us further refine and generalize the framework, and streamline the process of instantiating BLAS-like libraries on other similarly exotic systems.
— As mentioned previously, given that BLIS isolates performance-sensitive code to a few simple kernels, the framework may aid those who wish to automate the generation of high-performance linear algebra libraries from domain and hardware specifications [Püschel et al. 2005; Marker et al. 2012; Siek et al. 2008; Belter et al. 2009].
— As computing systems become less reliable, whether because of quantum physical effects, power consumption restrictions, or outright power failures, the community may become increasingly interested in adding algorithmic fault-tolerance to the BLAS (or BLAS-equivalent) layer of the dense linear algebra software stack [Gunnels et al. 2001b; Huang and Abraham 1984]. We plan to investigate the suitability of BLIS as a vehicle to provide such fault-tolerance.
— We may also look toward adding support for extended precision arithmetic.

Thus, the contributions of this work provide several avenues for further research.

Acknowledgements

We kindly thank Lee Killough, Bryan Marker, Francisco Igual, Tze Meng Low, and Rhys Ulerich for participating in early discussions on design. We also thank Vernon Austel, and John Gunnels, Francisco Igual, Michael Kistler, Mikhail Smelyanskiy, Tyler Smith, and Xianyi Zhang for porting BLIS to more modern architectures, as reported in [Van Zee et al. 2013]. Finally, we thank John Gunnels for offering thoughtful feedback on an advanced draft of the paper. We also thank the Texas Advanced Computing Center for providing access to the the Intel Xeon "Dunnington" system on which performance data was gathered.

This research was partially sponsored by grants from Microsoft and the National Science Foundation (Awards CCF-0917167 and ACI-1148125).

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).

REFERENCES

AGARWAL, R., GUSTAVSON, F., AND ZUBAIR, M. 1994. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38,* 5.

AGULLO, E., BOUWMEESTER, H., DONGARRA, J., KURZAK, J., LANGOU, J., AND ROSENBERG, L. 2010. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures.

AGULLO, E., DEMMEL, J., DONGARRA, J., HADRI, B., KURZAK, J., LANGOU, J., LTAIEF, H., LUSZCZEK, P., AND TOMOV, S. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series 180*.

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

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.

BELTER, G., JESSUP, E. R., KARLIN, I., AND SIEK, J. G. 2009. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. 59:1–59:12.

BIENTINESI, P., GUNNELS, J. A., MYERS, M. E., QUINTANA-ORTÍ, E. S., AND VAN DE GEIJN, R. A. 2005. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft. 31,* 1, 1–26.

BILMES, J., ASANOVIĆ, K., WHYE CHIN, C., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*. Vienna, Austria.

BISCHOF, C. AND VAN LOAN, C. 1987. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput. 8,* 1, s2–s13.

BLAS 2012. http://www.netlib.org/blas/.

BLAST 2002. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing 16,* 1.

CHAN, E., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2007. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, San Diego, CA, USA, 116–125.

CHAN, E., VAN ZEE, F. G., BIENTINESI, P., QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., AND VAN DE GEIJN, R. 2008. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and practices of parallel programming (PPoPP'08)*. 123–132.

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Comput. Soc. Press, 120–127.

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.

DONGARRA, J. J., VAN DE GEIJN, R. A., AND WHALEY, R. C. 1993. Two dimensional basic linear algebra communication subprograms. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*.

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.

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001a. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft. 27,* 4, 422–455.

GUNNELS, J. A., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001b. 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. AND VAN DE GEIJN, R. A. 2001. Formal methods for high-performance linear algebra libraries. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press, 193–210.

HOWELL, G. W., DEMMEL, J. W., FULTON, C. T., HAMMARLING, S., AND MARMOL, K. 2008. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software 34,* 3, 14:1–14:33.

HUANG, K. AND ABRAHAM, J. 1984. Algorithm–based fault tolerance for matrix operations. *IEEE Trans. on Computers 33,* 6, 518–528.

IBM. 2012. Engineering and Scientific Subroutine Library. http://www.ibm.com/systems/software/essl/.

INTEL. 2012. Math Kernel Library. http://developer.intel.com/software/products/mkl/.

JOFFRAIN, T., LOW, T. M., QUINTANA-ORTÍ, E. S., VAN DE GEIJN, R., AND VAN ZEE, F. 2006. Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software 32,* 2, 169–179.

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.

MARKER, B., POULSON, J., BATORY, D., AND VAN DE GEIJN, R. 2012. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *International Workshop on Automatic Performance Tuning (iWAPT2012). Proceedings of VECPAR 2012 Conference*.

MOLER, C., LITTLE, J., AND BANGERT, S. 1987. *Pro-Matlab, User's Guide*. The Mathworks, Inc.

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

PEDRAM, A., GERSTLAUER, A., AND VAN DE GEIJN, R. A. 2012a. On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. *International Symposium on Computer Architecture and High Performance Computing*, 19–26.

PEDRAM, A., VAN DE GEIJN, R. A., AND GERSTLAUER, A. 2012b. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers 61,* 12, 1724–1736.

POULSON, J., MARKER, B., VAN DE GEIJN, R. A., HAMMOND, J. R., AND ROMERO, N. A. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw. 39,* 2, 13:1–13:24.

PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GACIC, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93,* 2, 232–275.

QUINTANA-ORTÍ, G., QUINTANA-ORTÍ, E. S., VAN DE GEIJN, R. A., VAN ZEE, F. G., AND CHAN, E. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw. 36,* 3, 14:1–14:26.

SCHATZ, M. D., LOW, T. M., VAN DE GEIJN, R. A., AND TAMARA G. KOLDA, F. W. N. . 2012. Exploiting symmetry in tensors for high performance: an initial study. Technical Report TR-12-33, The University of Texas at Austin, Department of Computer Sciences. December.

SCHREIBER, R. AND VAN LOAN, C. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput. 10,* 1, 53–57.

SIEK, J. G., KARLIN, I., AND JESSUP, E. R. 2008. Build to order linear algebra kernels. In *International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008)*. 1–8.

SMITH, T. M., VAN DE GEIJN, R. A., SMELYANSKIY, M., HAMMOND, J. R., AND VAN ZEE, F. G. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 28th International Parallel & Distributed Processing Symposium (IPDPS)*. To appear.

SOLOMONIK, E., HAMMOND, J., AND DEMMEL, J. 2012. A preliminary analysis of Cyclops Tensor Framework. Tech. Rep. UCB/EECS-2012-29, EECS Department, University of California, Berkeley. Mar.

VAN DE GEIJN, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

VAN DE GEIJN, R. A. AND QUINTANA-ORTÍ, E. S. 2008. *The Science of Programming Matrix Computations*. `www.lulu.com/contents/contents/1911788/`.

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., SMITH, T., IGUAL, F. D., SMELYANSKIY, M., ZHANG, X., KISTLER, M., AUSTEL, V., GUNNELS, J., LOW, T. M., MARKER, B., KILLOUGH, L., AND VAN DE GEIJN, R. A. 2013. Implementing level-3 BLAS with BLIS: Early experience, FLAME Working Note #69. Technical Report TR-13-03, The University of Texas at Austin, Department of Computer Sciences. April. Submitted to ACM TOMS.

VAN ZEE, F. G., VAN DE GEIJN, R. A., AND QUINTANA-ORTÍ, G. 2012. Restructuring the tridiagonal and bidiagonal QR algorithms for performance. *ACM Trans. Math. Soft.*. submitted.

VAN ZEE, F. G., VAN DE GEIJN, R. A., QUINTANA-ORTÍ, G., AND ELIZONDO, G. J. 2012. Families of algorithms for reducing a matrix to condensed form. *ACM Trans. Math. Soft. 39,* 1, 2:1–2:32.

VOLKOV, V. AND DEMMEL, J. 2008. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley. May.

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

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.