PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP

JACK DONGARRA, MARK GATES, AZZAM HAIDAR, JAKUB KURZAK, PIOTR LUSZCZEK, PANRUO WU, ICHITARO YAMAZAKI, and ASIM YARKHAN,
University of Tennessee, USA
MAKSIMS ABALENKOVS, NEGIN BAGHERPOUR, SVEN HAMMARLING, JAKUB ŠÍSTEK, DAVID STEVENS, and MAWUSSI ZOUNON,
The University of Manchester, UK
SAMUEL D. RELTON, The University of Leeds, UK

The recent version of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) library is based on tasks with dependencies from the OpenMP standard. The main functionality of the library is presented. Extensive benchmarks are targeted on three recent multicore and manycore architectures, namely, an Intel Xeon, Intel Xeon Phi, and IBM POWER 8 processors.

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; *Solvers*; • **Computing methodologies** → *Shared memory algorithms*;

Additional Key Words and Phrases: Numerical linear algebra libraries, tile algorithms, task-based programming, multicore processors, OpenMP, PLASMA

ACM Reference format:
Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Softw.* 45, 2, Article 16 (April 2019), 35 pages.
https://doi.org/10.1145/3264491

The authors thank the Intel Corporation for their generous hardware donation and continuous financial support. They also thank the Oak Ridge Leadership Computing Facility for providing access to the POWER8 system. This work has been supported by the National Science Foundation under Grants No. CCF-1339822 (SILAS) and No. CCF-1527706 (DARE), by the European Commission H2020 Projects No. 671633 (NLAFET) and No. 671602 (INTERTWinE), and by the Engineering and Physical Sciences Research Council Project No. EP/M01147X/1 (SERT).
Authors' addresses: J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, I. Yamazaki, and A. YarKhan, University of Tennessee, Department of Electrical Engineering & Computer Science, 1122 Volunteer Blvd. Suite 203, Knoxville, TN 37996-3450, USA; emails: {dongarra, mgates3, haidar, kurzak, luszczek, iyamazak, yarkhan}@icl.utk.edu; P. Wu, University of Houston, Department of Computer Science, 3551 Cullen Blvd, Houston, TX 77204, USA; email: pwu7@uh.edu; M. Abalenkovs, N. Bagherpour, S. Hammarling, and D. Stevens, The University of Manchester, School of Mathematics, Manchester, M13 9PL, UK; emails: m.abalenkovs@manchester.ac.uk, n.bagherpour@gmail.com, sven.hammarling@btinternet.com, david.stevens@manchester.ac.uk; J. Šístek, Institute of Mathematics of the Czech Academy of Sciences, Žitná 25, 115 67 Prague, Czech Republic; email: sistek@math.cas.cz; M. Zounon, Numerical Algorithms Group (NAG), Manchester One, 53 Portland Street Manchester, M1 3LD, UK; email: mawussi.zounon@nag.co.uk; S. D. Relton, The University of Leeds, Institute of Health Sciences, Leeds, LS2 9LJ, UK; email: s.d.relton@leeds.ac.uk.

1 INTRODUCTION

The *Parallel Linear Algebra Software for Multicore Architectures* (PLASMA) numerical library is a dense linear algebra package at the forefront of multicore computing. PLASMA has been a response to the advent of multicore processors, proclaimed in the prominent article by Sutter (2005). At that time, it became apparent that both LAPACK[1] (Anderson et al. 1999) and ScaLAPACK[2] (Blackford et al. 1997) were ill suited for efficient multicore execution. Initial work focused on efficient multithreading of standard dense linear algebra algorithms (LU with partial pivoting, Cholesky, QR) using the canonical, column-major, data layout of LAPACK (Kurzak and Dongarra 2006). The debut of the STI Cell processor in 2006 pushed the developments in new directions.

The most influential aspect of the STI Cell was the memory architecture based on software controlled caches. This motivated tiling of the input matrices for efficient communication between the main memory and the caches. The memory architecture, and its high internal bandwidth, promoted systolic algorithms with high degrees of pipelining. At the same time, the 14× performance advantage of single precision over double precision, in the original Cell design, stimulated the development of mixed precision algorithms. Notable papers from that era highlighted tiling, scheduling, and mixed precision iterative refinement (Buttari et al. 2007; Gustavson et al. 2012; Kurzak et al. 2008; Kurzak and Dongarra 2007, 2009; Langou et al. 2006). All these artifacts influenced the design of the PLASMA library in one form or another.

Seminal to PLASMA developments was also the idea of superscalar scheduling, which also gained initial traction as a solution for the Cell processor (Bellens et al. 2006). The CellSs system from the Barcelona Supercomputer Center served as the initial inspiration for the development of the QUARK scheduler and its subsequent adoption in PLASMA alongside Pthreads-based routines (Kurzak et al. 2013).

Before PLASMA managed to get significant traction with the user community, GPUs entered the mainstream of HPC, and the MAGMA library (Agullo et al. 2009) became the focal point of dense linear algebra developments at UTK. Due to the differences between GPUs and multicore processors, the design of MAGMA differs significantly from that of PLASMA. Nevertheless, throughout its existence, PLASMA has served as a tremendous research vehicle for the development of new algorithms and scheduling techniques.

Eventually, adoption of superscalar scheduling in the OpenMP standard motivated the retirement of QUARK in favor of OpenMP, as well as retirement of the Pthreads routines. This transition was decided after our successful first experiments with selected functions using the OpenMP tasks summarized in YarKhan et al. (2016).

This article describes the final design of the OpenMP version of PLASMA, and assesses performance on a variety of current multicore hardware configurations for a large set of routines. The most recent version, PLASMA 17,[3] offers an extensive collection of optimized routines for solving linear systems of equations and least-squares problems.

PLASMA is designed to deliver high performance from a system with multiple sockets of multicore processors, an objective achieved by combining state of the art solutions in parallel algorithms, scheduling, and software engineering. In particular, PLASMA is built around the following three concepts.

Tile Matrix Layout. PLASMA utilizes a tile-based storage approach. The matrix is subdivided into square blocks, called *tiles*, of relatively small size, with each tile occupying a continuous memory region. Tiles are loaded to the cache memory efficiently with little risk of eviction while being

[1] http://www.netlib.org/lapack.
[2] http://www.netlib.org/scalapack.
[3] https://bitbucket.org/icl/plasma.

Table 1. An Example of the Naming Conventions for Matrix Matrix Multiply (□gemm)

	Real	**Complex**
64 bit **(double)**	**d**gemm	**z**gemm
32 bit **(single)**	**s**gemm	**c**gemm

processed. The use of the tile layout minimizes conflict cache misses, translation lookaside buffer (TLB) misses, and false sharing, and maximizes potential for prefetching. PLASMA contains parallel and cache efficient routines for converting between the conventional column-major and the tile layouts. PLASMA currently stores both versions of the matrices, so it has larger memory requirements than LAPACK.

Tile Algorithms. PLASMA is based on algorithms redesigned to work on tiles, which maximize data reuse in the cache levels of multicore systems. Tiles are loaded to the cache and processed completely before being transferred back to the main memory. Operations on small tiles create fine grained parallelism providing enough work to keep a large number of cores occupied. Initial work on tile algorithms was published in Agullo et al. (2009), Buttari et al. (2009), and a recent overview for the development of tile algorithms can be found in Abdelfattah et al. (2016).

Dynamic Scheduling. PLASMA relies on concurrent runtime scheduling of sequential tasks. Runtime scheduling is based on the idea of assigning work to cores based on the availability of data for processing at any given point in time, and thus is also sometimes called data-driven scheduling. The concept is related closely to the idea of expressing computation through a task graph, often referred to as the Directed Acyclic Graph (DAG), and the flexibility of exploring the DAG at runtime. This is in direct opposition to the fork-and-join scheduling, where artificial synchronization points expose serial sections of the code and multiple cores are idle while sequential processing takes place. Currently, PLASMA relies on OpenMP for dynamic, task-based, scheduling. Comparison of the two runtimes for PLASMA was presented in YarKhan et al. (2016) showing that the more general-purpose tasks of OpenMP are able to provide the same performance as the more specialized QUARK. PLASMA makes use of tasks with dependencies and priorities, therefore a compiler supporting these features of the OpenMP 4.5 standard is required.

The asynchronous execution of the sequential tasks generally makes very efficient use of the hardware, leading to compact traces throughout the majority of the runtime. Traces typically become sparse only at the very beginning or end of the algorithm, where algorithms do not expose enough parallelism and communication costs may dominate.

2 ALGORITHMS IN PLASMA

2.1 Structure of PLASMA

PLASMA closely follows the structuring of functionality found in the LAPACK and BLAS libraries (Dongarra et al. 1990a, 1990b, 1988a, 1988b; Lawson et al. 1979). Let us take the example of matrix-matrix multiply in double real precision; the well-known `dgemm` routine. Four different versions are provided for most subroutines, related to different data precisions and distinguished by the leading letter; see Table 1. We use □ as a generic symbol for any of these precisions.

The PLASMA routine stack is depicted in Figure 1. Two different levels of functions are exposed to the user. The top level function, `plasma_dgemm`, is a parallel analog of the `dgemm` from BLAS; see the listing in Figure 2. Note that this is rather different from the implementation in the well known PBLAS library[4] for distributed memory architectures. During the execution of this function,

[4]http://www.netlib.org/scalapack/pblas_qref.html.

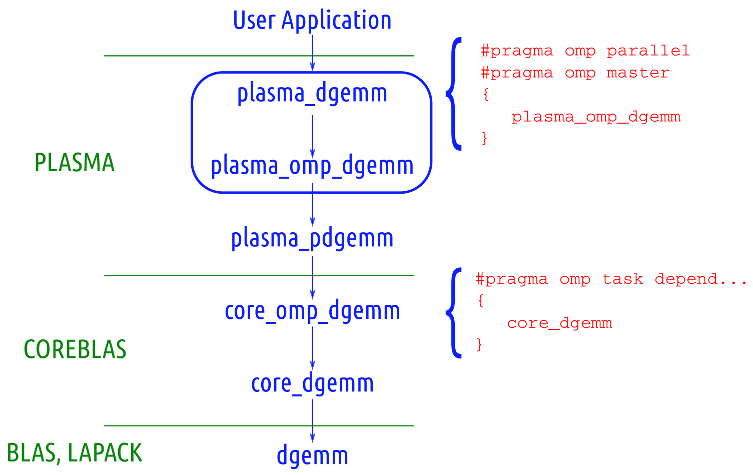

Fig. 1. Overview of PLASMA structure with key parts of the OpenMP implementation. User-level routines are in the box.

```
int plasma_dgemm(plasma_enum_t transA, plasma_enum_t transB,
                 double alpha, double *pA, int lda,
                               double *pB, int ldb,
                 double beta,  double *pC, int ldc)
{
    ...

    // asynchronous block
    #pragma omp parallel
    #pragma omp master
    {
        // Translate matrices from LAPACK to tile layout
        plasma_omp_zge2desc(pA, lda, A);
        plasma_omp_zge2desc(pB, ldb, B);
        plasma_omp_zge2desc(pC, ldc, C);

        // Call the asynchronous function
        plasma_omp_dgemm(transA, transB,
                         alpha, A,
                                B,
                         beta,  C);

        // Translate result back to LAPACK layout
        plasma_omp_zdesc2ge(C, pC, ldc);
    }
    // implicit synchronization

    ...
}
```

Fig. 2. An example of the plasma_dgemm top-level function. Only the master thread executes the code in the parallel block and then creates and enqueues sequential tasks. The plasma_omp_zge2desc and plasma_omp_zdesc2ge serve for translating a matrix from LAPACK to tile layout of the PLASMA matrix descriptor and vice versa.

a parallel section of OpenMP is opened by #pragma omp parallel. This is where a number of OpenMP threads are spawned, as specified by the OMP_NUM_THREADS environment variable. All of the code within this block is executed by the master thread only; note the #pragma omp master directive in Figure 2. Tasks are generated by the master thread, inside the function calls within

```
void plasma_omp_dgemm(plasma_enum_t transA, plasma_enum_t transB,
                      double alpha, plasma_desc_t A,
                                    plasma_desc_t B,
                      double beta,  plasma_desc_t C)
{
    // Call the parallel function
    plasma_pdgemm(transA, transB,
                  alpha, A,
                         B,
                  beta,  C);
}
```

Fig. 3. An example of the second-level function `plasma_omp_dgemm`. The code is executed only by the master thread.

the parallel region, and are executed by all the available threads in an asynchronous manner. The master thread proceeds to the end of the parallel region, where it joins the working threads in executing the tasks it has produced. The end of the parallel block acts as a synchronization point, and the execution proceeds beyond this point only after all the tasks have been completed and OpenMP threads closed.

The second level functions in this example are `plasma_omp_zge2desc`, `plasma_omp_dgemm`, and `plasma_omp_zdesc2ge`. The `plasma_omp_zge2desc` and `plasma_omp_zdesc2ge` functions serve for translation of the data layout between LAPACK column-major and tile storage; see Section 2.8. The main function here is `plasma_omp_dgemm`, which is also exposed to the user, and its simplified body is shown in Figure 3. For our chosen `dgemm` example, this function contains just one call to an internal routine with a tile algorithm, i.e., `plasma_pdgemm`; however, multiple algorithms may be combined on this level. Combining multiple algorithms in this way allows their execution to overlap in an asynchronous manner. This overlap of algorithms can significantly reduce the overall execution time for these combined functions, and it is one of the main strengths of PLASMA. This is also the primary reason for exposing the second level of PLASMA functions, which an advanced user can fuse in a custom order inside a user-defined OpenMP parallel region. This interface also allows an advanced user to have fine control over the number of OpenMP threads and their placement. For example, one can run a PLASMA algorithm on a prescribed number of threads specified at the `#pragma omp parallel` clause by the `num_threads()` keyword.

The heart of PLASMA, a tile-based algorithm, is implemented inside the `plasma_pdgemm` function; see Figure 4. Inside this function, which is still executed only by the master thread, loops over matrix tiles appear and functions that process tiles are called. These functions, also called *computational kernels*, are part of the COREBLAS library, which forms a self-standing part of PLASMA.

In the `dgemm` example, the only computational kernel involved is the `core_omp_dgemm` function, see Figure 5. An OpenMP task with data dependencies is generated inside this function by the master thread and enqueued into the OpenMP runtime. From the body of the task, a call to the `core_dgemm` function (Figure 6) is made. In this example the task consists of calling a sequential version of the `dgemm` routine from the CBLAS library (i.e., a C wrapper of BLAS), involving three tiles. In general, the sequential kernels in PLASMA map to simple calls to BLAS routines, calls to LAPACK routines, or custom implementations derived specifically for tile algorithms (e.g. in the case of the QR factorization). The reason for separating the `core_omp_dgemm` function, which creates the task, from the `core_dgemm`, which implements the kernel is allowing the same kernel to be used from different runtime systems, and even from outside of PLASMA (e.g., by the DPLASMA library (Bosilca et al. 2010a, 2010b, 2011, 2012)[5]).

[5]http://icl.cs.utk.edu/dplasma.

```
#define A(m, n) (double*)plasma_tile_addr(A, m, n)
#define B(m, n) (double*)plasma_tile_addr(B, m, n)
#define C(m, n) (double*)plasma_tile_addr(C, m, n)

void plasma_pdgemm(plasma_enum_t transA, plasma_enum_t transB,
                   double alpha, plasma_desc_t A, plasma_desc_t B,
                   double beta,  plasma_desc_t C)
{
    for (int m = 0; m < C.mt; m++) {
        int mvcm = plasma_tile_mview(C, m);
        int ldcm = plasma_tile_mmain(C, m);
        for (int n = 0; n < C.nt; n++) {
            int nvcn = plasma_tile_nview(C, n);
            if (transA == PlasmaNoTrans && transB == PlasmaNoTrans) {
                for (int k = 0; k < A.nt; k++) {
                    int nvak = plasma_tile_nview(A, k);
                    int ldbk = plasma_tile_mmain(B, k);
                    double zbeta = k == 0 ? beta : 1.0;

                    // Call the kernel
                    core_omp_dgemm(transA, transB,
                                   mvcm, nvcn, nvak,
                                   alpha, A(m, k), ldam,
                                          B(k, n), ldbk,
                                   zbeta, C(m, n), ldcm);
                 }
            }
            else {
                // These options were omitted from the listing.
            }
        }
    }
}
```

Fig. 4. Skeleton of the tile matrix matrix multiply plasma_pdgemm. The mt and nt are numbers of rows and columns of tiles of a matrix stored in the tile layout. The macros A(m, n), B(m, n), and C(m, n) at the top expand to the plasma_tile_addr function, which returns the address of the first entry of the tile on the mth tile-row and in the nth tile-column of the corresponding matrix. The plasma_tile_mview and plasma_tile_nview functions return the number of rows and columns in a local tile. The plasma_tile_mmain function returns the leading dimension of the tile, which can be different from m if the matrix descriptor is a submatrix (called "view") of another matrix descriptor without a deep data copy.

2.2 Parallel BLAS

PLASMA contains a full set of routines from the Level 3 BLAS; see Table 2. BLAS routines in PLASMA are parallelized by tiling. Their implementations are mostly straightforward loop nests, and individual tasks are essentially calls to sequential BLAS. The listing in Figure 4 has already shown the simplified tile matrix matrix multiplication (plasma_pdgemm routine).

Parallel BLAS routines in PLASMA are algorithmically equivalent to their reference Netlib implementations.[6]

2.3 Parallel Norms

PLASMA contains a set of routines for computing matrix norms, specifically the *max*, *one*, *infinity*, and *Frobenius* norms. PLASMA employs tiling for increased parallelism within the norm computations. While being mostly memory bound, PLASMA norm routines still benefit from

[6] http://www.netlib.org/blas.

```
void core_omp_dgemm(plasma_enum_t transA, plasma_enum_t transB,
                    int m, int n, int k,
                    double alpha, const double *A, int lda,
                                  const double *B, int ldb,
                    double beta,         double *C, int ldc)
{
    int ak = (transA == PlasmaNoTrans) ? k : m;
    int bk = (transB == PlasmaNoTrans) ? n : k;

    #pragma omp task depend(in:A[0:lda*ak]) \
                     depend(in:B[0:ldb*bk]) \
                     depend(inout:C[0:ldc*n])
    {
        core_dgemm(transA, transB,
                   m, n, k,
                   alpha, A, lda,
                          B, ldb,
                   beta,  C, ldc);
    }
}
```

Fig. 5. An example of the definition of the sequential core_omp_dgemm task. The OpenMP task consists of a call to sequential core_dgemm routine. Some parameters have been omitted for brevity.

```
void core_dgemm(plasma_enum_t transA, plasma_enum_t transB,
                int m, int n, int k,
                double alpha, const double *A, int lda,
                              const double *B, int ldb,
                double beta,         double *C, int ldc)
{
    cblas_dgemm(CblasColMajor,
                transA, transB,
                m, n, k,
                alpha, A, lda,
                       B, ldb,
                beta,  C, ldc);
}
```

Fig. 6. An example of the definition of the sequential core_dgemm kernel. In this example, the function just calls a sequential BLAS dgemm routine. Some parameters have been omitted for brevity.

Table 2. Level 3 BLAS Routines

Name	Description
□gemm	matrix matrix multiply
□hemm	Hermitian matrix matrix multiply
□her2k	Hermitian rank-2k update to a matrix
□herk	Hermitian rank-k update to a matrix
□symm	symmetric matrix matrix multiply
□syr2k	symmetric rank-2k update to a matrix
□syrk	symmetric rank-k update to a matrix
□trmm	triangular matrix matrix multiply
□trsm	triangular solve with multiple right-hand sides

multithreading, as usually a single core cannot saturate the memory bandwidth. Table 3 lists all the norm routines implemented in PLASMA, and Table 4 lists all the types of norms supported.

An example of the tile version of the function computing the *one* norm of a general matrix is provided in Figure 7. In this routine, the vector of column sums is first computed for each tile. Then the partial results are combined in a final reduction step. In the *infinity* norm routine, the same

Table 3. Matrix Norm Routines

Name	Description
□lange	norm of a (general) matrix
□lanhe	norm of a Hermitian matrix
□lansy	norm of a symmetric matrix
□lantr	norm of a triangular or trapezoidal matrix

Table 4. Matrix Norm Types

Name	Description	Definition
PlasmaMaxNorm	max norm—maximum absolute value	$\|\|A\|\|_{\max} = \max_{1\le i\le m, 1\le j\le n} \|a_{ij}\|$
PlasmaOneNorm	one norm—maximum column sum	$\|\|A\|\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} \|a_{ij}\|$
PlasmaInfNorm	infinity norm—maximum row sum	$\|\|A\|\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} \|a_{ij}\|$
PlasmaFrobeniusNorm	Frobenius norm—square root of sum of squares	$\|\|A\|\|_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} \|a_{ij}\|^2\right)^{1/2}$

```
void plasma_pdlange(plasma_enum_t norm,
                    plasma_desc_t A, double *work, double *value)
{
    switch (norm) {
    double *workspace;

    case PlasmaOneNorm:
        for (int m = 0; m < A.mt; m++) {
            int mvam = plasma_tile_mview(A, m);
            int ldam = plasma_tile_mmain(A, m);

            for (int n = 0; n < A.nt; n++) {
                int nvan = plasma_tile_nview(A, n);

                core_omp_dlange_aux(PlasmaOneNorm,
                                    mvam, nvan,
                                    A(m, n), ldam,
                                    &work[A.n*m+n*A.nb]);
            }
        }
        #pragma omp taskwait
        workspace = work + A.mt*A.n;
        core_omp_dlange(PlasmaInfNorm,
                        A.n, A.mt,
                        work, A.n,
                        workspace, value);
        break;
    case ...
        // Other options were omitted from the listing.
    }
}
```

Fig. 7. Example of the routine for norm of a general matrix plasma_pdlange. Only the *one*-norm branch is kept in the listing. The #pragma omp taskwait is a necessary synchronization and the master thread waits for completion of all the enqueued tasks before proceeding to the final accumulation. The work and workspace arrays contain memory preallocated by the user that is used by the subroutines for intermediate storage. The listing has been simplified for brevity.

Table 5. Linear Systems Routines

Name	Description
□gesv	linear system solve
□getrf	triangular factorization
□getrs	linear system solve (previously factored)
[z|c] hesv	Hermitian linear system solve
[z|c] hetrf	Hermitian triangular factorization
[z|c] hetrs	Hermitian linear system solve (previously factored)
□posv	positive definite linear system solve
□potrf	positive definite triangular factorization
□potrs	positive definite linear system solve (previously factored)
[d|s] sysv	symmetric linear system solve
[d|s] sytrf	symmetric triangular factorization
[d|s] sytrs	symmetric linear system solve (previously factored)

approach is applied row-wise. In the *Frobenius* norm routine, the sum of squares is computed for each tile, then the partial results are combined, and then the square root is taken. The *Frobenius* norm follows the LAPACK approach of scaling the results along the way, to avoid unnecessary underflow and overflow (see the LAPACK `□lassq` routine for details). In general, computing partial sums should be beneficial, rather than detrimental, to the numerical stability of the norm computations. Tiling has no effect on the *max* norm, as the operation is order invariant.

In Figure 7, the `core_omp_dlange` kernel is just a simple call to the sequential `dlange` function from LAPACK, which computes the matrix norm of the tile. The `core_omp_dlange_aux` kernel is a custom kernel computing the row or column sums of the tile into a vector without finding their maxima.

In addition, PLASMA contains the `[dz|sc|d|s]amax` routine, which computes the *max* norm for each column of a matrix, and returns the result as a vector. This routine is needed for checking the convergence of the solution in the iterative refinement process of the mixed precision solvers; see Section 2.5.

2.4 Linear Systems

PLASMA contains a set of routines for solving linear systems of equations, both full and band. Routines for solving general systems of equations rely on the LU factorization with partial (row) pivoting, routines for solving symmetric positive definite (SPD) systems rely on the Cholesky factorization, and routines for solving symmetric (not necessarily positive definite) systems rely on the LDL^T factorization by Aasen's algorithm (Aasen 1971).

Dense. Table 5 lists all the linear systems routines implemented in PLASMA. Like LAPACK, PLASMA provides a routine for solving a system of linear equations, as well as a routine for only factoring the matrix, and a routine for solving a system using a previously factored matrix. This allows for a matrix to be factorized once and reusing the result for repeatedly solving different right-hand sides.

Band. Table 6 lists all the band linear solvers that PLASMA implements. PLASMA's nonsymmetric band linear solver is based on a band version of the LU factorization, while for solving an SPD band system of linear equations, it uses a band version of the Cholesky factorization.

To maintain the numerical stability, our band LU routine performs partial (row) pivoting. When the pivoting is applied to the previous columns of L, it could completely destroy its band

Table 6. Band Linear Systems Routines

Name	Description
□gbsv	band linear system solve
□gbtrf	band triangular factorization
□gbtrs	band linear system solve (previously factored)
□pbsv	band positive definite linear system solve
□pbtrf	band positive definite triangular factorization
□pbtrs	band positive definite linear system solve (previously factored)

structure. To avoid these fills, LAPACK only applies the pivoting to the remaining columns. However, PLASMA's band LU routine relies on the PLASMA's LU panel factorization routine that explicitly applies the pivots to the previous columns within the panel. Hence, though PLASMA returns the LU factors in the LAPACK's band matrix format, to store these potential fills, its leading dimension must accommodate the additional $n_d - 1$ entries on the bottom, where n_d is the tile size.

2.4.1 Cholesky Factorization. The Cholesky factorization is a straightforward algorithm to be written in the tile-oriented fashion (Buttari et al. 2009; Haidar et al. 2011), and Figure 8 shows the algorithm used in PLASMA. Apart from using the `core_omp_dgemm` kernel from Figure 5, it uses the `core_omp_dpotrf` kernel for Cholesky factorization of a tile by calling the LAPACK `dpotrf` function, the `core_omp_dtrsm` function for solving a system with a triangular matrix, and the `core_omp_dsyrk` for a rank-k update of a symmetric matrix.

The Cholesky factorization is the basis for solving linear systems of equations, where coefficients form a symmetric positive definite (SPD) matrix. It is part of the `plasma_omp_□posv` routine (Algorithm 1), in which the individual stages are overlapped. A call to the `plasma_□potrf` routine should also precede a call to the `plasma_□potrs` routine, which can be called repeatedly for new right-hand sides, and uses the Cholesky factors as input. While this version based on the top-level PLASMA interfaces would not be overlapped, doing the same with the second-level interfaces of `plasma_omp_□potrf` and `plasma_omp_□potrs` allows a user to benefit from the overlapping. Cholesky factorization is also the basis for computing the inverse of an SPD matrix; see Section 2.6.

ALGORITHM 1: Cholesky-based solution of $AX = B$ (`plasma_omp_dposv`)

```
Data: $A$, $B$
Result: $X$
$A = LL^T$    Cholesky factorization of matrix $A$, plasma_pdpotrf;
$LY = B$    forward solve, plasma_pdtrsm;
$L^T X = Y$    backward solve, plasma_pdtrsm;
```

2.4.2 LU Factorization. The critical component of the LU factorization is the step of factoring a panel, which in PLASMA is a column of tiles. This operation is on the critical path of the algorithm and has to be optimized to the fullest. At the same time, a naive implementation, such as the `□getf2` routine in LAPACK, is memory bound.

The current implementation of the LU panel factorization in PLASMA is a result of convergence of multiple different research efforts, specifically the work on *Parallel Cache Assignment* (PCA) by Castaldo and Whaley (2010), and the work on parallel recursive panel factorization by Dongarra et al. (2014). Also, the survey by Donfack et al. (2015) provides a good overview of different implementations of the LU factorization.

```
void plasma_pdpotrf(plasma_enum_t uplo, plasma_desc_t A)
{
    if (uplo == PlasmaUpper) {
        for (int k = 0; k < A.nt; k++) {
            int nvak = plasma_tile_nview(A, k);
            int ldak = plasma_tile_mmain(A, k);

            core_omp_dpotrf(PlasmaUpper, nvak,
                            A(k, k), ldak);

            for (int m = k+1; m < A.nt; m++) {
                int nvam = plasma_tile_nview(A, m);
                core_omp_dtrsm(PlasmaLeft, PlasmaUpper,
                               PlasmaConjTrans, PlasmaNonUnit,
                               A.nb, nvam,
                               1.0, A(k, k), ldak,
                                    A(k, m), ldak);
            }
            for (int m = k+1; m < A.nt; m++) {
                core_omp_dsyrk(
                    PlasmaUpper, PlasmaConjTrans,
                    nvam, A.mb,
                    -1.0, A(k, m), ldak,
                     1.0, A(m, m), ldam);

                for (int n = k+1; n < m; n++) {
                    core_omp_dgemm(
                        PlasmaConjTrans, PlasmaNoTrans,
                        A.mb, nvam, A.mb,
                        -1.0, A(k, n), ldak,
                              A(k, m), ldak,
                         1.0, A(n, m), ldan);
                }
            }
        }
    }
    else {
        // This option was omitted from the listing.
    }
}
```

Fig. 8. Algorithm for the Cholesky factorization plasma_pdpotrf. Only the branch for storing the upper triangle of the matrix is shown. The listing has been simplified for brevity.

The panel factorization is shown in Figure 9. It relies on internal blocking and persistent assignment of tiles to threads. Unlike past implementations, it is not recursive, as plain recursion proved inferior to blocking. Memory residency provides cache reuse for the factorization of sub-panels, while blocking provides some level of compute intensity for the sub-tile update operations. The result is an implementation that is not memory bound and scales well with the number of cores.

Priorities on tasks serve as hint for the OpenMP runtime to schedule the tasks on the critical path, namely the panel factorization, the update of the subsequent block column, and their nested tasks, as soon as their dependencies are satisfied.

The complete LU factorization, including the panel factorization, and the updates to the trailing submatrix, is multithreaded differently than other operations in PLASMA. Due to some operations affecting entire columns of tiles, data-dependent tasks are created for column operations, not tile operations, i.e., dependency tracking is resolved at the granularity of columns, not individual tiles. To allow transition between the tile-oriented algorithms and the column-oriented LU, dummy tasks have been introduced. These tasks do not perform any useful work, and their only purpose is inserting data dependencies of the column on all its tiles and vice versa. This translation of data

```
double *a00, *a20;
a00 = A(k, k);
a20 = A(A.mt-1, k);

int ma00k = (A.mt-k-1)*A.mb;
int na00k = plasma_tile_nmain(A, k);
int lda20 = plasma_tile_mmain(A, A.mt-1);

int nvak = plasma_tile_nview(A, k);
int mvak = plasma_tile_mview(A, k);
int ldak = plasma_tile_mmain(A, k);

int num_panel_threads = imin(plasma->max_panel_threads,
                             minmtnt-k);
#pragma omp task depend(inout:a00[0:ma00k*na00k]) \
                 depend(inout:a20[0:lda20*nvak]) \
                 depend(out:ipiv[k*A.mb:mvak]) \
                 priority(1)
{
    volatile int *max_idx = (int*)malloc(num_panel_threads*sizeof(int));
    volatile double *max_val = (double*)malloc(num_panel_threads*sizeof(double));
    volatile int info = 0;

    plasma_barrier_t barrier;
    plasma_barrier_init(&barrier);

    #pragma omp taskloop untied shared(barrier) \
                         num_tasks(num_panel_threads) \
                         priority(2)
    for (int rank = 0; rank < num_panel_threads; rank++) {
        {
            plasma_desc_t view =
                plasma_desc_view(A,
                                 k*A.mb, k*A.nb,
                                 A.m-k*A.mb, nvak);

            core_dgetrf(view, &ipiv[k*A.mb], ib,
                        rank, num_panel_threads,
                        max_idx, max_val, &info,
                        &barrier);
        }
    }
    #pragma omp taskwait

    free((void*)max_idx);
    free((void*)max_val);

    for (int i = k*A.mb+1; i <= imin(A.m, k*A.mb+nvak); i++)
        ipiv[i-1] += k*A.mb;
}
```

Fig. 9. Implementation of the LU panel factorization based on nested tasks with priorities. The `plasma_desc_view` function creates a descriptor for a submatrix, using the original memory of the parent matrix. The `ipiv` array stores the indices of rows for permutation due to pivoting.

dependency seems needed due to the lack of multi-dependencies in OpenMP, which would allow a loop over addresses in the depend clause. An example of dummy tasks inserted in front of the panel factorization is shown in Figure 10.

Nested tasks are created within each panel factorization, and internally synchronized using thread barriers. Similarly, nested tasks are created within each column of □gemm updates, and synchronized with the `#pragma omp taskwait` clause, before exiting the parent task. Waiting

```
double *a00 = A(k, k);
for (int m = k+1; m < A.mt-1; m++) {
    double *amk = A(m, k);
    #pragma omp task depend (in:amk[0]) \
                     depend (inout:a00[0])
    {   // Some useless work is done here.
        int l = 1;
        l++;
    }
}
```

Fig. 10. An example of dummy tasks introduced for creating dependency of a panel starting at address A(k, k) on all its tiles A(m, k).

for completion of the nested tasks is necessary for correct dependency-tracking at the column granularity.

The LU factorization is the basis for routines for solving systems of linear equations. It is part of the `plasma_omp_□gesv` routine (Algorithm 2), in which the individual stages are overlapped. A call to the `plasma_dgetrf` routine should also precede a call to the `plasma_dgetrs` routine, which can be called repeatedly for new right-hand sides, and uses the LU factors as input.

ALGORITHM 2: LU-based solution of $AX = B$ (`plasma_omp_dgesv`)

```
Data: $A$, $B$
Result: $X$
$PA = LU$      LU factorization of matrix $A$, plasma_pdgetrf;
$\widetilde{B} = PB$     row permutation of $B$, plasma_pdgeswp;
$LY = \widetilde{B}$     forward solve, plasma_pdtrsm;
$UX = Y$      backward solve, plasma_pdtrsm;
```

2.4.3 LDL^T Factorization. To solve a symmetric indefinite linear system, PLASMA first reduces the symmetric matrix into a band form by the tiled Aasen's algorithm (Aasen 1971; Ballard et al. 2014), see also Higham (2002) for its analysis. This is different from the blocked Aasen's algorithm (Rozložník et al. 2011) implemented in LAPACK, and the bound on the backward error depends linearly on the tile size. At each step, the algorithm first updates the panel in a left-looking fashion. To exploit the limited parallelism for updating each tile of the panel, PLASMA applies a parallel reduction and accumulates a set of independent updates into a user-supplied workspace. How much parallelism the algorithm can exploit depends on the number of tiles in the panel and the amount of the workspace provided by the user. Once the update is completed, the panel is factorized using the LU panel factorization routine. Hence, the algorithm follows the task dependencies by columns in the nested fashion, as described in the previous section.

Then, in the second stage of the algorithm, the band matrix is factored using the PLASMA band LU factorization routine. Since there is no explicit global synchronization, a task to factor the band matrix can be started as soon as all the data dependencies are satisfied. This allows the execution of these two algorithms to be merged, improving the parallel performance, especially since both algorithms have limited amount of parallelism that can be exploited. A more detailed description and performance analysis of the algorithm can be found in Yamazaki et al. (2018).

2.5 Mixed Precision

PLASMA implements mixed precision routines for the solution of general linear systems of equations and SPD systems of equations. PLASMA mixed precision routines are algorithmically

Table 7. Mixed Precision Routines

Name	Description
[zc|ds]gesv	linear system solve
[zc|ds]posv	positive definite linear system solve

equivalent to their LAPACK counterparts. Table 7 lists all the mixed precision routines implemented in PLASMA.

The algorithms are based on factorizing the matrix in reduced precision (32 bits) and recovering the full precision accuracy (64 bits) in the process of iterative refinement. The approach is motivated by the performance advantage of single precision arithmetic over double precision arithmetic, which is typically twofold. If the input matrix is well conditioned, and the full precision can be recovered in a few steps of refinements, then double precision solution can be delivered almost at the speed of computing the single precision solution (Baboulin et al. 2009; Buttari et al. 2007; Langou et al. 2006).

Algorithm 3 summarizes the iterative refinement method in mixed precision for SPD matrices implemented in PLASMA. Dotted quantities $\dot{A}$, $\dot{L}$, $\dot{x}$, $\dot{y}$, $\dot{b}$, $\dot{d}$, $\dot{r}$ denote values in single precision. Adding and removing a dot to a vector corresponds to conversion from double to single precision and vice versa. In order for the algorithm to calculate a residual, a copy of the matrix in full precision needs to be preserved. This incurs additional memory requirements.

In case the refinement procedure does not converge (i.e., backward error stopping criterion is not met) after 30 iterations, the routine falls back to solving the system with a standard algorithm in full precision.

ALGORITHM 3: Iterative refinement procedure for solution of linear system $Ax = b$ with an SPD matrix A in mixed precision (plasma_dsposv)

```
Data: $A, b$
Result: $x$
$\dot{A} = \dot{L}\dot{L}^T$    Factorize $\dot{A}$ using Cholesky algorithm, plasma_pspotrf;
$\dot{L}\dot{y} = \dot{b}$    Solve linear system, plasma_pstrsm;
$\dot{L}^T\dot{x} = \dot{y}$    Solve linear system, plasma_pstrsm;
$r = b - Ax$    Compute residual, plasma_pdsymm;
if $||r||_{\max} \geq ||x||_{\max}\, ||A||_\infty\, \varepsilon \sqrt{n}$ then
    $x_1 = x$    Save computed solution;
    repeat
        $\dot{L}\dot{y} = \dot{r}_i$    Solve linear system for vector $\dot{y}$, plasma_pstrsm;
        $\dot{L}^T\dot{d}_i = \dot{y}$    Solve linear system for vector $\dot{d}$, plasma_pstrsm;
        $x_{i+1} = x_i + d_i$    Update computed solution, plasma_pdgeadd;
        $r_{i+1} = b - Ax_{i+1}$    Compute residual, plasma_pdsymm;
    until $||r_{i+1}||_{\max} < ||x_{i+1}||_{\max}\, ||A||_\infty\, \varepsilon \sqrt{n}$;
end
```

2.6 Matrix Inversion

PLASMA contains a set of routines for computing the inverse of a matrix. Routines for inverting general matrices rely on the LU factorization with partial (row) pivoting, whilst routines for inverting SPD matrices rely on the Cholesky factorization; see Algorithms 4 and 5. The inversion

Table 8. Matrix Inversion Routines

Name	Description
□getri	matrix inversion (LU factorization as input)
□potri	positive definite matrix inversion (Cholesky factorization as input)
□geinv	matrix inversion (includes the LU factorization)
□poinv	positive definite matrix inversion (includes the Cholesky factorization)

routines are split into three phases: the factorization of the matrix into triangular factors, the inversion of a triangular factor, and the reconstruction of the inverse from its factor.

In general, matrix inversion should not be used for solving linear systems of equations for stability reasons. Instead, matrix factorizations such as LU, LL^T, or LDL^T should be used, followed by forward and backward substitution. Yet, finding the explicit inverse of a matrix is still required in some applications, such as inverting the covariance matrix in statistics.

Table 8 lists all the matrix inversion routines implemented in PLASMA. The `poinv` function uses the Cholesky factorization (`potrf`) for finding the inverse of a positive definite matrix (Algorithm 4). This function was introduced to PLASMA to allow overlapping between the three phases of the inversion using the asynchronous tasks. By contrast, the `potri` function does not include the Cholesky factorization, and it expects a Cholesky factor as input. Similarly, the new `geinv` function for computing the inverse of a general matrix computes the LU factorization (Algorithm 5), whereas the traditional `getri` function expects LU factors as input.

Merging the individual stages is known to lead to high-performance implementations (Agullo et al. 2010; Bouwmeester and Langou 2010), and it provides very compact traces.

ALGORITHM 4: Cholesky-based computation of A^{-1} (`plasma_omp_dpoinv`)

```
Data: $A$
Result: $A^{-1}$
$A = LL^T$    Cholesky factorization of matrix $A$, plasma_pdpotrf;
$L^{-1}$    inverse of $L$, plasma_pdtrtri;
$A^{-1} = (L^T)^{-1}L^{-1}$    multiplication of the triangular parts, plasma_pdlauum;
```

ALGORITHM 5: LU-based computation of A^{-1} (`plasma_omp_dgeinv`)

```
Data: $A$
Result: $A^{-1}$
$PA = LU$    LU factorization of matrix $A$, plasma_pdgetrf;
$U^{-1}$    inverse of $U$, plasma_pdtrtri;
$\widetilde{A}^{-1}L = U^{-1}$    find $A^{-1}$ as the solution to a linear system of equations, plasma_pdgetri_aux;
$A^{-1} = \widetilde{A}^{-1}P$    column permutation of $\widetilde{A}^{-1}$, plasma_pdgeswp;
```

2.7 Least Squares

PLASMA contains routines for solving overdetermined and underdetermined systems of linear equations. It uses QR and LQ factorizations, based on block Householder transformations. PLASMA implementations are rather different from LAPACK. While LAPACK reduces the input matrices by columns, PLASMA does so by tiles. This approach produces algorithms with higher levels of parallelism and excellent scheduling properties (Buttari et al. 2008, 2009).

Table 9. Least-squares Routines

Name	Description
□gelqf	LQ factorization
□gelqs	minimum norm solve using LQ factorization
□gels	overdetermined or underdetermined linear systems solve
□geqrf	QR factorization
□geqrs	least squares solve using QR factorization
□[un|or]glq	generate the Q matrix from LQ factorization
□[un|or]gqr	generate the Q matrix from QR factorization
□[un|or]mlq	apply the Q matrix from LQ factorization
□[un|or]mqr	apply the Q matrix from QR factorization

Generally, PLASMA QR and LQ algorithms show exceptional *strong scaling*, while being somewhat handicapped in *asymptotic performance*, due to reliance on more complex serial kernels than simple calls to BLAS.

Table 9 lists all the PLASMA routines related to solving overdetermined and under-determined systems of linear equations. This includes routines for QR and LQ factorizations, generation of the Q matrices, as well as application of the orthogonal transformations without explicit generation of the Q matrices.

PLASMA routines have the same numerical stability as LAPACK, but are not algorithmically equivalent to LAPACK. This is because PLASMA reduces the input matrices by tiles, not by full columns, and generates sets of tile reflectors in the process. This makes no difference to the user, as long as PLASMA functions are used for operations involving the reflectors, such as generation of the Q matrix or application of the transformations to another matrix.

PLASMA includes support for QR factorization of tall and skinny matrices, for which the number of rows m is much larger than the number of columns n. In this scenario, algorithmic parallelism is increased by concurrent elimination of blocks within a panel, and proceeds according to a reduction tree until all tiles below the diagonal are eliminated. The approach was described in Demmel et al. (2008), and extended, e.g., in Dongarra et al. (2013). Tree-based Householder reductions were recently used for singular value decomposition (SVD) in Faverge et al. (2016).

Since different trees may be beneficial in different circumstances, PLASMA 17 has introduced several trees and a new flexible implementation of this functionality. A tree is first traversed and the elimination kernels are registered into a 1D array. After this, tasks are created following the order given by this array. This approach permits a quick reuse of a certain tree across all QR and LQ routines as well as the possibility to apply Householder reflectors to form an action of Q or its transpose.

The QR and LQ algorithms are the basis for solving systems with rectangular matrices—the least-squares problems for $m \geq n$ and the underdetermined systems for $m < n$. The structure of the `plasma_omp_dgels` routine is in Algorithm 6. In addition, the QR and LQ factorizations are performed either by the standard algorithms in `plasma_pdgeqrf` (see Figure 11) and `plasma_pdgelqf` or by their versions based on the reduction tree `plasma_pdgeqrf_tree` and `plasma_pdgelqf_tree`. QR algorithm requires custom kernels `core_omp_dtsqrt` and `core_omp_dtsmqt` for QR factorization and Q^T application of a matrix composed from two tiles. Although the tile QR factorization (`core_omp_dgeqrt`) and Q^T application (`core_omp_dormqr`) correspond to their LAPACK counterparts, inner blocking with block size ib is performed inside these kernels and nonblocked implementations from LAPACK are called.

ALGORITHM 6: Solving overdetermined and underdetermined systems of equations $AX = B$ (`plasma_omp_dgels`)

```
Data: $A, B$
Result: $X$
if $m \geq n$ then
    $A = QR$    QR factorization of $A$, plasma_pdgeqrf;
    $Y = Q^T B$    application of $Q^T$ to $B$, plasma_pdormqr;
    $RX = Y$    finding the least-squares solution $X$, plasma_pdtrsm;
else
    $A = LQ$    LQ factorization of $A$, plasma_pdgelqf;
    $LY = B$    solve the linear system for $Y$, plasma_pdtrsm;
    $X = Q^T Y$    find the minimum norm solution to the underdetermined system, plasma_pdormlq;
end
```

2.8 Other Implementation Details

PLASMA is written in C, with interfaces for Fortran provided via automatic code generation during compilation of the library. In particular, PLASMA is shipped with a Python script, which parses the C header files, and generates a Fortran module with the interface. The bindings are based on features provided by the Fortran 2003 standard, most importantly the `iso_c_binding` intrinsic module. PLASMA includes several Fortran examples of using the top-level as well as the second-level functions.

The four different precisions (Table 1) are generated by another Python script. This takes the prototypes in the double complex precision and produces the other precisions by textual substitutions in the source codes.

Additional details on functionality implemented in PLASMA can be found in Abalenkovs et al. (2017a).

3 PERFORMANCE EVALUATION

In this section, we present a comprehensive set of benchmarks for the PLASMA routines previously described. Performance is reported for each PLASMA routine on three distinct platforms within a shared memory environment. In each case, we utilize the maximum available number of cores and examine performance across a range of matrix sizes. We use real double precision variables throughout, with the exception of the mixed precision routines, which combine real double and real single precision variables. A major focus of this study is to asses the performance of state-of-the-art tile-based algorithms, in comparison to block-column-based algorithms, such as those present in the LAPACK library.

3.1 Hardware, Library and Compiler Details

Three recent shared-memory multicore platforms have been selected for this study, namely: a two-socket compute node based on Intel Xeon processors (Haswell generation, 20 cores), an Intel Xeon Phi 7250 processor (Knights Landing generation, 68 cores), and a two-socket machine based on an IBM Power 8 processor (20 cores). Details of the individual platforms are presented in Table 10.

On these three hardware platforms, we compare the performance of PLASMA to other numerical libraries. In particular the Netlib LAPACK library version 3.7.0, linked with a multithreaded optimized BLAS library, provides the baseline for performance comparisons. In the case of Intel architectures, we also compare the performance of PLASMA against that of the Intel Math Kernel Library (MKL). For the IBM Power 8-based system, the IBM Engineering and Scientific Subroutine

```
void plasma_pdgeqrf(plasma_desc_t A, plasma_desc_t T)
{
    for (int k = 0; k < imin(A.mt, A.nt); k++) {
        int mvak = plasma_tile_mview(A, k);
        int nvak = plasma_tile_nview(A, k);
        int ldak = plasma_tile_mmain(A, k);

        core_omp_dgeqrt(mvak, nvak, ib,
                        A(k, k), ldak,
                        T(k, k), T.mb);

        for (int n = k+1; n < A.nt; n++) {
            int nvan = plasma_tile_nview(A, n);

            core_omp_dormqr(PlasmaLeft, PlasmaTrans,
                            mvak, nvan, imin(mvak, nvak), ib,
                            A(k, k), ldak,
                            T(k, k), T.mb,
                            A(k, n), ldak);
        }
        for (int m = k+1; m < A.mt; m++) {
            int mvam = plasma_tile_mview(A, m);
            int ldam = plasma_tile_mmain(A, m);

            core_omp_dtsqrt(mvam, nvak, ib,
                            A(k, k), ldak,
                            A(m, k), ldam,
                            T(m, k), T.mb);

            for (int n = k+1; n < A.nt; n++) {
                int nvan = plasma_tile_nview(A, n);

                core_omp_dtsmqr(PlasmaLeft, PlasmaTrans,
                                A.mb, nvan, mvam, nvan, nvak, ib,
                                A(k, n), ldak,
                                A(m, n), ldam,
                                A(m, k), ldam,
                                T(m, k), T.mb);
            }
        }
    }
}
```

Fig. 11. Skeleton of the standard tile QR factorization. A.mt and A.nt are numbers of rows and columns of tiles in matrix A stored in the tile layout.

Library (ESSL) is used for comparison instead. It should be noted that PLASMA is also linked with these libraries and relies on their sequential implementations of BLAS.

For most tests square matrices of growing size are used; however, non-square matrices are also examined where appropriate to the algorithm, such as in the case of QR factorization. For each test, three runs were performed at each matrix size, using randomly generated matrices. The highest performance obtained from these three runs is reported in the plots.

A crucial parameter within PLASMA is the size of the square tile; i.e., the nb parameter. Lower values will typically increase the parallelism of the algorithm, while higher values allow more efficient utilization of arithmetic units. Consequently PLASMA performance is examined for several tile sizes, with the highest performing tile size reported. The optimal tile size tends to grow slightly as the matrix size is increased. A quick way for users to determine an nb parameter suitable for their architecture and matrix size is to run the PLASMA tester on a range of tile sizes, and then set the one that leads to the best performance through a call to the `plasma_set` function.

Table 10. Platforms Selected for the Benchmarks

Label	Hardware overview	Compilers and libraries
Haswell	• 2 × Intel Xeon CPU E5-2650 v3, 2.30GHz • 2 × 10 = 20 cores • 32GB DRAM • theoretical peak performance 736Gflop/s	• GNU Compiler Collection (GCC) 7.1.0 • Intel C Compiler 16.0.3 • MKL 17.2
Phi	• Intel Xeon Phi 7250 • 68 cores • 16GB MCDRAM • theoretical peak performance 3046Gflop/s • quadrant cluster mode • flat memory mode	• GNU Compiler Collection (GCC) 7.0.1 • Intel C Compiler 16.0.3 • MKL 17.2
POWER8	• 2 × IBM POWER8, 3.5GHz • 2 × 10 = 20 cores • 256GB DRAM • theoretical peak performance 560Gflop/s	• GNU Compiler Collection (GCC) 6.3.1 • IBM XL 20161123 • IBM ESSL 5.5.0

On both the Haswell and Phi platforms, PLASMA and MKL are linked using the GNU C compiler, and utilize the GNU OpenMP (`gomp`) runtime library. Due to issues with using this combination for LAPACK linked with MKL BLAS, we present results for this combination using the Intel C compiler, and the Intel OpenMP (`iomp`) runtime library.

On the Haswell platform tests are performed using the following options:

```
OMP_NUM_THREADS=20 OMP_PROC_BIND=true OMP_MAX_TASK_PRIORITY=100 numactl --inter-
leave=all
```

For the Phi platform tests are run using:

```
OMP_NUM_THREADS=68 OMP_PROC_BIND=true OMP_MAX_TASK_PRIORITY=100 numactl -m=1
```

where the last flag has led to using the fast MCDRAM memory for storing the matrices. The Phi processor was in the flat memory mode, allowing the allocation of large matrices in the MCDRAM memory. This had a significant effect on performance compared to allocating matrices in RAM. The quadrant cluster mode was used, although this did not seem to have a significant impact on performance. The effect of the different memory modes of Phi on performance for linear algebra has been studied in more detail in Haidar et al. (2017).

POWER8 runs use the following:

```
OMP_NUM_THREADS=20 OMP_PROC_BIND=true OMP_PLACES="{0}:20:8"  OMP_MAX_TASK_
PRIORITY=100
```

where the `OMP_PLACES` environment variable maps each OpenMP thread to one physical CPU core, rather than simply taking the first 20 available logical cores.

We present an execution trace for the Cholesky-based matrix inversion. Many more traces can be found in Abalenkovs et al. (2017b).

3.2 Parallel BLAS

PLASMA contains a full parallel implementation of the Level 3 BLAS routines. However, this section focuses only on the performance results of gemm and trsm. This is motivated by the fact that all level 3 BLAS routines, except trsm, can be viewed as a specialized implementation of gemm (Kågström et al. 1998).

We present the performance of `plasma_dgemm` and `plasma_dtrsm` routines on three different architectures, and compare it with the performance of the vendor-provided optimized implementations. Unlike LAPACK routines we do not report the performance of the Netlib reference implementation of BLAS, as it is fully sequential.

Figures 12 and 13 show the performance results on Haswell. For the dgemm routine, MKL performs about 15% better than PLASMA throughout the range of matrix sizes. This result suggests that the current plasma_dgemm routine may have potential room for performance improvement. For the plasma_dtrsm routine, PLASMA provides performance similar to MKL, while offering a more smooth and predictable performance scaling.

The results on the Phi architecture (Figures 14 and 15) demonstrate that MKL is making significantly better use of the 68 available cores. For moderate-sized matrices; i.e., in the 2,000 to 10,000 range, a performance gap of around 500Gflop/s can be observed for dgemm. For dtrsm the performance gap is consistently around 200Gflop/s. These results suggest a significant deficit in efficiency for the PLASMA BLAS routines, in comparison to multithreaded MKL, despite the fact that PLASMA calls sequential MKL BLAS for processing individual tiles.

On the IBM POWER8 platform, PLASMA and the vendor optimized multithreaded library, ESSL, exhibit comparable results for both dgemm and dtrsm. As shown in Figures 16 and 17, both routines reach in excess of 450Gflop/s, representing around 85% of the theoretical peak performance (560Gflop/s). This result demonstrates the capability of PLASMA to efficiently exploit all 20 cores of the POWER8 machine. Again, PLASMA tasks call sequential dgemm from ESSL to process individual tiles.

The optimal tile size parameter (nb) was either 336 or 560 on Haswell, for matrices larger than 4000. The optimal size was 560 on Phi, and 384 on POWER8. Smaller matrices require somewhat smaller tiles for optimal performance.

3.3 Parallel Norms

We present benchmarks for computation of the *one* norm, for general and symmetric matrices; i.e., the dlange and dlansy routines, respectively. These routines are heavily memory bound; hence, their performance is reported in GB/s rather than Gflop/s.

Results on Haswell are summarized in Figures 18 and 19. For both general matrices (Figure 18) and symmetric matrices (Figure 19), MKL significantly out-performs PLASMA and LAPACK. It should be noted that the high performance of MKL was obtained only after calling the C interface function without the "not a number" (NaN) checking; namely, the LAPACKE_dlange_work and LAPACKE_dlansy_work. With LAPACKE_dlange and LAPACKE_dlansy, the performance was significantly worse. In this benchmark, PLASMA is penalized due to the inclusion of translation to the tile layout into the measured time, since this conversion significantly increases the number of memory accesses required. To quantify this effect, we have also performed an experiment excluding the layout conversion from the timing. These results are denoted as "PLASMA*" (with asterisk) in the plots. We can see that if the matrix is already in the tile layout, the norm computations can be performed even faster than by MKL.

The results on the Phi platform are presented in Figures 20 and 21. As in the case of Haswell, MKL significantly outperforms both PLASMA and LAPACK, with the performance differential growing significantly with increasing matrix size. The increased parallelism of the Phi platform does allow PLASMA to significantly out-perform LAPACK, however. We have repeated the experiment excluding the layout conversion time of PLASMA also on Phi. In this scenario, MKL still performs better for general matrices, while PLASMA outperforms MKL for symmetric ones.

Results obtained on the POWER8 platform are shown in Figures 22 and 23. For both general and symmetric matrices, PLASMA out-performs ESSL by around 50%, and offers roughly twice the performance of LAPACK.

The dominant optimal tile size parameter (nb) was found to be 560 on Haswell, 1024 on Phi, and 384 on POWER8.

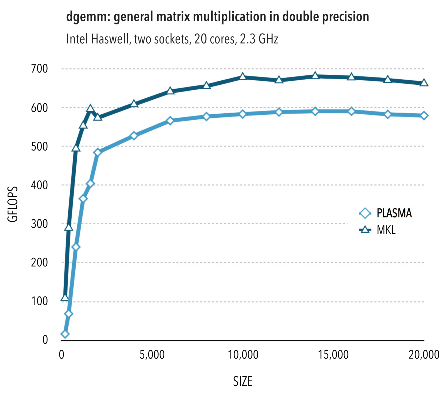

Fig. 12. Performance of dgemm on Haswell.

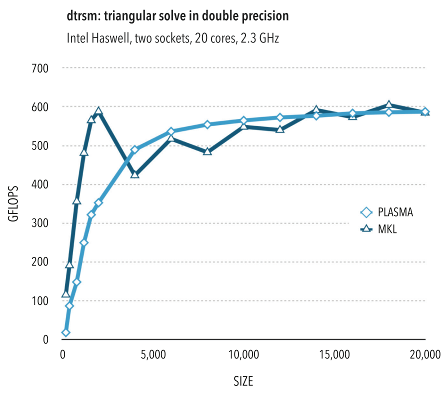

Fig. 13. Performance of dtrsm on Haswell.

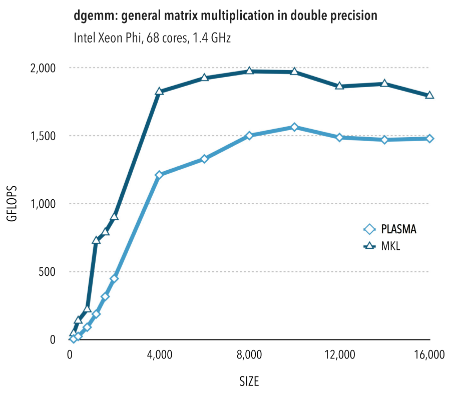

Fig. 14. Performance of dgemm on Phi.

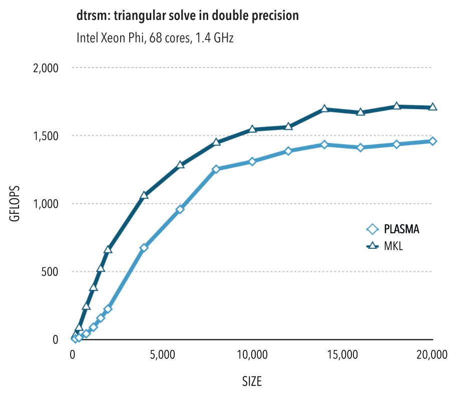

Fig. 15. Performance of dtrsm on Phi.

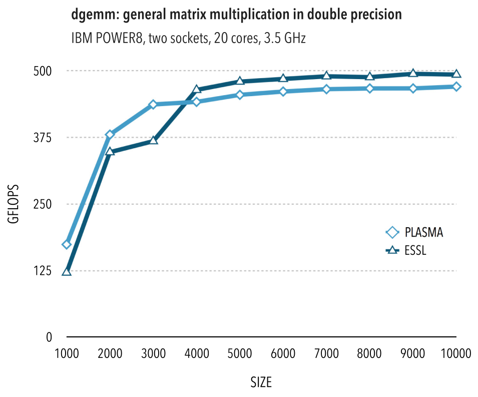

Fig. 16. Performance of dgemm on POWER8.

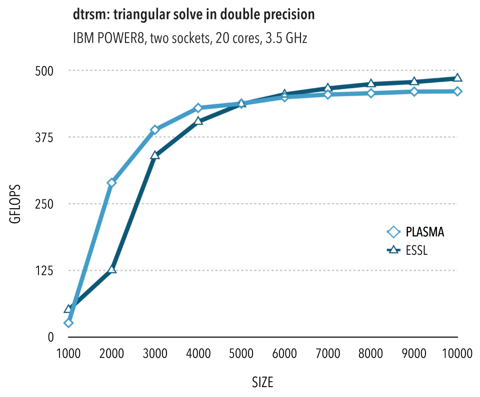

Fig. 17. Performance of dtrsm on POWER8.

3.4 Linear Systems

The PLASMA library provides a range of routines for the factorization of matrices. In this section, we examine performance for the PLASMA implementations of LU factorization (plasma_dgetrf), Cholesky factorization (plasma_dpotrf), and LDL^T factorization (plasma_dsytrf) with dense

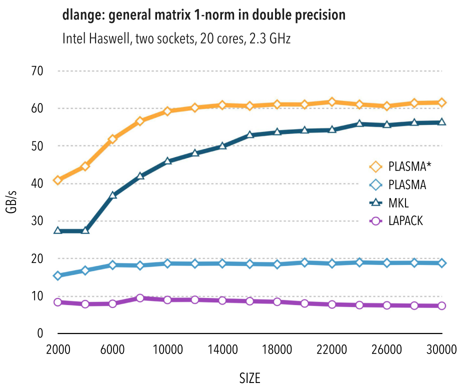

Fig. 18. Performance of dlange on Haswell.

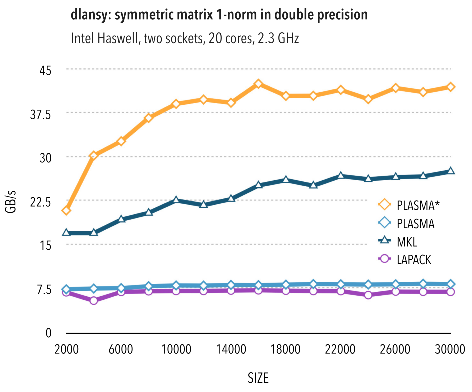

Fig. 19. Performance of dlansy on Haswell.

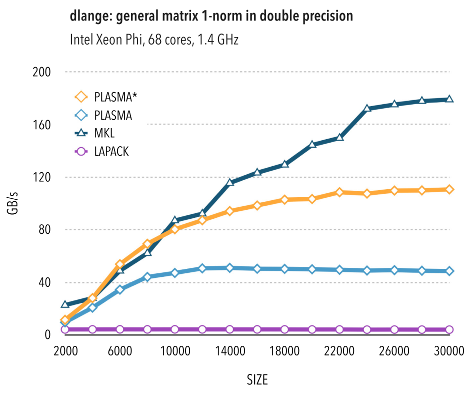

Fig. 20. Performance of dlange on Phi.

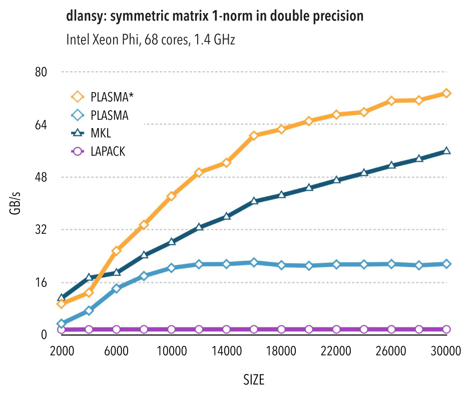

Fig. 21. Performance of dlansy on Phi.

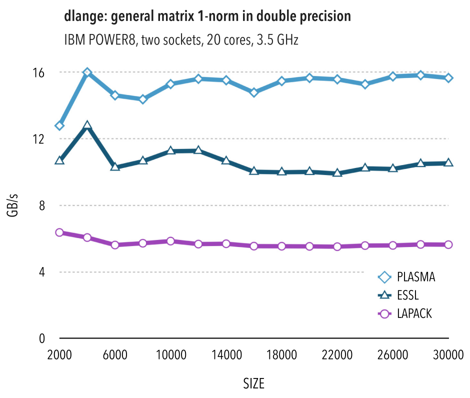

Fig. 22. Performance of dlange on POWER8.

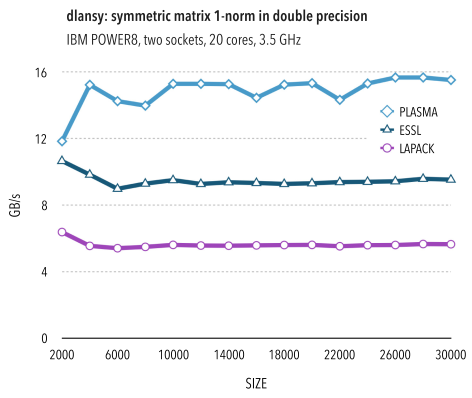

Fig. 23. Performance of dlansy on POWER8.

matrices. We also consider performance for the band-matrix versions of LU and Cholesky factorization; i.e., (`plasma_dgbtrf`) and (`plasma_dpbtrf`). The performance of QR factorization routines for solving least-squares problems are presented in Section 3.7.

We consider first the performance of PLASMA on the Haswell platform. For LU factorization (Figure 24), MKL shows a moderate performance gain over PLASMA throughout the range of matrix sizes, at around 15%. PLASMA does significantly outperform LAPACK, however, showing around a 50% improvement. This improvement is partially due to the parallel panel factorization of PLASMA, in contrast to the standard LU algorithm of LAPACK, which introduces parallelism only through parallel BLAS used for the trailing matrix update.

Figure 25 shows the performance of Cholesky factorization. Here PLASMA and MKL offer very similar performance, with MKL slightly faster for small matrices, and PLASMA slightly ahead for mid-sized matrices. Both MKL and PLASMA again offer significantly improved performance over LAPACK.

Results for the LDL^T factorization are shown in Figure 26. While LU and Cholesky factorization shows a high performance up to 600Gflop/s, which is around 80% of the theoretical peak performance (see Table 10), none of the `dsytrf` implementations achieve even 50% of the theoretical peak. The bottlenecks to providing a scalable implementation of the symmetric indefinite matrix have been discussed in Section 2.4.3, the main issue being the need for symmetric pivoting. Nevertheless, PLASMA is able to outperform MKL and LAPACK by a significant margin, for moderate to large matrices.

Results on the Phi platform are shown in Figures 27–29. Overall trends are very similar to Haswell; in particular, for the LU and Cholesky algorithms. In the case of LDL^T factorization, PLASMA outperforms MKL by a more significant margin than on Haswell, offering more than double the performance on larger matrices.

On the IBM POWER8 architecture, both the ESSL and PLASMA implementations of LU factorization substantially outperform the LAPACK equivalent, as showed in Figure 30. For smaller matrices ESSL demonstrates good performance relative to PLASMA; however, it stagnates early whilst PLASMA performance continues to grow with increasing matrix sizes. The POWER8 experiments for Cholesky factorization: Figure 31 shows ESSL and PLASMA achieving similar performance for smaller matrices, while PLASMA pulls ahead by around 25% for moderately large matrices.

The results for LDL^T factorization in Figure 32 are more complex. For matrices of size ranging from 1,000 to 5,000, the three curves completely overlap, which suggests that either for such small matrices there is not much room for parallelism exploitable by the LDL^T algorithm, or ESSL and PLASMA failed to achieve a better optimization than LAPACK. The latter is more true for ESSL that did not succeed in showing any performance gain over LAPACK for all the matrix sizes considered. However, the performance of PLASMA's LDL^T, increased almost linearly, with the matrix size.

To examine the performance of band routines, we consider the `dgbtrf` and `dpbtrf` functions, which employ LU and Cholesky factorization, respectively, to solve the band systems. In each case, we consider a matrix with a 10 percent band occupancy; that is, the bandwidth is equal to one tenth of the matrix size.

Figures 33 and 34 show the performance of the band routines on the Haswell platform. For Cholesky factorization (Figure 34), we see a very similar performance profile across all three platforms; however, with LU factorization (Figure 33) PLASMA pulls ahead of MKL and LAPACK for large matrix sizes, showing up to a 100% increase in performance over MKL. The improved relative performance of PLASMA in this case appears to be a result of the multi-threaded panel factorization.

Performance on the Xeon Phi is shown in Figures 35 and 36. Here performance with LU is similar between PLASMA and MKL; PLASMA offers better performance on large matrix sizes, with MKL ahead for small matrices. For Cholesky all three routines provide similar performance with smaller matrices, while PLASMA offers increasingly superior performance as the matrix size grows.

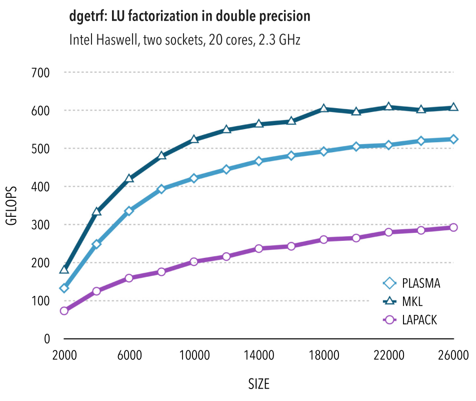

Fig. 24. Performance of dgetrf on Haswell.

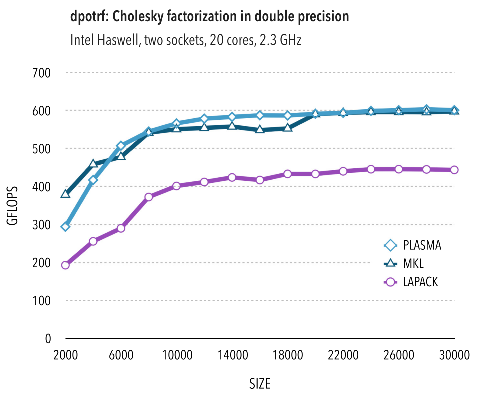

Fig. 25. Performance of dpotrf on Haswell.

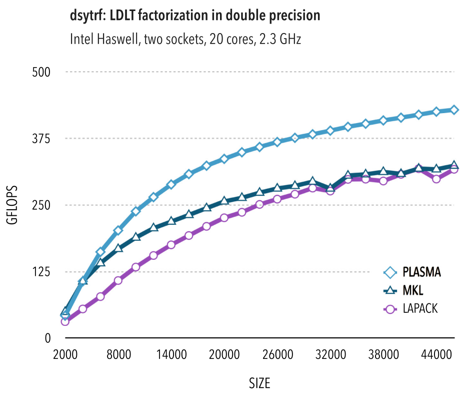

Fig. 26. Performance of dsytrf on Haswell.

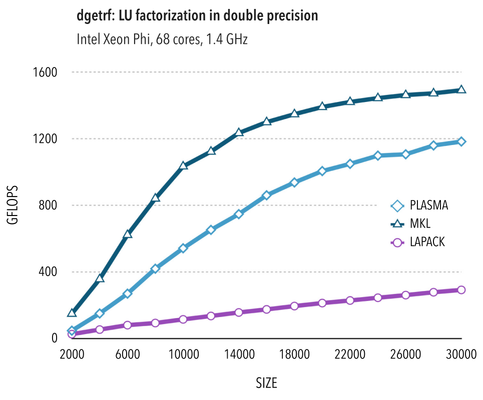

Fig. 27. Performance of dgetrf on Phi.

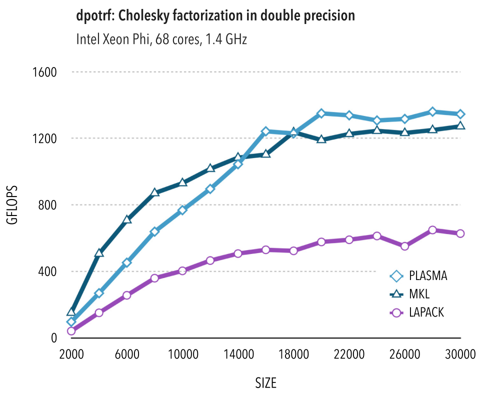

Fig. 28. Performance of dpotrf on Phi.

The results on POWER8, given in Figures 37 and 38, show significant performance improvements for PLASMA over LAPACK and ESSL throughout the range of matrix sizes on both routines. The difference is particularly evident for LU factorization, where PLASMA performance grows to more than double that of the other routines at large matrix sizes.

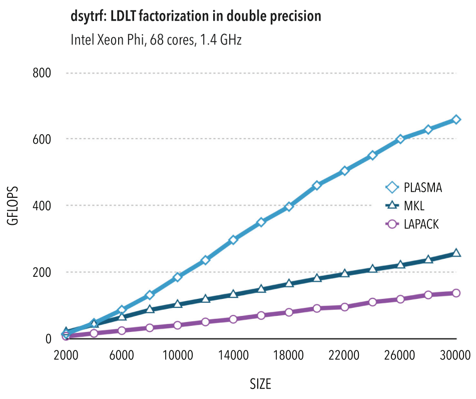

Fig. 29. Performance of dsytrf on Phi.

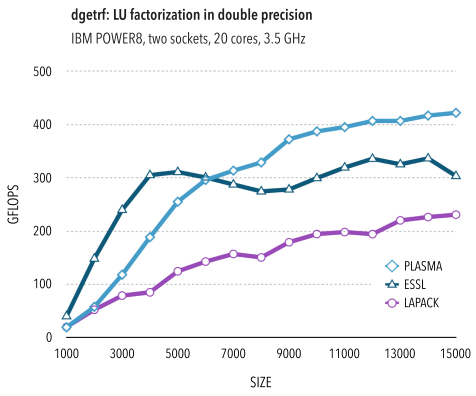

Fig. 30. Performance of dgetrf on POWER8.

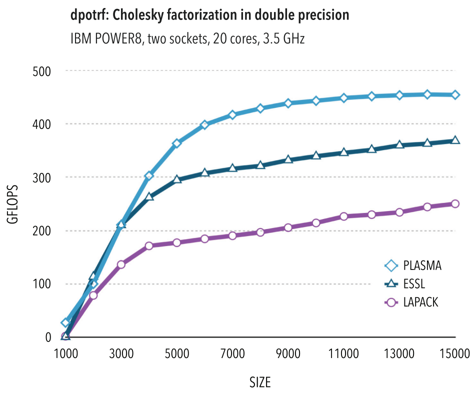

Fig. 31. Performance of dpotrf on POWER8.

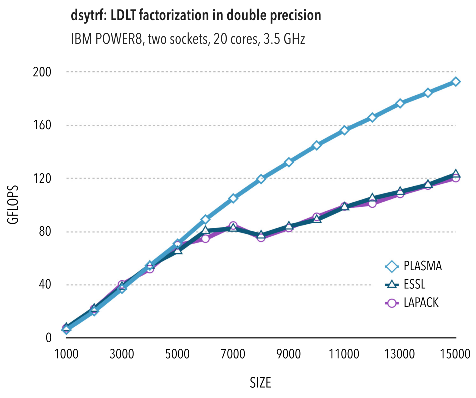

Fig. 32. Performance of dsytrf on POWER8.

It is important to note that, for band routines in general, performance will scale much more strongly with bandwidth than with matrix size. Increasing the matrix size whilst using a fixed bandwidth of modest size will typically provide a flat performance profile, as memory bandwidth becomes saturated before the floating point capacity is exhausted.

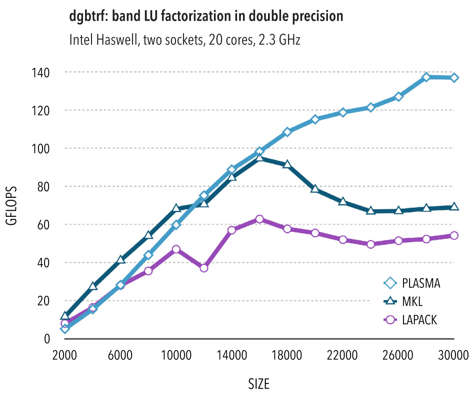

Fig. 33. Performance of dgbtrf on Haswell.

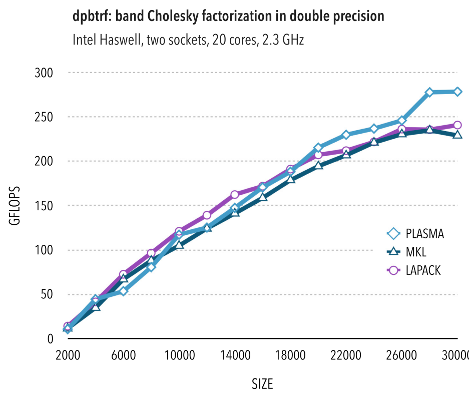

Fig. 34. Performance of dpbtrf on Haswell.

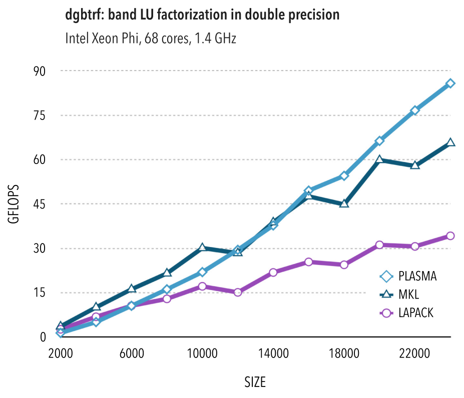

Fig. 35. Performance of dgbtrf on Phi.

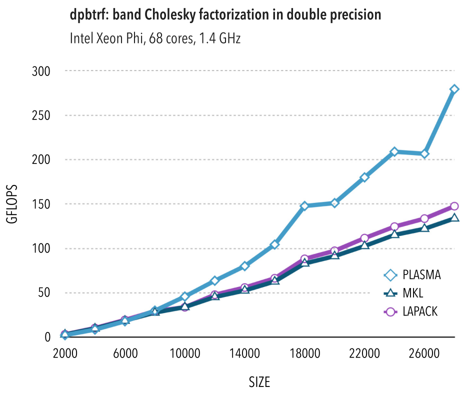

Fig. 36. Performance of dpbtrf on Phi.

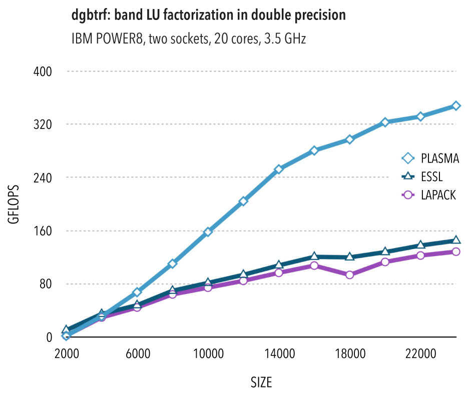

Fig. 37. Performance of dgbtrf on POWER8.

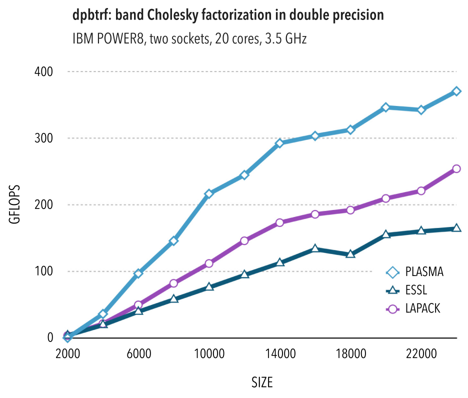

Fig. 38. Performance of dpbtrf on POWER8.

The dominant optimal tile size parameter (nb) for Cholesky factorization of a dense matrix was 336 on Haswell, 448 on Phi, and 384 on POWER8. For LU factorization, it was 228 on Haswell, 448 on Phi and 336 on POWER8. Performance of the LU factorization was found to be sensitive to the maximum number of threads for panel factorization (mtpf), which was set to 8 on Haswell, 20 on

Phi, and 4 on POWER8. The inner blocking parameter (ib) was set to 16 on Haswell, 40 on Phi, and 32 on POWER8. Finally, for the LDL^T factorization, the dominant optimal nb was found to be 192 on Haswell, 352 on Phi, and 128 on POWER8.

For band Cholesky factorization, the dominant nb was 224 on Haswell and Phi, and 128 on POWER8. For band LU factorization, the dominant optimal nb was 168 on Haswell, 224 on Phi, and 128 on POWER8. The maximal number of threads for panel factorization (mtpf) was set to 4 on Haswell and POWER8, and to 8 on Phi.

3.5 Mixed Precision

Performance of the mixed precision iterative refinement based on the LU factorization is presented in Figures 39, 41, and 43. At the time of writing, an issue is present with the coupling of PLASMA to the `gomp` runtime library version 7. This issue is related to the assignment of priorities for nested OpenMP tasks; an approach employed by the LU factorization in PLASMA. To circumvent this issue task priorities are not enabled for the presented results; i.e., setting `OMP_MAX_TASK_PRIORITY=0`. Even with this limitation PLASMA is able to achieve around 25% higher performance than MKL, and more than double the performance of LAPACK on Haswell. On Phi the difference is even more significant; PLASMA achieves around double the performance of MKL, and around quadruple the performance of LAPACK. On POWER8 the performance of PLASMA is comparable with that of ESSL. Here PLASMA offers slightly better performance for smaller matrices, with ESSL slightly ahead for larger matrices. Both libraries provide roughly double the performance of LAPACK.

Performance results of the mixed precision iterative refinement routine `dsposv`, based on Cholesky factorization, are presented in Figures 40, 42, and 44. The number of right-hand side vectors is set to one in all experiments. On the Haswell platform, PLASMA achieves significantly higher performance than MKL for large matrix sizes. On Phi, the PLASMA routine provides a dramatic four- to fivefold improvement compared to its MKL counterpart, for moderate to large matrix sizes.

The dominant optimal tile size for the `plasma_dsgesv` routine on Haswell was 384, with inner blocking ib = 40, and 4 threads used for panel factorization. The same setup was used on POWER8. On Phi, tile size was 352, the inner blocking ib = 64, and 8 threads were assigned to panel factorization. Optimal tile sizes for the `plasma_dsposv` routine were more varied, while being dominated by 480 for Haswell, 576 for Phi, and 384 for POWER8.

Figures 41 and 42 present two additional curves corresponding to conventional linear system solutions in single and double precisions, denoted by PLASMA(S) and PLASMA(D), respectively. To compare performance of all three variants of the linear system solution; `s{ge,po}sv`, `d{ge,po}sv` and `ds{ge,po}sv`, the performance for all routines was calculated using the same formula for the number of floating point operations.

As expected, the mixed precision routine delivers a performance curve that lies between that of the native single and double precision results. The mixed precision performance curve lies much closer to the single precision results in case of the □posv routine, whereas in case of □gesv, performance is more comparable to that of the native double precision routine.

3.6 Matrix Inversion

The routines used to explicitly invert a matrix are described in Section 2.6. The performance results on the various systems are collated in Figures 46–51. On the Haswell architecture (Figures 46 and 47) we see that PLASMA is more performant than MKL and LAPACK for all matrix sizes. PLASMA is particularly effective for Cholesky inversion, where performance improvements over MKL are mostly around 50–100%. On the Phi platform (Figures 48 and 49) the performance of

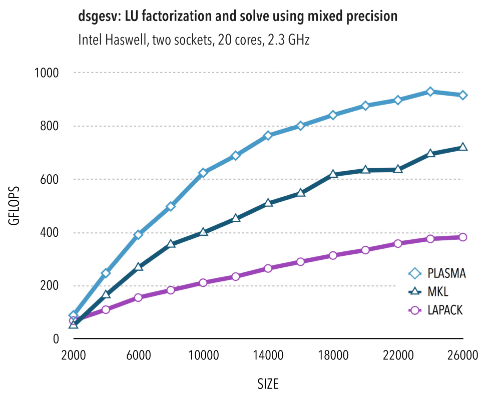

Fig. 39. Performance of dsgesv on Haswell.

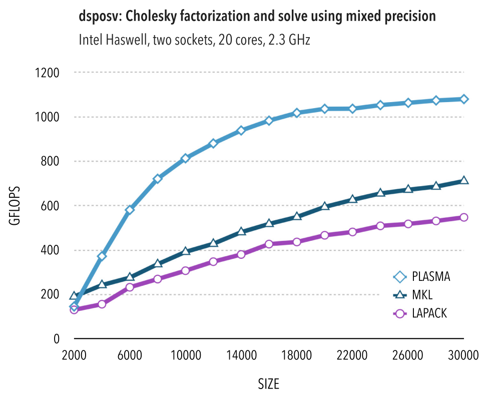

Fig. 40. Performance of dsposv on Haswell.

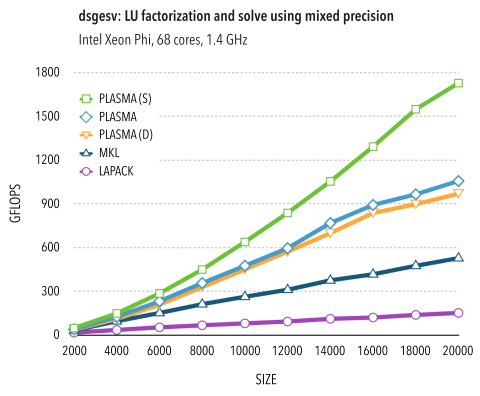

Fig. 41. Performance of dsgesv on Phi.

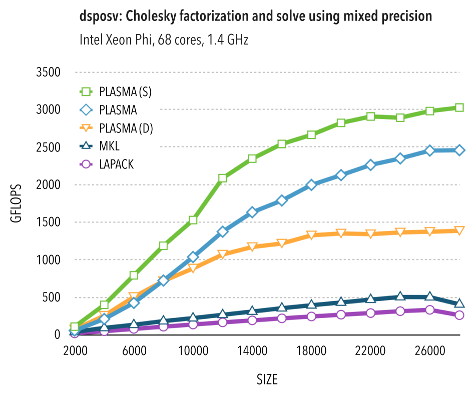

Fig. 42. Performance of dsposv on Phi.

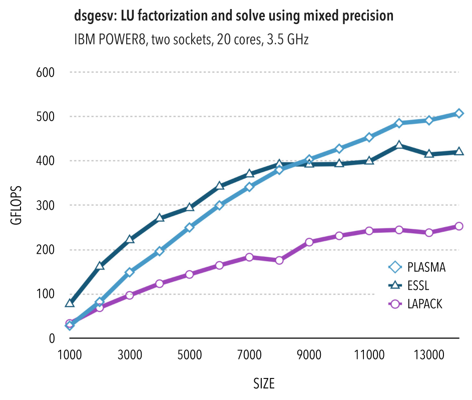

Fig. 43. Performance of dsgesv on POWER8.

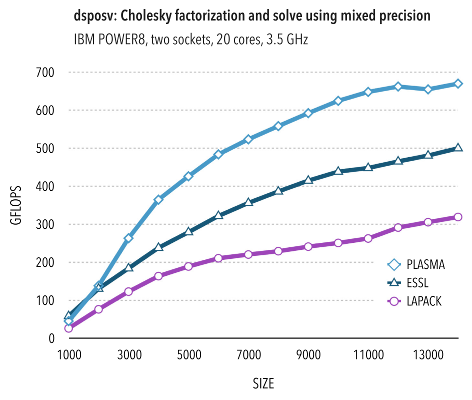

Fig. 44. Performance of dsposv on POWER8.

PLASMA is slightly higher than MKL for `dgeinv`, though PLASMA is once again around twice as fast for `dpoinv`. For both algorithms the performance of LAPACK is well below that of the more heavily optimized libraries. On the POWER8 machine (Figures 50 and 51) things behave rather differently. For `dgeinv` the performance of PLASMA is the best for matrices larger than around

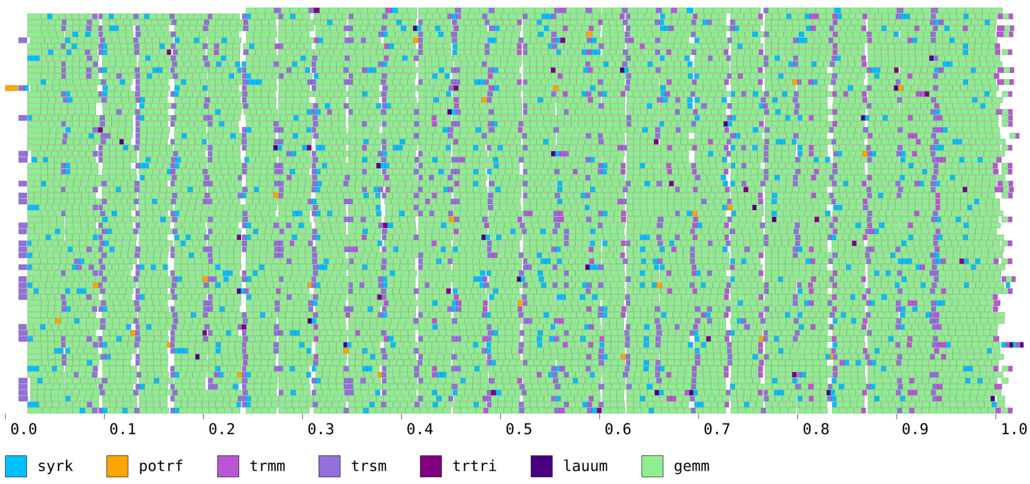

Fig. 45. Trace of plasma_dpoinv on Phi, matrix size 11,648, tile size 448.

5,000. The dpoinv implementation in PLASMA once again provides the fastest implementation, though by a smaller margin than on the other platforms. Here ESSL is roughly halfway between the performance of PLASMA and LAPACK.

In summary, for dgeinv PLASMA obtains slightly superior performance to MKL on Intel architectures and it is faster than ESSL for large matrices on the POWER8 system. However, for dpoinv, PLASMA significantly outperforms the other implementations on all systems.

The performance benefit of the plasma_dpoinv routine stems from the fact that its tile-based Cholesky factorization algorithm is well suited for overlapping the factorization with the subsequent stages (see Algorithm 4). The effect is best shown in an execution trace; see, e.g., Figure 45. It is clear that kernels of the subsequent stages start before the end of the factorization itself, filling the gaps of the factorization algorithm towards the end of the factorization, where this does not generate enough parallelism itself. Unfortunately, partial pivoting prevents such a high degree of overlap in the LU-based routine plasma_dgeinv.

The optimal tile size parameter (nb) did not tend to change much with the matrix size. For the dgeinv routine, the dominant optimal nb was 384 for Haswell, 448 for Phi, and 256 for POWER8. For the dpoinv implementation, the dominant nb was 544 on Haswell, 448 on Phi, and 256 on POWER8.

3.7 Least Squares

Solving overdetermined problems in PLASMA relies on the QR factorization of the matrix (see Algorithm 6). We perform the benchmarks for the QR factorization routine (plasma_dgeqrf), which allows us to avoid dependence on the number of right-hand sides. We run PLASMA using the standard QR algorithm, in which the plasma_pdgeqrf function (Figure 11) is called, and also with the tree-based QR algorithm, in which the plasma_pdgeqrf_tree function is used instead. In the charts to follow, "PLASMA" (no asterisk) refers to the standard algorithm, while "PLASMA*" (with asterisk) refers to the tree-based one.

Results are summarized in Figures 52–57. The first experiment is monitoring the performance of the QR factorization on square matrices of increasing dimension. In this scenario, updating the trailing matrix provides enough parallelism to keep the cores busy. As a result, for PLASMA, the tree-based algorithm is providing slightly lower performance than the standard algorithm on

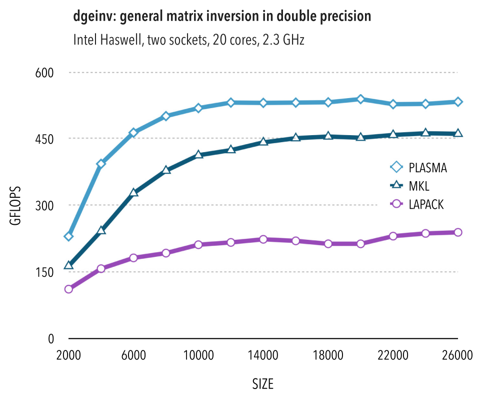

Fig. 46. Performance of dgeinv on Haswell.

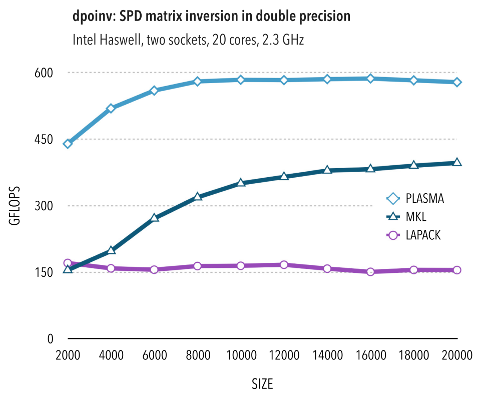

Fig. 47. Performance of dpoinv on Haswell.

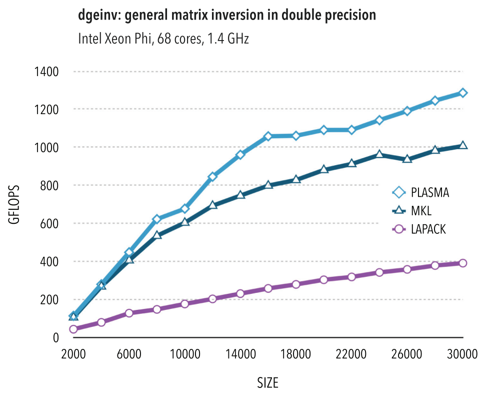

Fig. 48. Performance of dgeinv on Phi.

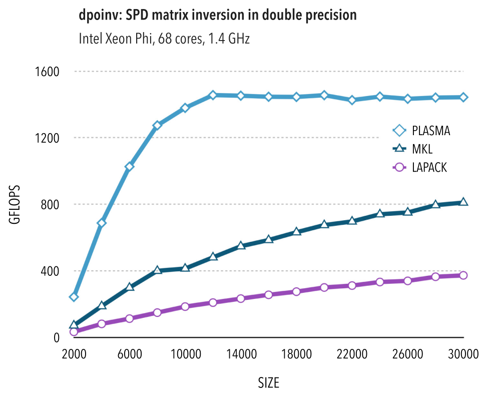

Fig. 49. Performance of dpoinv on Phi.

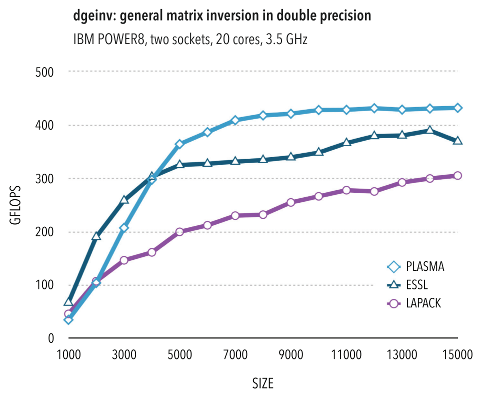

Fig. 50. Performance of dgeinv on POWER8.

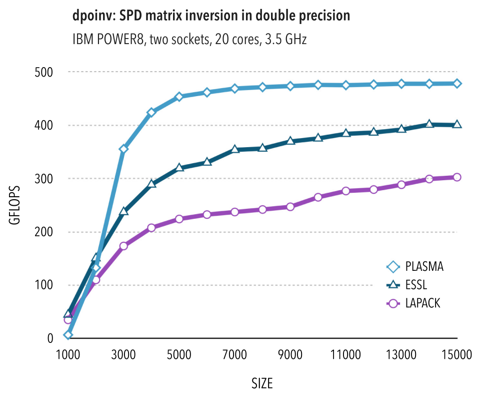

Fig. 51. Performance of dpoinv on POWER8.

all the tested platforms. This is related to the worse data locality due to the need for visiting some tiles twice when eliminating them by the `tt` (triangle-on-top-of-triangle) kernels rather than by the `ts` (triangle-on-top-of-square) kernels; see, e.g., Bouwmeester and Langou (2010) for related discussion. This experiment also suits the MKL library, which is faster than PLASMA by about 15%

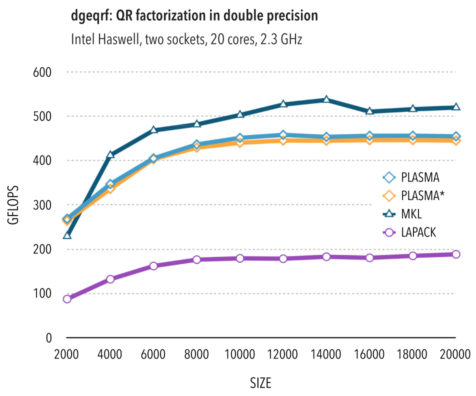

Fig. 52. Performance of dgeqrf on Haswell.

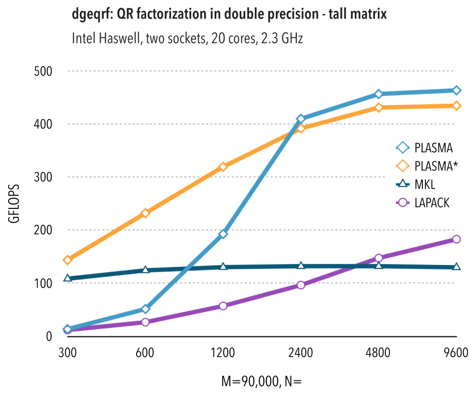

Fig. 53. Performance of dgeqrf on Haswell, tall matrix.

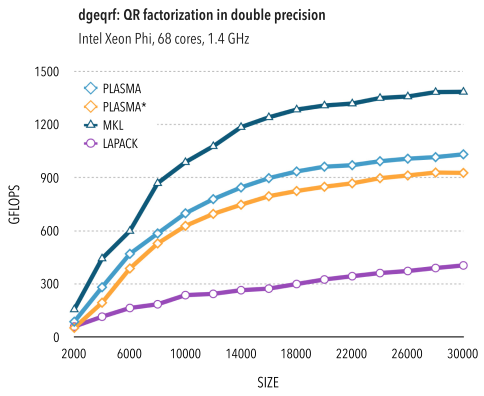

Fig. 54. Performance of dgeqrf on Phi.

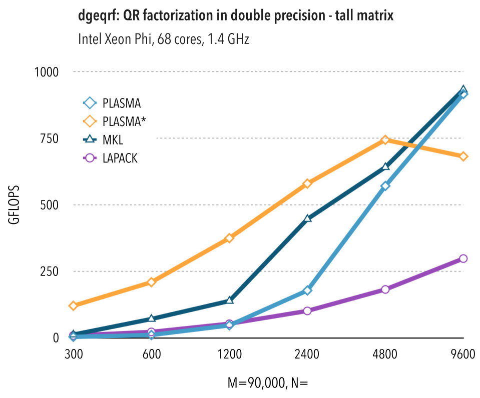

Fig. 55. Performance of dgeqrf on Phi, tall matrix.

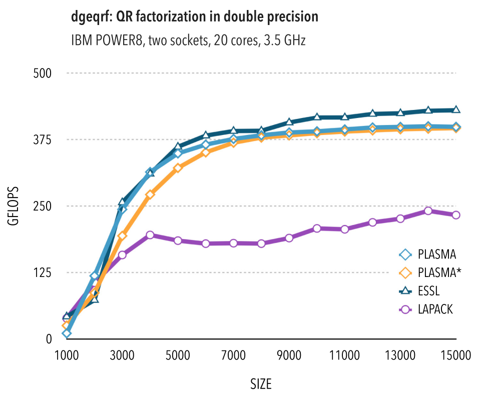

Fig. 56. Performance of dgeqrf on POWER8.

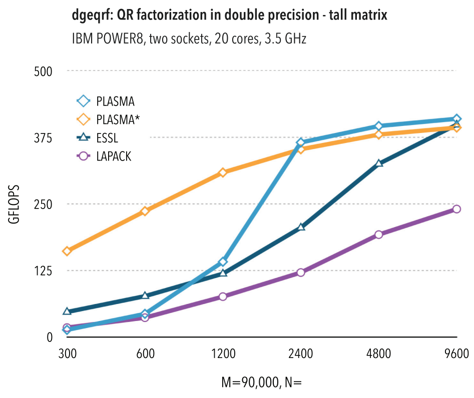

Fig. 57. Performance of dgeqrf on POWER8, tall matrix.

on Haswell and by more than 30% on Phi. It suits also the ESSL, which is about 10% faster than PLASMA on POWER8. Finally, LAPACK with multithreaded BLAS provides significantly lower performance, which is around 40% of the one by PLASMA on Haswell and Phi and around 60% for POWER8.

The situation changes significantly if updating the trailing matrix does not provide enough parallelism, which is the case for matrices with $m \gg n$, also called "tall and skinny" in literature. Our second experiment aims at performance of QR factorization for such matrices, in particular, on a matrix with 90,000 rows, and variable number of columns. In this scenario, it is crucial to introduce parallelism also into elimination of the columns of tiles, as it is done for the tree-based algorithm of PLASMA. Indeed, this algorithm significantly outperforms the standard algorithm in this regime. Nevertheless, for increasing number of matrix columns, the standard algorithm gets enough parallelism and reaches the performance of the tree-based elimination. In our experiments, this has happened already for 8 columns of tiles. LAPACK results follow the trend of the standard PLASMA algorithm, not having parallelism for very skinny matrices and resembling the results for square matrices for increasing number of columns.

A somewhat surprising performance profile was provided by MKL for this experiment. On Haswell (Figure 53), the initial performance for matrix with 300 columns is almost as high as for the tree-based PLASMA algorithm, suggesting that MKL also introduces some parallelism into the panel elimination. However, the performance does not increase for larger matrices, and it got even lower than for LAPACK for the case with 9,600 columns. On Phi, however, the performance of MKL started as low as for the standard PLASMA algorithm, while keeping higher than it for more columns, consistently with the square-matrix results. Performance of ESSL on POWER8 starts between the two PLASMA algorithms, while being lower between 1,200 and 4,800 columns, and matching them for the case with 9,600 columns.

The performance of PLASMA is not particularly sensitive to the tile size parameter (nb) on Haswell, with most of the results obtained using nb = 288. The dependence was stronger on Phi, with 448 and 560 being the optimal values for larger matrices. The results on POWER8 were obtained with nb = 336. The ib parameter for inner blocking inside the kernels for QR factorization has been consistently set to ib = nb/4 on Haswell and Phi, while it has been set to 64 on POWER8.

4 CONCLUSIONS

During the latest major revision of PLASMA, the library has been ported from an in-house developed runtime system; QUARK, to OpenMP tasks with dependencies. While QUARK has features specific to the needs of a numerical library, OpenMP is a more general purpose tool. Consequently, the transition has also led to the redesigning of some algorithms; most notably the LU factorization code.

A comprehensive set of performance benchmarks has been performed, considering three recent multicore shared memory architectures, namely, Haswell, Xeon Phi, and POWER8. In general, the performance of PLASMA is comparable to the optimized vendor libraries; Intel MKL in the case of Intel architectures, and IBM ESSL for POWER8. In addition, the LAPACK library using multithreaded BLAS from the vendor optimized library has been also included for the comparison.

Testing shows that MKL provides higher performance for BLAS routines; especially on the Xeon Phi platform. A significant performance difference in favour of MKL has been also observed for matrix norm computations. However, PLASMA has proven superior to the other libraries for algorithms suited to tile-oriented implementation. This includes the LDL^T factorization, and QR factorization of tall and skinny matrices, where tiling readily provides potential for increased parallelism.

PLASMA offers an important advantage for operations composed of several base algorithms, such as solving a system of linear equations composed of matrix factorization and back-substitution. While executing the corresponding algorithms in a synchronous way suffers from lack of parallelism at the beginning and toward the end of the execution, asynchronous execution allows the merging of these parts of the execution. An operation with a potentially large performance advantage from such merging is computing an inverse of an SPD matrix. Thanks to the asynchronous execution, the performance of PLASMA is typically as much as two times higher than that of the other libraries.

PLASMA 17 currently does not contain all the functionality of the previous version. Specifically, extending the library to eigenvalue problems and singular value decomposition is the current work in progress.

REFERENCES

Jan Ole Aasen. 1971. On the reduction of a symmetric matrix to tridiagonal form. *BIT Numer. Math.* 11, 3 (1971), 233–242. DOI:https://doi.org/10.1007/BF01931804

Maksims Abalenkovs, Negin Bagherpour, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Samuel Relton, Jakub Sistek, David Stevens, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, and Mawussi Zounon. 2017b. *PLASMA 17 Performance Report.* Technical Report 292. LAPACK Working Note. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn292.pdf.

Maksims Abalenkovs, Negin Bagherpour, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Samuel Relton, Jakub Sistek, David Stevens, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, and Mawussi Zounon. 2017a. *PLASMA 17.1 Functionality Report.* Technical Report 293. LAPACK Working Note. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn293.pdf.

Ahmad Abdelfattah, Hartwig Anzt, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Asim YarKhan. 2016. Linear algebra software for large-scale accelerated multicore computing. *Acta Numer.* 25 (2016), 1–160. DOI:https://doi.org/10.1017/S0962492916000015

Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. 2010. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *Proceedings of the International Conference on High Performance Computing for Computational Science.* Springer, 129–138. DOI:https://doi.org/10.1007/978-3-642-19328-6_14

Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012037. DOI:https://doi.org/10.1088/1742-6596/180/1/012037

Edward Anderson, Zhaojun Bai, Christian Bischof, Susan L. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven J. Hammarling, Alan McKenney, and Danny C. Sorensen. 1999. *LAPACK User's Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia.

Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. 2009. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.* 180, 12 (2009), 2526–2533. DOI:https://doi.org/10.1016/j.cpc.2008.11.005

Grey Ballard, Dulceneia Becker, James Demmel, Jack Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. 2014. A communication avoiding symmetric indefinite factorization. *SIAM J. Matrix Anal. Appl.* 35, 4 (2014), 1364–1406. DOI:https://doi.org/10.1137/130929060

Pieter Bellens, Josep M. Perez, Rosa M Badia, and Jesus Labarta. 2006. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC Conference.* IEEE, 5–5. DOI:https://doi.org/10.1109/SC.2006.17

Susan L. Blackford, Jaeyoung Choi, Andrew Cleary, Ed D'Azeuedo, James W. Demmel, Inderjit Dhillon, Jack J. Dongarra, Sven J. Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and Clint R. Whaley. 1997. *ScaLAPACK User's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. 2011. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW'11).* IEEE Computer Society, Washington, DC, 1432–1441. DOI:https://doi.org/10.1109/IPDPS.2011.299

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack J. Dongarra. 2010a. *Distributed*

Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA. Technical Report. Innovative Computing Laboratory, University of Tennessee. Retrieved from http://icl.cs.utk.edu/news_pub/submissions/ut-cs-10-660.pdf.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack J. Dongarra. 2010b. *Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project*. Technical Report 232. LAPACK Working Note. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn232.pdf UT-CS-10-660.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high-performance computing. *Parallel Comput.* 38, 1–2 (2012), 37–51. DOI: https://doi.org/10.1016/j.parco.2011.10.003

Henricus Bouwmeester and Julien Langou. 2010. A critical path approach to analyzing parallelism of algorithmic variants. Application to Cholesky inversion. Retrieved from https://arxiv.org/abs/1010.2000.

Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. 2007. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.* 21, 4 (2007), 457–466. DOI: https://doi.org/10.1177/1094342007084026

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2008. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput.: Pract. Exper.* 20, 13 (2008), 1573–1590. DOI: https://doi.org/10.1002/cpe.1301

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53. DOI: https://doi.org/10.1016/j.parco.2008.10.002

Anthony Castaldo and Clint Whaley. 2010. Scaling LAPACK panel operations using parallel cache assignment. In *ACM SIGPLAN Notices*, Vol. 45. 223–232. DOI: https://doi.org/10.1145/1693453.1693484

James W. Demmel, Laura Grigori, Mark F. Hoemmen, and Julien Langou. 2008. *Communication-optimal Parallel and Sequential QR and LU Factorizations.* Technical Report 204. LAPACK Working Note. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn204.pdf.

Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. 2015. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurr. Comput.: Pract. Exper.* 27, 5 (2015), 1292–1309. DOI: https://doi.org/10.1002/cpe.3306

Jack Dongarra, Mathieu Faverge, Thomas Hérault, Mathias Jacquelin, Julien Langou, and Yves Robert. 2013. Hierarchical QR factorization algorithms for multi-core clusters. *Parallel Comput.* 39, 4–5 (2013), 212–232. DOI: https://doi.org/10.1016/j.parco.2013.01.003

Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. 2014. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurr. Comput.: Pract. Exper.* 26, 7 (2014), 1408–1431. DOI: https://doi.org/10.1002/cpe.3110

Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven J. Hammarling. 1990a. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16 (1990), 1–17.

Jack J. Dongarra, J. Du Croz, Iain S. Duff, and Sven J. Hammarling. 1990b. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16 (1990), 18–28.

Jack J. Dongarra, J. Du Croz, Sven J. Hammarling, and R. Hanson. 1988a. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* 14 (1988), 18–32.

Jack J. Dongarra, J. Du Croz, Sven J. Hammarling, and R. Hanson. 1988b. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.* 14 (1988), 1–17.

Mathieu Faverge, Julien Langou, Yves Robert, and Jack Dongarra. 2016. Bidiagonalization with Parallel Tiled Algorithms. Retrieved from https://arxiv.org/abs/1611.06892.

Fred Gustavson, Lars Karlsson, and Bo Kågström. 2012. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Trans. Math. Softw.* 38, 3 (2012), article No. 17.

Azzam Haidar, Heike Jagode, Asim YarKhan, Phil Vaccaro, Stanimire Tomov, and Jack Dongarra. 2017. Power-aware computing: Measurement, control, and performance analysis for Intel Xeon Phi. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'17).* 1–7. DOI: https://doi.org/10.1109/HPEC.2017.8091085

Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. 2011. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput.: Pract. Exper.* 24, 3 (2011), 305–321. DOI: https://doi.org/10.1002/cpe.1829

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics (SIAM), Philadelphia.

Bo Kågström, Per Ling, and Charles van Loan. 1998. GEMM-based Level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.* 24, 3 (1998), 268–302. DOI: https://doi.org/10.1145/292395.292412

Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. 2008. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Trans. Parallel Distrib. Syst.* 19, 9 (2008), 1175–1186. DOI: https://doi.org/10.1109/TPDS.2007.70813

Jakub Kurzak and Jack Dongarra. 2006. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 147–156.

Jakub Kurzak and Jack Dongarra. 2007. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurr. Comput.: Pract. Exper.* 19, 10 (2007), 1371–1385. DOI: https://doi.org/10.1002/cpe.1164

Jakub Kurzak and Jack Dongarra. 2009. QR factorization for the cell broadband engine. *Sci. Program.* 17, 1–2 (2009), 31–42. DOI: http://dx.doi.org/10.3233/SPR-2009-0268

Jakub Kurzak, Piotr Luszczek, Asim YarKhan, Mathieu Faverge, Julien Langou, Henricus Bouwmeester, and Jack Dongarra. 2013. Multithreading in the PLASMA library. In *Multicore Computing: Algorithms, Architectures, and Applications*, S. Rajasekaran, L. Fiondella, M. Ahmed, R. A. Ammar (Eds.). Chapman and Hall/CRC, 119–141.

Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. 2006. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *Proceedings of the ACM/IEEE SC Conference*. IEEE, 50–50. DOI: https://doi.org/10.1109/SC.2006.30

Charles L. Lawson, Richard J. Hanson, David Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.* 5 (1979), 308–323.

Miroslav Rozložník, Gil Shklarski, and Sivan Toledo. 2011. Partitioned triangular tridiagonalization. *ACM Trans. Math. Softw.* 37, 4 (2011), 1–16. DOI: http://dx.doi.org/10.1145/1916461.1916462

Herb Sutter. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's J.* 30, 3 (2005), 202–210.

Ichitaro Yamazaki, Jakub Kurzak, Panruo Wu, Mawussi Zounon, and Jack Dongarra. 2018. Symmetric indefinite linear solver using OpenMP task on multicore architecture. *IEEE Trans. Parallel Distrib. Syst.* 29, 8 (2018), 1879–1892. DOI: https://doi.org/10.1109/TPDS.2018.2808964

Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2016. Porting the PLASMA numerical library to the OpenMP standard. *Int. J. Parallel Program.* 45, 3 (2016), 1–22. DOI: https://doi.org/10.1007/s10766-016-0441-6

Received October 2017; revised June 2018; accepted July 2018