PLASMA 17 Performance Report

Linear Systems and Least Squares
Haswell, Knights Landing, POWER8

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
Mawussi Zounon

Department of Electrical Engineering & Computer Science, University of Tennessee
School of Mathematics, The University of Manchester

June 7, 2017

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

PLASMA (Parallel Linear Algebra for Multicore Architectures) is a dense linear algebra package at the forefront of multicore computing. PLASMA is designed to deliver the highest possible performance from a system with multiple sockets of multicore processors. PLASMA achieves this objective by combining state of the art solutions in parallel algorithms, scheduling, and software engineering. PLASMA currently offers a collection of routines for solving linear systems of equations and least square problems.

1.1 Tile Matrix Layout

PLASMA lays out matrices in square tiles of relatively small size, such that each tile occupies a continuous memory region. Tiles are loaded to the cache memory efficiently with little risk of eviction while being processed. The use of tile layout minimizes conflict cache misses, TLB misses, and false sharing, and maximizes the potential for prefetching. PLASMA contains parallel and cache efficient routines for converting between the conventional LAPACK layout and the tile layout.

1.2 Tile Algorithms

PLASMA introduces new 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 evicted back to the main memory. Operations on small tiles create fine grained parallelism providing enough work to keep a large number of cores busy.

1.3 Dynamic Scheduling

PLASMA relies on runtime scheduling of parallel 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 referred to as data-driven scheduling. The concept is related closely to the idea of expressing computation through a task graph, often referred to as the DAG (Directed Acyclic Graph), 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 [11]

CHAPTER 2

Parallel BLAS

PLASMA contains a full set of Level 3 BLAS routines. BLAS routines in PLASMA are parallelized by tiling. Their implementations are mainly straightforward loop nests. Parallel tasks are essentially calls to serial BLAS. Most routines achieve good serial performance and scale well.

2.1 Haswell

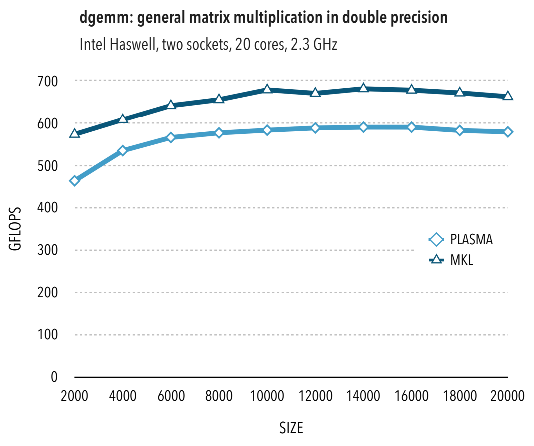

Figure 2.1: Performance of dgemm on Haswell.

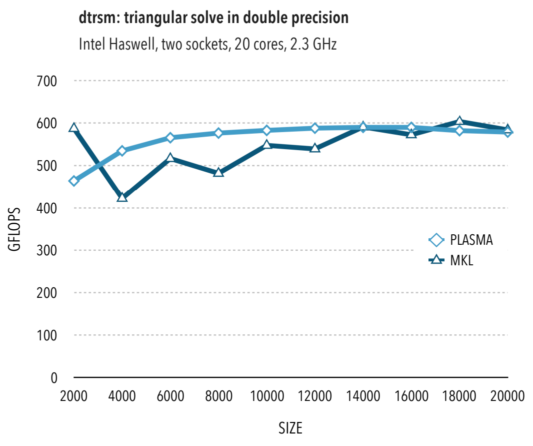

Figure 2.2: Performance of dtrsm on Haswell.

2.2 Knights Landing

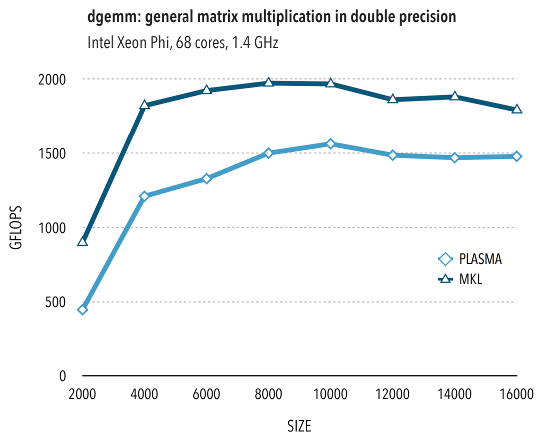

Figure 2.3: Performance of dgemm on KNL.

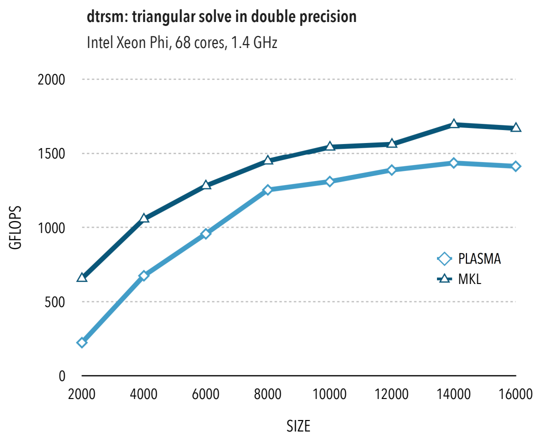

Figure 2.4: Performance of dtrsm on KNL.

2.3 POWER8

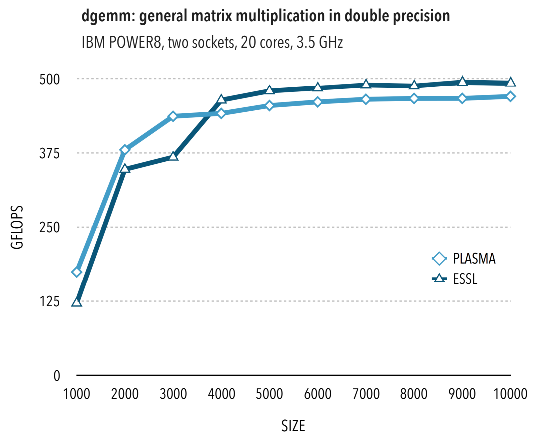

Figure 2.5: Performance of dgemm on POWER8.

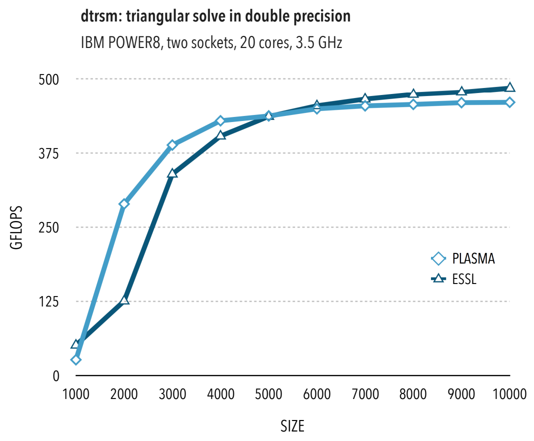

Figure 2.6: Performance of dtrsm on POWER8.

CHAPTER 3

Parallel Norms

PLASMA contains a set of routines for computing matrix norms: *max*, *one*, *infinity*, and *Frobenius* norm. PLASMA also contains a routine for computing the *max* norm for each column of a matrix. PLASMA employs tiling for increased parallelism of the norm computations. While being mostly memory bound, PLASMA norm routines still benefit from multithreading, as usually a single core cannot saturate the memory bandwidth.

3.1 Haswell

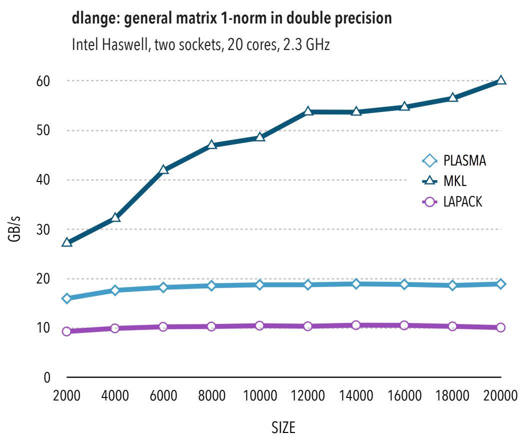

Figure 3.1: Performance of dlange on Haswell.

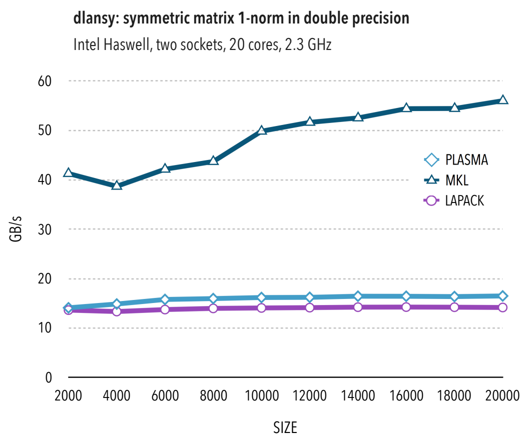

Figure 3.2: Performance of dlansy on Haswell.

3.2 Knights Landing

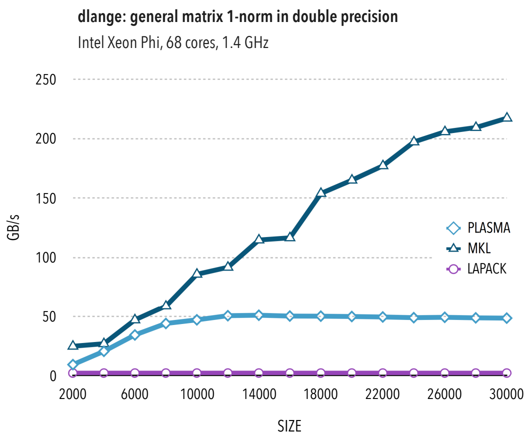

Figure 3.3: Performance of dlange on KNL.

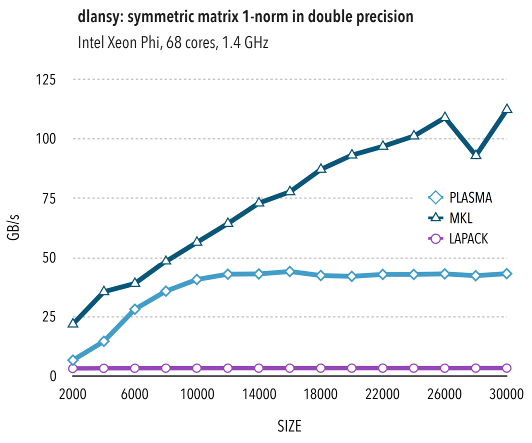

Figure 3.4: Performance of dlansy on KNL.

3.3 POWER8

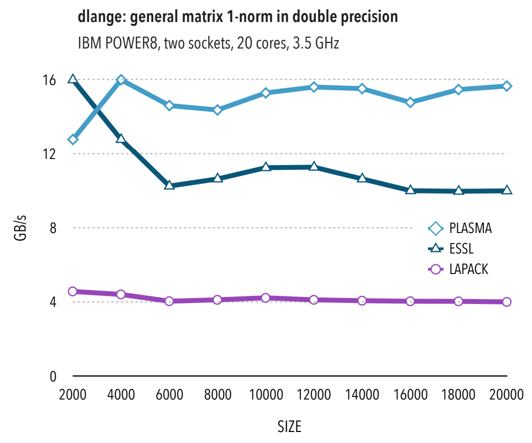

Figure 3.5: Performance of dlange on POWER8.

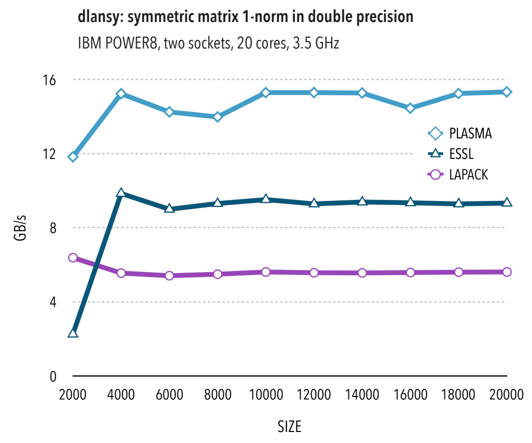

Figure 3.6: Performance of dlansy on POWER8.

CHAPTER 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 in the LU factorization with partial (row) pivoting, routines for solving symmetric positive definite systems rely on the Cholesky factorization, and routines for solving symmetric (non positive definite) systems rely on the Aasen algorithm. The Cholesky factorization is trivial to implement using tile operations. The other routines pose some unique challenges and are discussed in more detail.

4.1 LU Implementation

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 algorithms 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 et al. [5], and the work on parallel recursive panel factorization by Dongarra et al. [9]. Also, the survey by Donfack et al. [7] provides a good overview of different implementations of the LU factorization.

The LU panel factorization in PLASMA 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.

The complete LU factorization, including the panel factorization, and the updates to the trailing submatrix, is multithreaded differently then 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.

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 `#taskwait` clause, before exiting the nested parallel region.

Since the factorization resolves dependencies at a different granularity than other operations, such as the layout translation that precedes it, and the triangular solve that follows it, `#taskwait` synchronizations are required at the boundaries, which breaks the pristine model of asynchronous execution, but cannot easily be avoided.

4.2 LDLT Implementation

To solve a symmetric indefinite linear system, PLASMA first reduces the symmetric matrix into a band form based on the tiled Aasen's algorithm [?]. 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 accumulate 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 in the next section. 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 limitted amount of parallelism that can be exploited.

4.3 Performance

4.3.1 Haswell

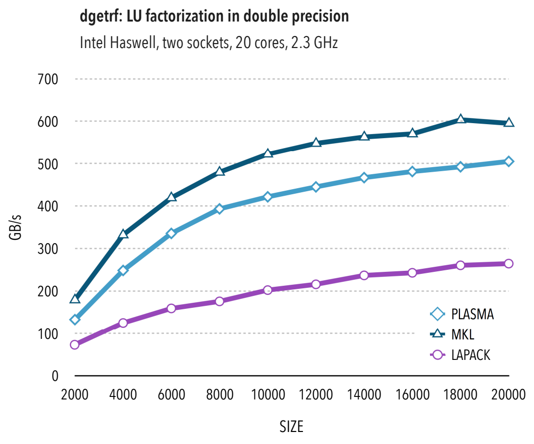

Figure 4.1: Performance of dgetrf on Haswell.

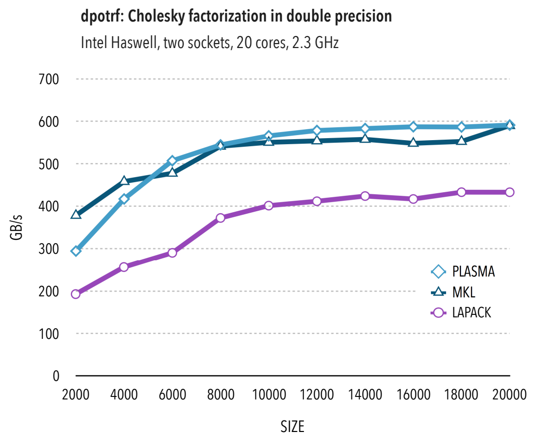

Figure 4.2: Performance of dpotrf on Haswell.

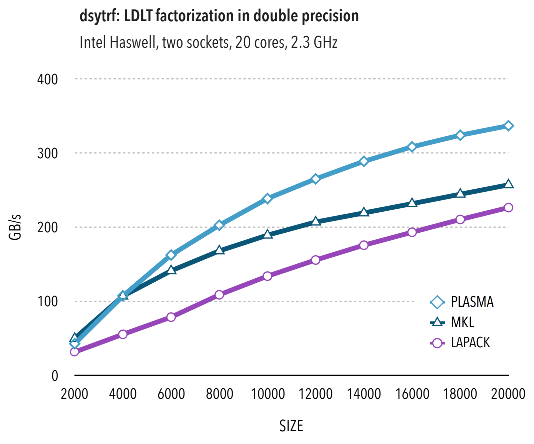

Figure 4.3: Performance of dsytrf on Haswell.

4.3.2 Knights Landing

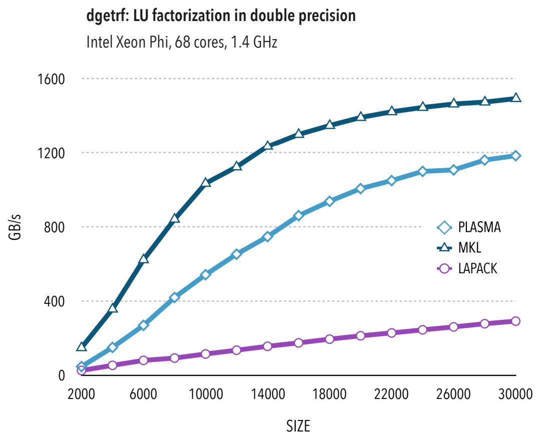

Figure 4.4: Performance of dgetrf on KNL.

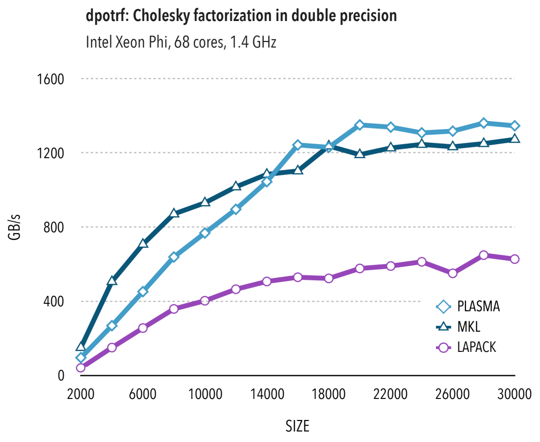

Figure 4.5: Performance of dpotrf on KNL.

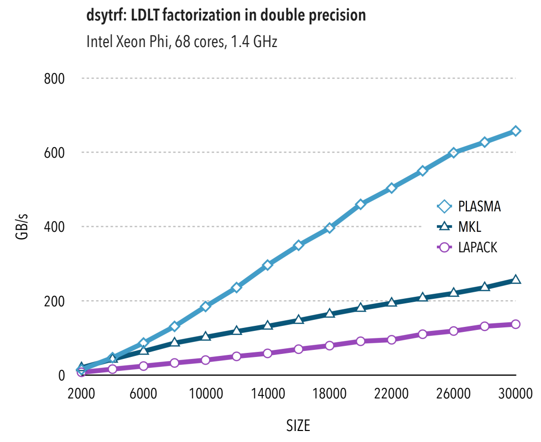

Figure 4.6: Performance of dsytrf on KNL.

4.3.3 POWER8

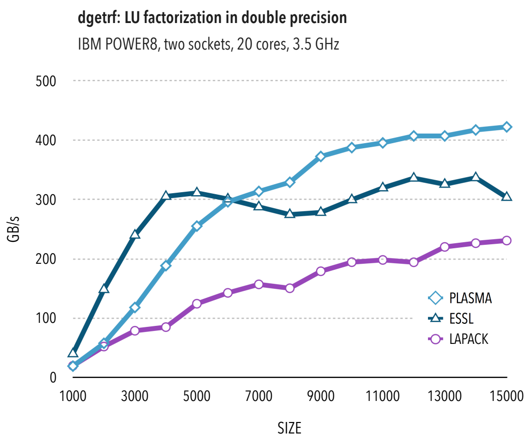

Figure 4.7: Performance of dgetrf on POWER8.

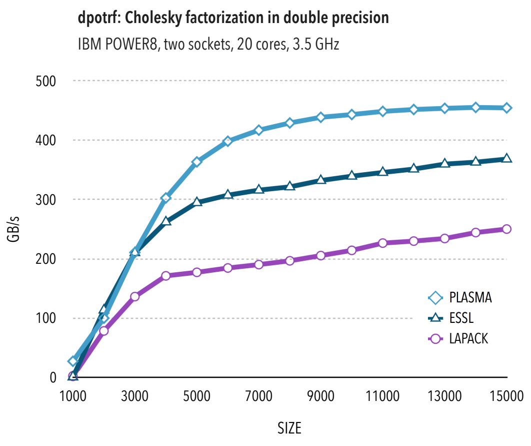

Figure 4.8: Performance of dpotrf on POWER8.

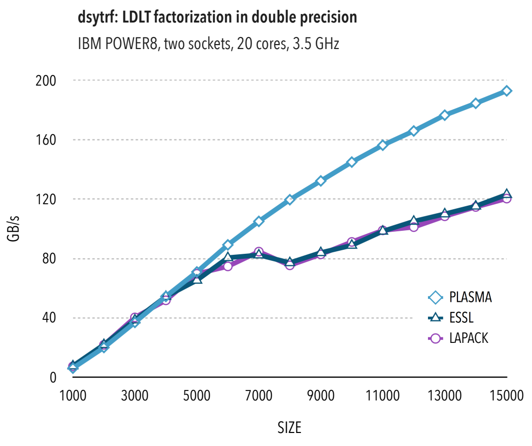

Figure 4.9: Performance of dsytrf on POWER8.

CHAPTER 5

Matrix Inversion

PLASMA contains a set of routines for explicitly computing an inverse of a matrix. Routines for inverting general matrices rely on the LU factorization with partial (row) pivoting, routines for inverting symmetric positive definitey matrices rely on the Cholesky factorization. Generally, the implementations are straightforward and involve: factorization of the matrix, computing an inverse of a triangular matrix, and multiplication of triangular matrices. In the case of the LU factorization, this is performed in three distinct steps, with synchronizations in between, because pivoting prevents any overlap of the diffrent stages. However, in the case of the Cholesky factorization all the three stages can be merged into one operation, providing for very efficient scheduling, and leading to high performance implementations [1, 2].

5.1 Haswell

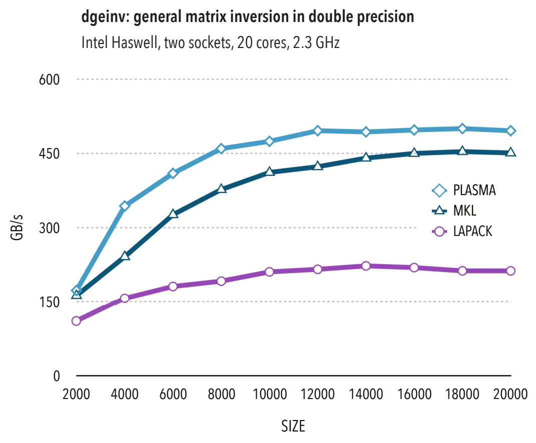

Figure 5.1: Performance of dgeinv on Haswell.

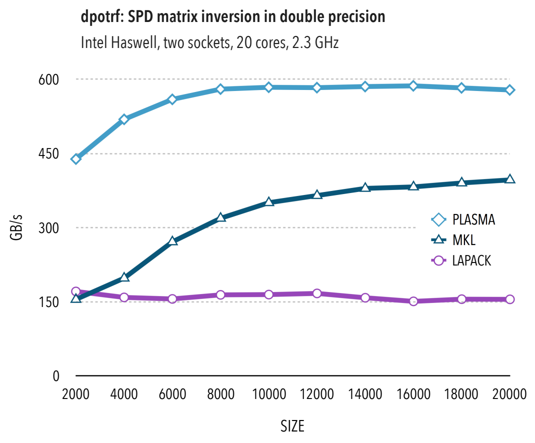

Figure 5.2: Performance of dpoinv on Haswell.

5.2 Knights Landing

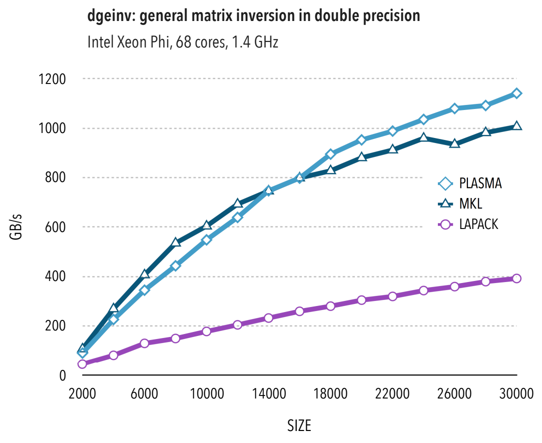

Figure 5.3: Performance of dgeinv on KNL.

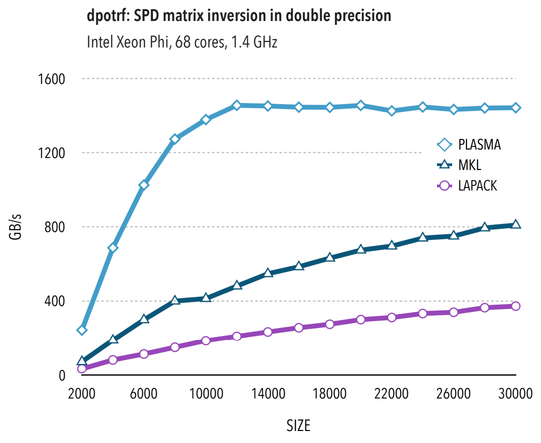

Figure 5.4: Performance of dpoinv on KNL.

5.3 POWER8

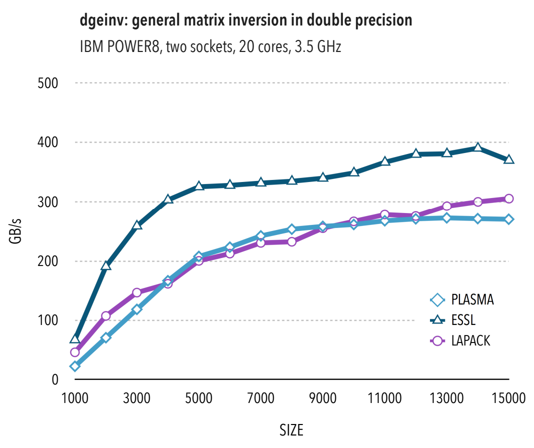

Figure 5.5: Performance of dgeinv on POWER8.

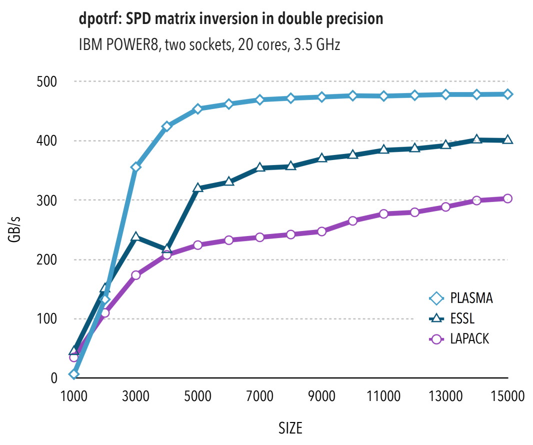

Figure 5.6: Performance of dpoinv on POWER8.

CHAPTER 6

Least Squares

PLASMA contains routines for solving overdetermined and underdetermined systems of linear equations, using on QR and LQ factorizations, based on Householder transformations. PLASMA implementations are profoundly different from LAPACK. While LAPACK reduces the input matrices by columns, PLASMA reduces the input matrices by tiles. This approach produces algorithms with much higher levels of parallelism and excellent scheduling properties [3, 4]. 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.

PLASMA includes support for QR factorization of tall and skinny matrices, for which number of rows is much larger than number of columns. 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 [6], and extended e.g. in [8]. Tree-based Householder reductions were recently used for SVD in [10].

Since different trees may be beneficial in different scenarios, 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.

In the charts to follow, “PLASMA” (no asterisk) refers to runs with reduction of tiles in the panel in consecutive order, while “PLASMA*” (with asterisk) refers to application of tree reduction to the panel tiles.

6.1 Haswell

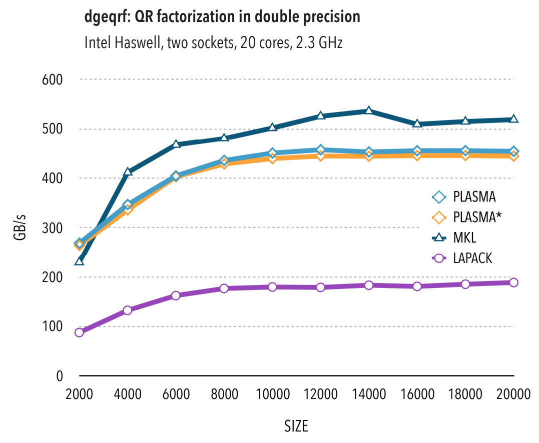

Figure 6.1: Performance of dgeqrf on Haswell.

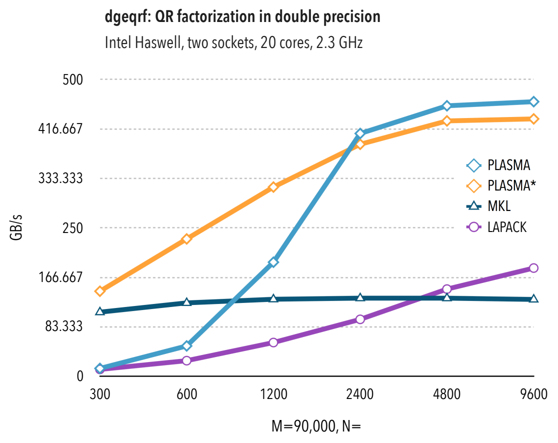

Figure 6.2: Performance of dgeqrf on Haswell.

6.2 Knights Landing

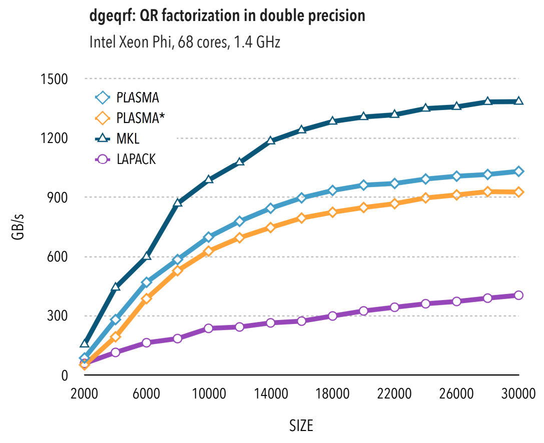

Figure 6.3: Performance of dgeqrf on KNL.

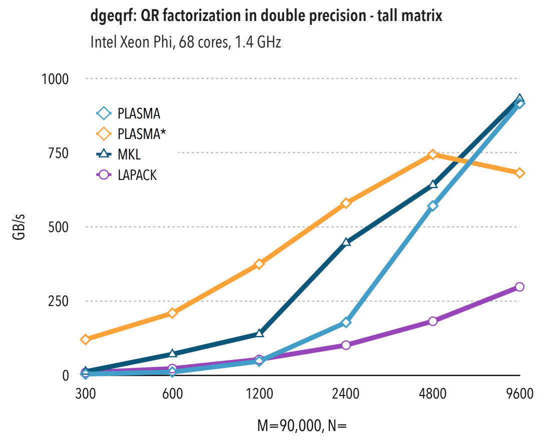

Figure 6.4: Performance of dgeqrf on KNL.

6.3 POWER8

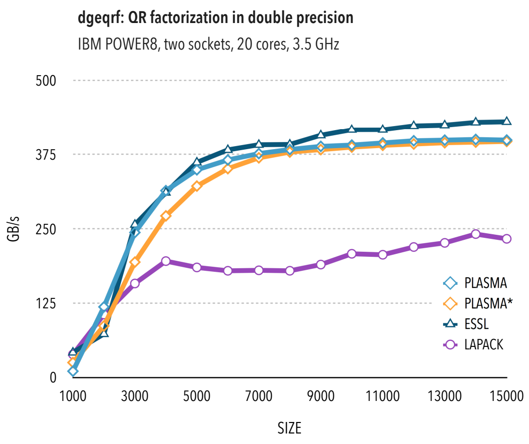

Figure 6.5: Performance of dgeqrf on POWER8.

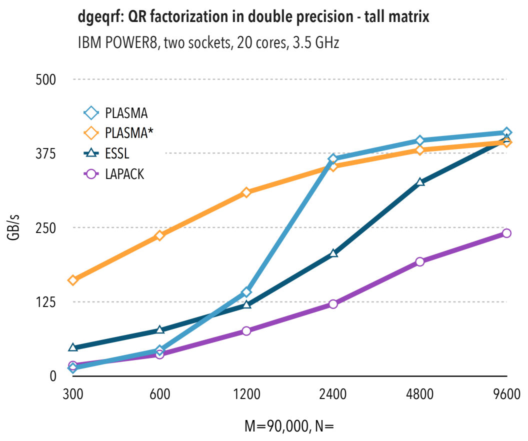

Figure 6.6: Performance of dgeqrf on POWER8.

CHAPTER 7

Appendices

7.1 MKL Blocking Factors for KNL

Table 7.1 shows MKL blocking factors for the KNL processor, i.e., the `NB` value returned by the `ILAENV` routine. While for routines like `DGETRF`, `DGEQRF`, and `DPOTRF`, the value of `NB` increases with the matrix size, for others, like `DSYTRF`, `DTRTRI`, and `DLAUUM`, it does not, which helps explain the unexpectedly low performance of the latter routines.

Most likely, the performance could be improved by using larger blocking for larger matrix sizes. This could be done by providing a custom implementation of the `ILAENV` routine. While this is not a difficult task, tuning of `NB` is beyond the scope of this report.

size	DGERRF	DGEQRF	DPOTRF	DSYTRF	DTRTRI	DLAUUM
2000	48	96	80	64	64	32
4000	96	128	112	64	64	32
6000	244	128	208	64	64	32
8000	244	128	224	64	64	32
10000	244	128	224	64	64	32
12000	244	128	448	64	64	32
14000	244	128	448	64	64	32
16000	336	128	448	64	64	32
18000	336	128	448	64	64	32
20000	336	128	536	64	64	32
22000	336	128	536	64	64	32
24000	336	128	536	64	64	32
26000	336	128	536	64	64	32
28000	336	128	536	64	64	32
30000	336	128	536	64	64	32

Table 7.1: MKL blocking factors for KNL

7.2 Traces

7.2.1 Haswell

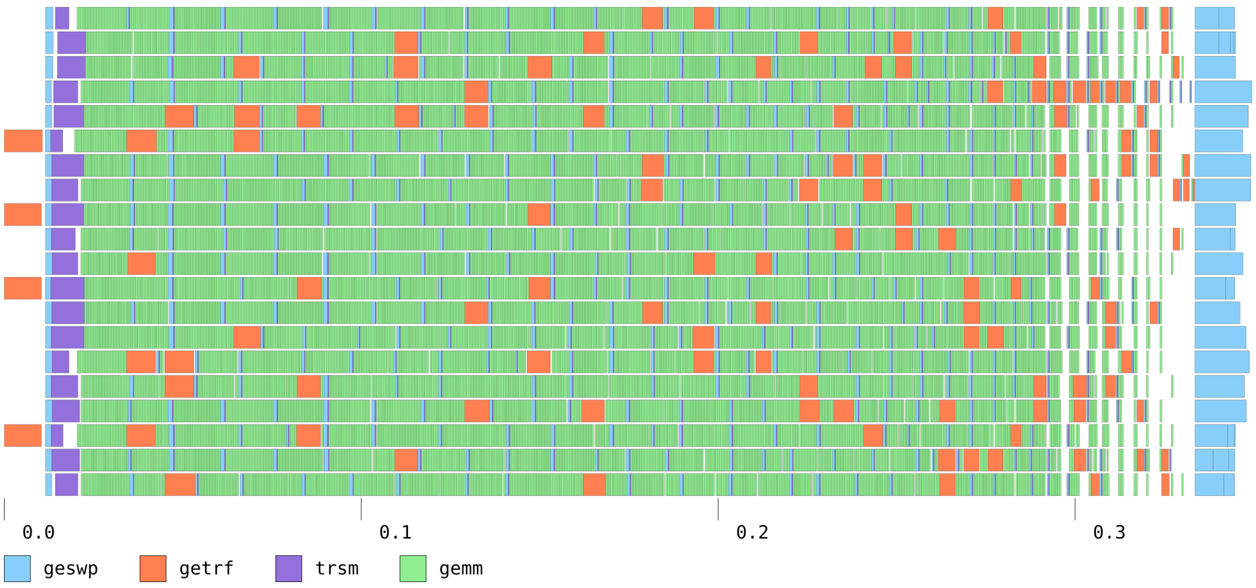

Figure 7.1: Trace of plasma_dgetrf() on Haswell:
`numactl --interleave=all ./test dgetrf --dim=6144 --nb=192 --ib=28 --mtpf=4`

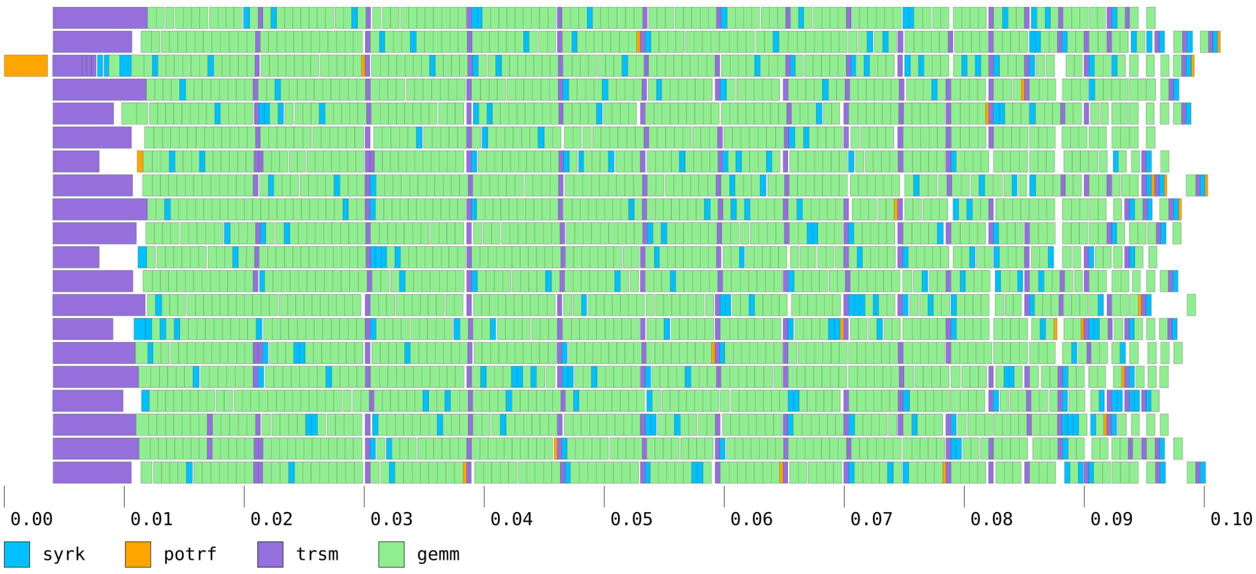

Figure 7.2: Trace of plasma_dpotrf() on Haswell:
`numactl --interleave=all ./test dpotrf --dim=5376 --nb=224`

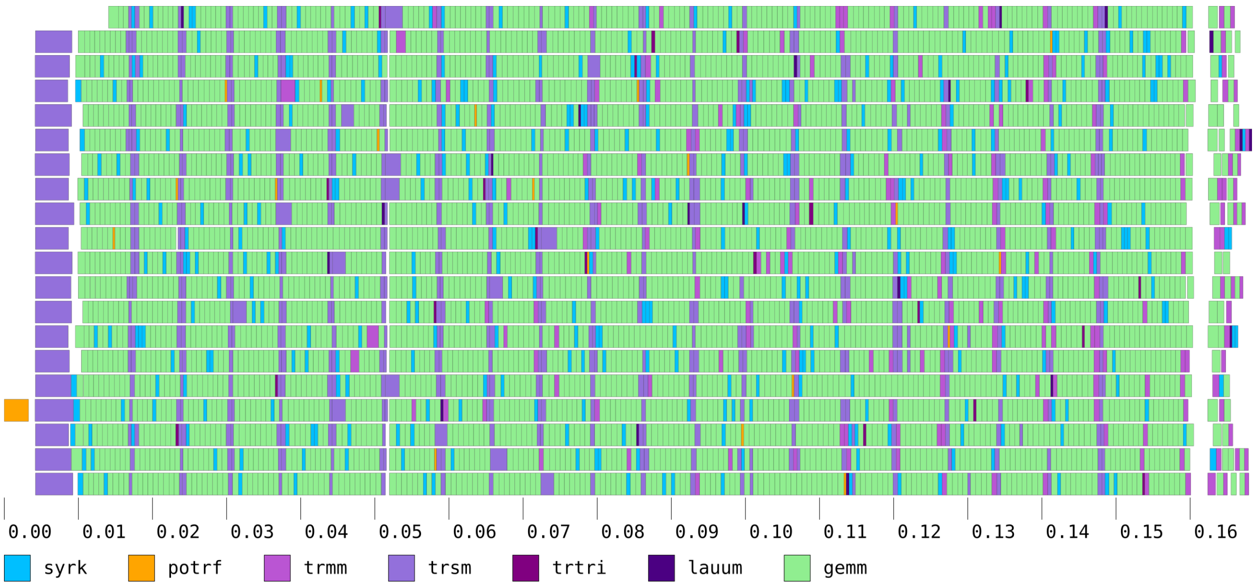

Figure 7.3: Trace of plasma_dpoinv() on Haswell:
`numactl --interleave=all ./test dpoinv --dim=4480 --nb=224`

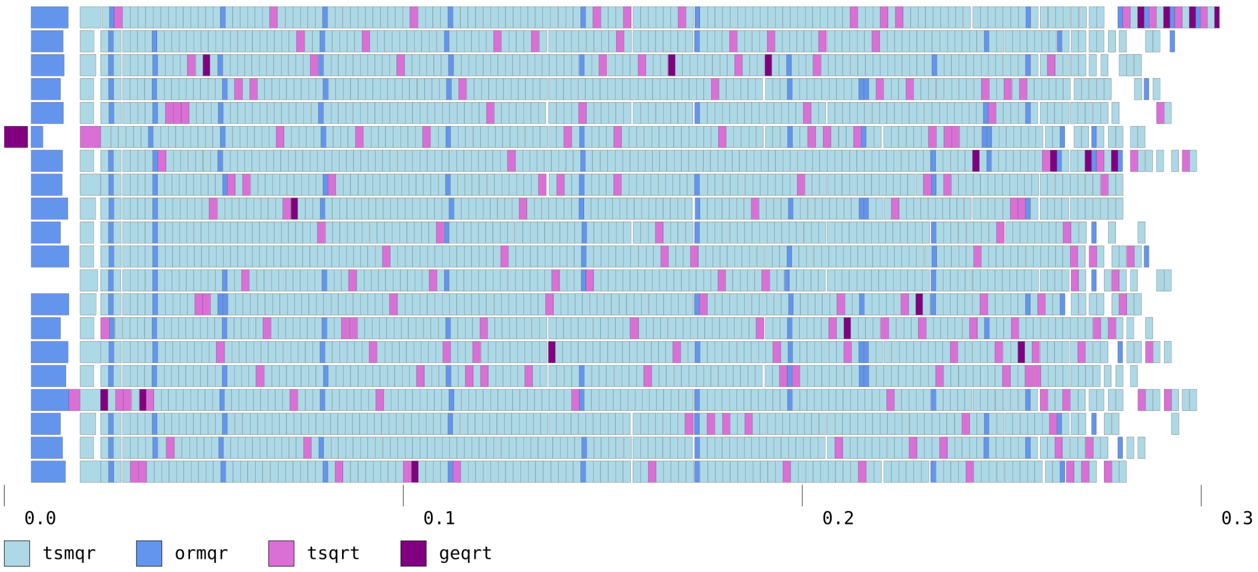

Figure 7.4: Trace of plasma_dgeqrf() on Haswell:
`numactl --interleave=all ./test dgeqrf --dim=4480 --nb=224 --ib=56`

7.2.2 Knights Landing

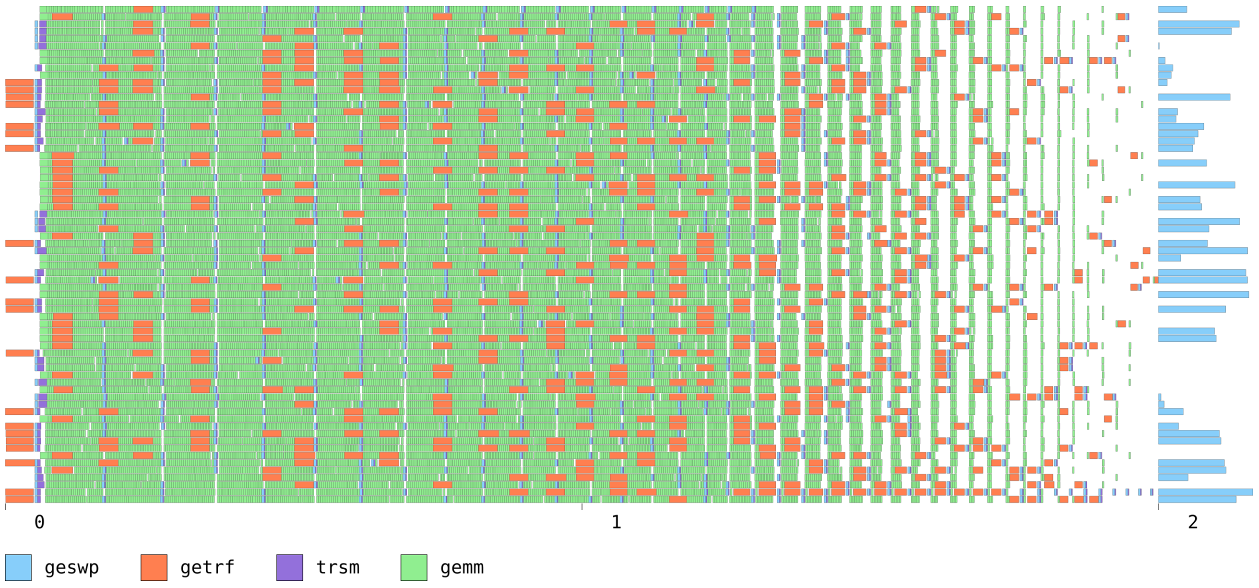

Figure 7.5: Trace of plasma_dgetrf() on KNL:
`numactl -m 1 ./test dgetrf --dim=14112 --nb=336 --ib=40 --mtpf=20`

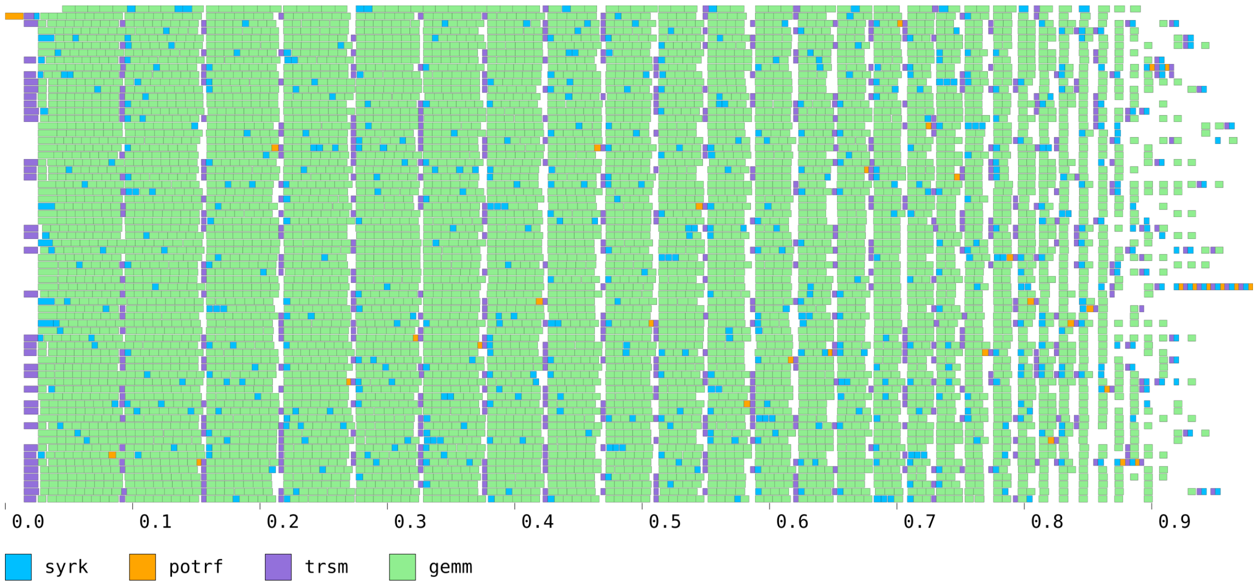

Figure 7.6: Trace of plasma_dpotrf() on KNL:
`numactl -m 1 ./test dpotrf --dim=15680 --nb=448`

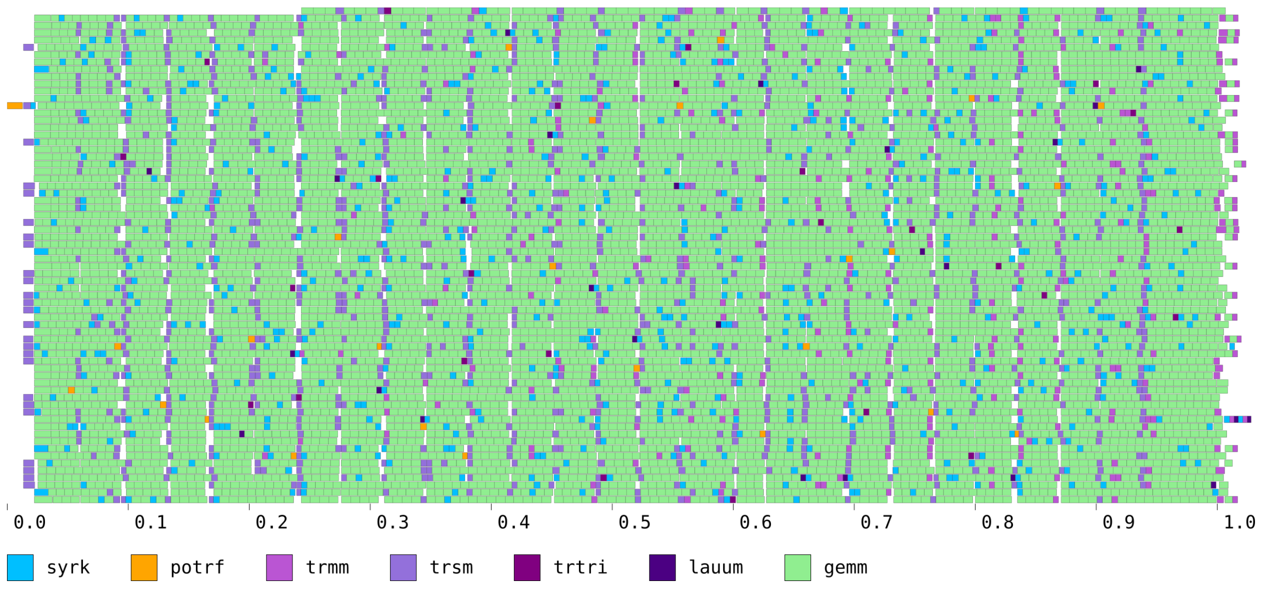

Figure 7.7: Trace of plasma_dpoinv() on KNL:
`numactl -m 1 ./test dpoinv --dim=11648 --nb=448`

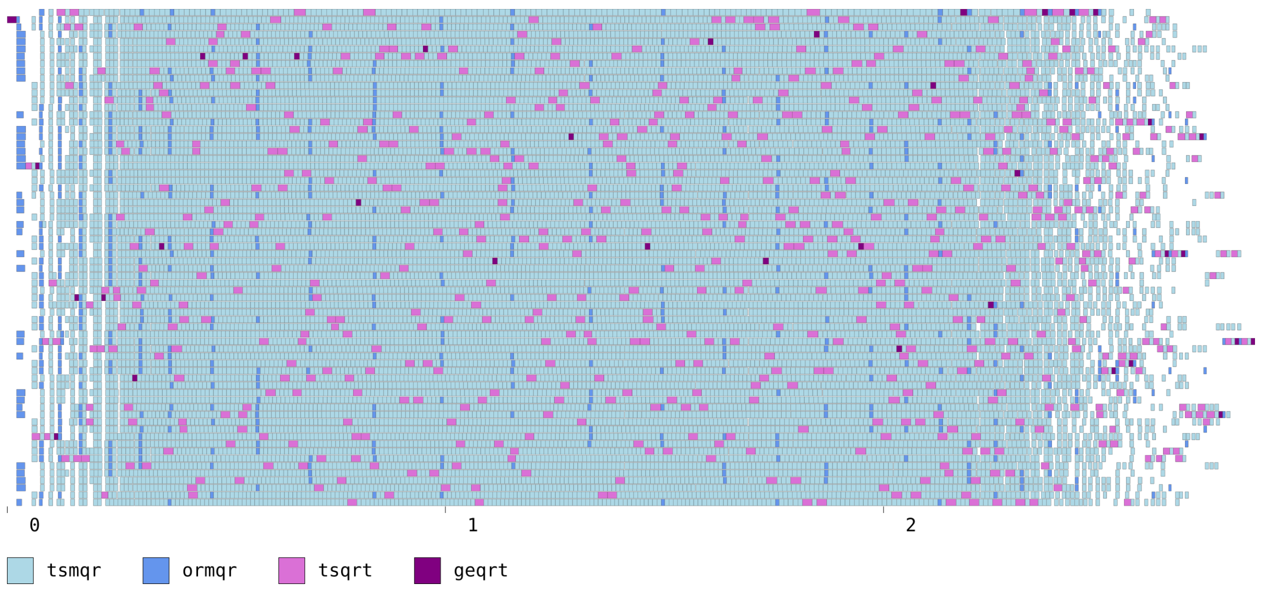

Figure 7.8: Trace of plasma_dgeqrf() on KNL:
`numactl -m 1 ./test dgeqrf --dim=12096 --nb=336 --ib=112`

7.2.3 POWER8

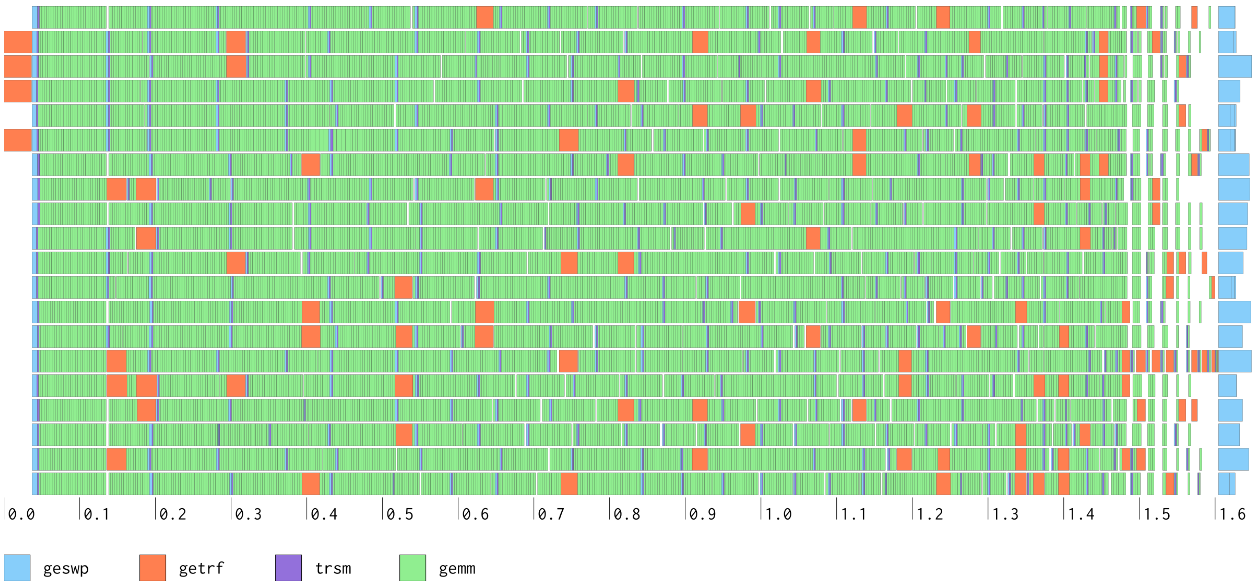

Figure 7.9: Trace of plasma_dgetrf() on POWER8:
`./test dgetrf --dim=10000 --nb=336 --ib=32 --mtpf=4`

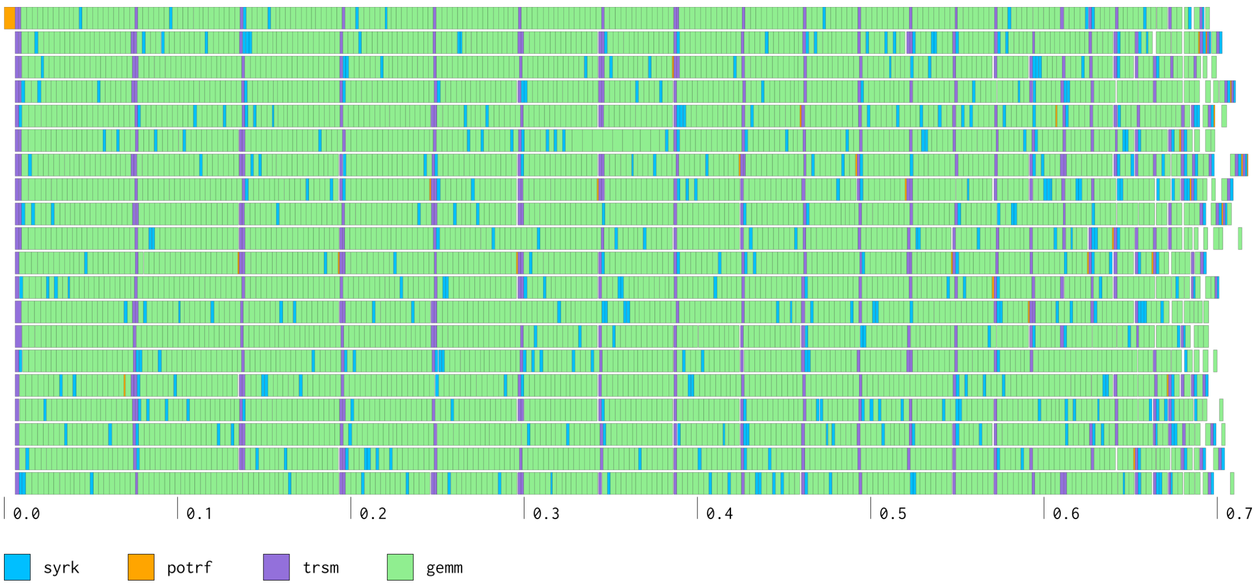

Figure 7.10: Trace of plasma_dpotrf() on POWER8:
`./test dpotrf --dim=10000 --nb=336`

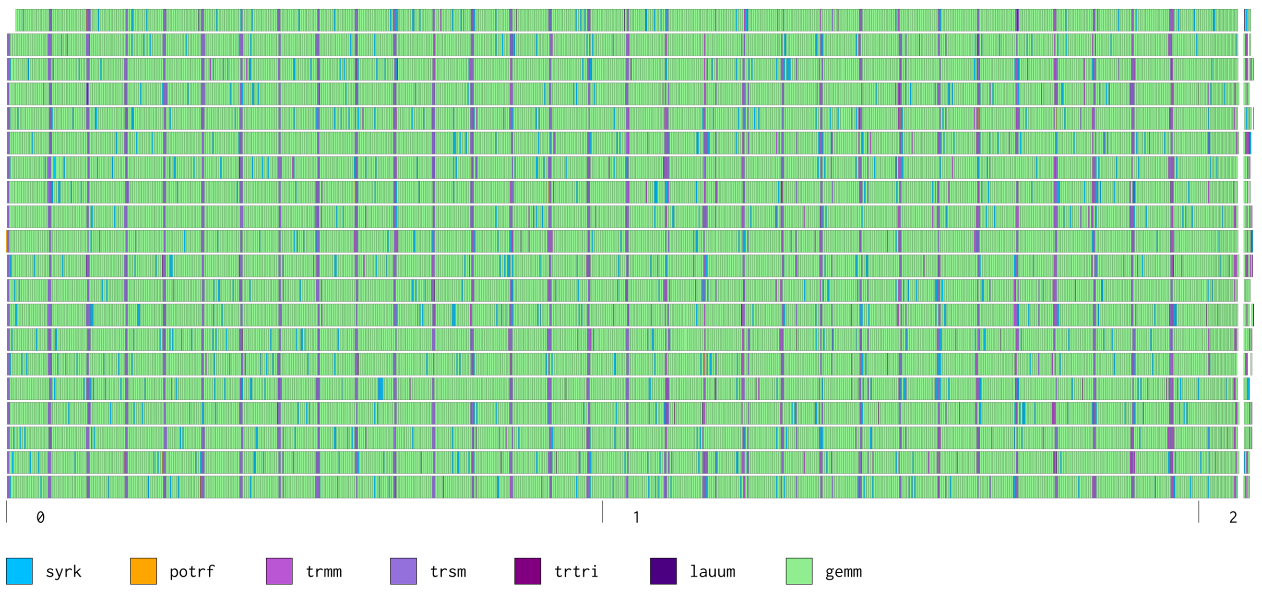

Figure 7.11: Trace of plasma_dpoinv() on POWER8:
`./test dpoinv --dim=10000 --nb=336`

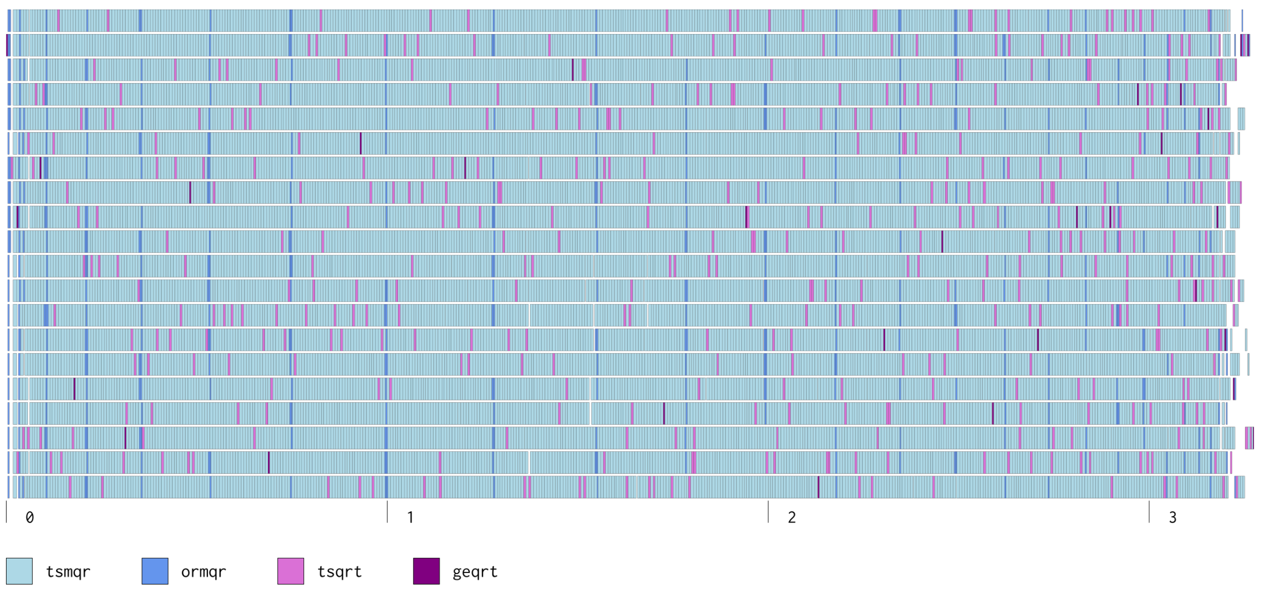

Figure 7.12: Trace of plasma_dgeqrf() on POWER8:
`./test dgeqrf --dim=10000 --nb=336`

Acknowledgments

We thank the Intel Corporation for their generous hardware donation and continuous financial support. We thank the Oak Ridge Leadership Computing Facility for providing access to the POWER8 system.

Bibliography

[1] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In *International Conference on High Performance Computing for Computational Science*, pages 129–138. Springer, 2010. DOI: 10.1007/978-3-642-19328-6˙14.

[2] Henricus Bouwmeester and Julien Langou. A critical path approach to analyzing parallelism of algorithmic variants. application to cholesky inversion. *arXiv preprint arXiv:1010.2000*, 2010. URL https://arxiv.org/abs/1010.2000.

[3] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

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

[5] Anthony Castaldo and Clint Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM Sigplan Notices*, volume 45, pages 223–232. ACM, 2010. DOI: 10.1145/1693453.1693484.

[6] James W. Demmel, Laura Grigori, Mark F. Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report 204, LAPACK Working Note, August 2008. URL http://www.netlib.org/lapack/lawnspdf/lawn204.pdf.

[7] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience*, 27(5):1292–1309, 2015. DOI: 10.1002/cpe.3306.

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

[9] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014. DOI: 10.1002/cpe.3110.

[10] Mathieu Faverge, Julien Langou, Yves Robert, and Jack Dongarra. Bidiagonalization with parallel tiled algorithms, 2016. URL http://arxiv.org/abs/1611.06892.

[11] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming*, pages 1–22, 2016. DOI: 10.1007/s10766-016-0441-6.