Parallel Reduction to Condensed Forms for Symmetric Eigenvalue Problems using Aggregated Fine-Grained and Memory-Aware Kernels

Azzam Haidar
University of Tennessee
1122 Volunteer Blvd
Knoxville, TN
haidar@eecs.utk.edu

Hatem Ltaief
KAUST Supercomputing Laboratory
Thuwal, Saudi Arabia
Hatem.Ltaief@kaust.edu.sa

Jack Dongarra[*] [†] [‡]
University of Tennessee
1122 Volunteer Blvd
Knoxville, TN
dongarra@eecs.utk.edu

ABSTRACT

This paper introduces a novel implementation in reducing a symmetric dense matrix to tridiagonal form, which is the preprocessing step toward solving symmetric eigenvalue problems. Based on *tile algorithms*, the reduction follows a two-stage approach, where the tile matrix is first reduced to symmetric band form prior to the final condensed structure. The challenging trade-off between algorithmic performance and task granularity has been tackled through a *grouping* technique, which consists of aggregating fine-grained and memory-aware computational tasks during both stages, while sustaining the applications overall high performance. A dynamic runtime environment system then schedules the different tasks in an out-of-order fashion. The performance for the tridiagonal reduction reported in this paper is unprecedented. Our implementation results in up to 50-fold and 12-fold improvement (130 Gflop/s) compared to the equivalent routines from LAPACK V3.2 and Intel MKL V10.3, respectively, on an eight socket hexa-core AMD Opteron multicore shared-memory system with a matrix size of 24000×24000.

1. INTRODUCTION

After traditional processor designs hit the edge of their power capabilities, the gap between theoretical peak performance and the actual performance realized by full applications has never been so substantial. Indeed, the current hardware and software landscapes show that (1) clock frequencies are now capped below 4 GHz and trending even downward, (2) latencies in key areas (e.g., memory access, bus) are expected to remain relatively stagnant and (3) the level of parallelism in the state-of-the-art numerical softwares is not sufficient to handle the multicore shared-memory systems available today. The exascale roadmap [12] has further exacerbated the challenges explained above, where complex systems of heterogeneous computational resources with hundreds of thousands of cores affirm to be the next ubiquitous generation of computing platforms. Therefore, designs of new numerical algorithms are not optional anymore and become paramount to efficiently address future hardware issues. The PLASMA [24] and FLAME [25] projects started a few years ago to revise the commonly accepted linear algebra algorithms present in LAPACK [4]. Successful results were reported on multicore architectures for one-sided factorizations e.g., QR, LU and Cholesky [3, 8, 9, 20, 21]. However, the performance of two-sided transformations e.g., Hessenberg, tridiagonal and bidiagonal reductions are very limited on multicore architectures and achieve only a small fraction of the systems theoretical peak. This is clearly still an open research problem. And this is primarily due to the inherently problematic portion of those reductions i.e., the panel factorizations mainly composed of expensive memory-bound operations, which impedes the overall high performance.

In particular, the authors focus on the tridiagonal reduction (TRD), which is the preprocessing step toward solving symmetric eigenvalue problems [13, 14, 23]. The TRD is actually the most time consuming part when only eigenvalues are required and can take more than 90% of the elapsed time. In fact, this paper introduces a novel implementation (PLASMA-xSYTRD) for reducing a symmetric dense matrix to tridiagonal form. Based on *tile algorithms*, the matrix is split into square tiles, where each data entry within a tile is contiguous in memory. Following a two-stage approach, the tile matrix is first reduced to symmetric band form prior to the final condensed structure. The general algorithm is then broken into tasks and proceeds on top of tile data layout, which eventually generates a directed acyclic graph (DAG) [7, 10], where nodes represent tasks and edges describe the data dependencies between them. The algorithm employs high performance compute intensive kernels for the first stage, which necessitates tasks with a coarse granularity in order to extract a substantial percentage of the machines theoretical peak performance. Those coarse-grained tasks engender a wide band structure of the matrix to be further reduced during the second stage. On the other hand, the bulge chasing procedure in the second stage annihilates, element-wise, the extra off-diagonal entries, and because of potentially being memory-bound, it demands a matrix with a small band size as input to prevent the application overall performance from dropping. This challenging trade-off between algorithmic performance and task granularity has been tackled through a *grouping* technique, which consists of aggregat-

[*]Research reported here was partially supported by the National Science Foundation, Department of Energy and Microsoft Research.

[†]Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA

[‡]School of Mathematics & School of Computer Science, University of Manchester, United Kingdom

ing fine-grained tasks together during the first stage while sustaining the application's overall high performance. Furthermore, new memory-aware computational kernels have been developed for the second stage, which are highly optimized for cache reuse and enable us to run at the cache speed by appropriately fitting the data into the small core caches. The dynamic runtime environment system QUARK (available in PLASMA) schedules the tasks from both stages across the processing units in an out-of-order fashion. Since the different computational kernels may operate on the same matrix data, a framework based on function dependencies tracks the data areas accessed during both stages and detects and prevents any overlapping region hazards to ensure the dependencies are not violated. A thread locality mechanism has also been implemented at the runtime level in order to dramatically decrease the memory bus traffic, especially for the memory-bound stage.

The remainder of this paper is organized as follows: Section 2 gives a detailed overview of previous projects in this area. Section 3 lays out clearly our research contributions. Section 4 recalls the bottlenecks seen in the standard TRD algorithm. Section 5 explains how the two-stage TRD approach using tile algorithms overcomes those bottlenecks. Section 6 describes the new fine-grained and cache-friendly numerical kernels used during both stages. Section 7 presents the new grouping technique. Section 8 gives some implementation details of our proposed PLASMA-xSYTRD. The performance numbers are shown in Section 9, comparing our implementation with the state-of-the-art, high performance dense linear algebra software libraries, LAPACK V3.2 [4] and Intel MKL V10.3 [1], an open-source and a commercial package, respectively. Finally, Section 10 summarizes the results of this paper and presents the ongoing work.

2. RELATED WORK

The idea of splitting the matrix reduction phase to condensed forms with multiple stages in the context of eigenvalue problems and singular value decomposition has been extensively studied in the past.

Grimes and Simon [15] reported the first time [1] was used a two-step reduction in their out-of-core solver for generalized symmetric eigenvalue problems. Later, Lang [17] used a multiple-stage implementation to reduce a matrix to tridiagonal, bidiagonal and Hessenberg forms. The actual number of stages necessary to reduce the matrix to the corresponding form is a tunable parameter, which depends on the underlying hardware architecture. The general idea is to cast expensive memory operations, occurring during the panel factorization into fast compute intensive ones. Later, Bischof et al. [5] integrated some of the work described above into a framework called Successive Band Reductions (SBR). SBR is used to reduce a symmetric dense matrix to tridiagonal form, required to solve the symmetric eigenvalue problem (SEVP). This toolbox applies two-sided orthogonal transformations to the matrix based on Householder reflectors and successively reduces the matrix bandwidth size until a suitable one is reached. The off-diagonal elements are then annihilated column-wise, which produces large fill-in blocks or bulges to be chased down and therefore, may result in substantial extra flops. If eigenvectors are additionally required, the transformations can be efficiently accumulated using Level 3 BLAS operations to generate the orthogonal matrix. It is also noteworthy that the SBR package relies heavily on multithreaded optimized BLAS to achieve parallel performance, which follows the expensive fork-join paradigm.

[1]up to our knowledge.

Davis and Rajamanickam [11] implemented a similar toolbox called PIRO_BAND, which only focusses on the last stage i.e., the reduction from band form to the condensed structures. This software enables us to reduce, not only symmetric band matrices to tridiagonal form but also non-symmetric band matrices to bidiagonal form needed for the SEVP and the singular value decomposition, respectively. This sequential toolbox employs fine-grained computational kernels, since it only operates on regions located around the diagonal structure of the matrix. However, the off-diagonal entries are annihilated element-wise and the number of fill-in elements is drastically reduced compared to the SBR implementation. As a consequence, the overall time to solution has been improved compared to SBR package, even though the PIRO_BAND implementation is purely sequential. Finally, PIRO_BAND relies on pipelined plane rotations (i.e., Givens rotations) to annihilate the off-diagonal entries.

More recently, Luszczek et al. [19] introduced a new parallel high performance implementation of the tile TRD algorithm on homogeneous multicore architectures using a two-stage approach. The first stage uses high compute intensive kernels and reduces the matrix to band tridiagonal form. The second stage follows the SBR principle of annihilating the extra entries column-wise. However, as opposed to the SBR toolbox, they brought the parallelism originally residing in the multithreaded BLAS-based kernels to the fore and exposed it to a dynamic runtime system environment. The various computational tasks then get scheduled in an out-of-order fashion, as long as data dependencies among them are not violated. A left-looking variant has been developed for the bulge chasing to increase data locality and to reduce memory bus traffic. Furthermore, the algorithmic data layout mismatch between both stage algorithms (tile and column-major data layout for the first and the second stage, respectively) has been handled through a data dependency layer (DTL), which provides to the dynamic scheduler crucial information to ensure numerical correctness. Last but not least, the overall performance of their implementation is solely guided by the tile size, which will eventually determine the bandwidth size for the second stage. Therefore, because of the second stage being memory-bound, a tuned tile size must be appropriately chosen in order to extract the best performance out of both stages.

The next Section clearly details the research contributions of the work presented in this paper.

3. RESEARCH CONTRIBUTIONS

The research contributions of this paper can be mainly listed in three points:

- New high performance fine-grained and memory-aware kernels have been implemented for the first and the second stages. The first stage kernels described in Luszczek et al. [19] have been fused to form more compute intensive tasks, while reducing the overhead of the dynamic runtime system. The new kernels employed during the second stage aggregate the left and right applications of Householder reflectors occurring within a single data block, in order to reduce the memory bus traffic and contentions and at the same time, taking full advantage of running at the cache speed. Section 6 provides more information about those new fine-grained and cache-friendly numerical kernels.

- Getting a small matrix bandwidth after the first reduction stage is a key benefit for the second stage i.e., the bulge chasing procedure. In other words, this procedure requires a small tile size, which is not optional for the first stage because it usually relies on coarse-grained compute intensive tasks to achieve high performance. This challenging trade-off has been handled using a *grouping* technique, which allows us to use fine-grained computational kernels during the first stage instead, while sustaining the original high performance of this first phase. A bench of small tiles are combined together to form what we call a *super* tile. The general tile TRD algorithm proceeds then on the super tiles in an oblivious manner, which permits to drastically reduces the overhead of the runtime system as well. This grouping technique has been also used during the second stage to merge the new fine-grained memory-aware kernels in order to further increase data reuse and to significantly reduce the memory bus traffic. This contribution is further described in Section 7.

- During the second stage, the dependence tracking has been performed using function dependencies rather than data dependencies. The former can translate to the latter but the main advantage of the former is that the number of dependencies is flat and equal to two, while many more dependencies need to be monitored by the dynamic runtime system when using a dependence translation layer (DTL) as suggested in Luszczek et al. [19]. Also, the function dependencies framework allows two different kernels to simultaneously run on a single data tile as long as there are no overlapping regions, as opposed to DTL, in which the access to the data tile by a kernel is atomic. Although function dependencies played a major role in getting the performance numbers reported in this paper, and thus perfectly suits the current problem being solved, DTL offers a more systematic and general mechanism to track data dependencies for any problem types. More details about this contribution can be found in Section 8.2.

Our research contributions can be seen as a substantial improvement of the first stage compared to SBR [5] by exposing more parallelism thanks to tile algorithms as well as compared to Luszczek et al. [19] due to the new kernel implementations and the grouping technique. Our second stage has also considerably benefited from the grouping technique and has therefore leveraged the contributions in Luszczek et al. [19], especially by chasing the extra entries one-by-one rather than an entire entry column, which generates less fill-in elements, and thus less additional flops. Last but not least, our second stage can also be seen to some extent as a very efficient parallel implementation of the PIRO_BAND package [11] using Householder reflectors instead and a dynamic runtime system optimized for thread and data locality to schedule the fine-grained memory-aware kernels.

The next Section describes the various drawbacks of the one-stage approach used to perform the standard TRD algorithm.

4. THE BOTTLENECKS OF THE STANDARD TRD ALGORITHM

It is very important to first identify the bottlenecks of the standard one-stage approach used to compute the TRD algorithm, as implemented in the state-of-the-art numerical package, namely LAPACK [4]. LAPACK is characterized by two successive computational steps: the panel factorization and the update of the trailing submatrix. The panel factorization computes the transformations within a specific rectangular region using Level 2 BLAS operations (memory-bound) and accumulates them so that they can be applied into the trailing submatrix using rather Level 3 BLAS operations (compute-bound). The parallelism in LAPACK resides within the BLAS library, which follows the expensive fork and join model. This produces unnecessary synchronization points between the panel factorization and the trailing submatrix update phases. The serious coarse granularity is also a significant drawback and prevents us from getting a higher degree of parallelism.

In particular, the TRD algorithm implemented in LAPACK is no exception. Furthermore, it follows a one-stage approach where the final condensed form is obtained in one shot, after performing successive panel-update sequences. The panel factorization step is actually more critical for two-sided reductions (e.g., TRD) than one-sided factorizations (e.g., LU) because the computations within the panel require loading the entire trailing submatrix into memory. As memory is a very scarce resource, this will obviously not scale for larges matrices, and thus will generate a tremendous amount of cache and TLB misses. Indeed, Figure 1 shows the performance evaluation in Gflop/s of the one-stage LAPACK TRD algorithm linked with optimized Intel MKL BLAS as well as the amount of TLB misses associated with it (determined using the performance counter library PAPI [2]). The machine is a dual-socket quad-core Intel Nehalem 2.93GHz Processors (8 cores total) with 8MB of cache memory and 16GB of main memory. The TRD of small matrices delivers super linear speedups since they are able to fit into cache memory. As the matrix size increases, the number of TLB misses starts to grow exponentially, which makes the asymptotic performance reach only 8% of the theoretical peak of the machine.

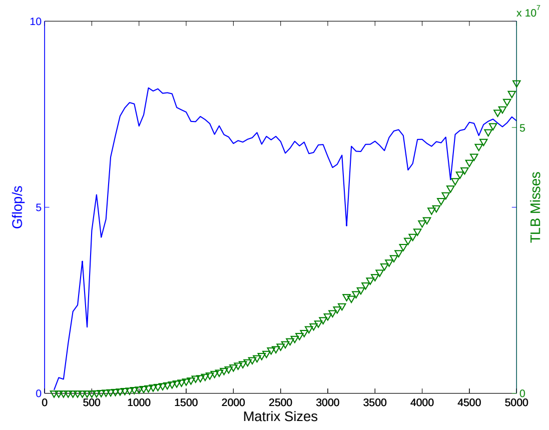


Figure 1: Performance evaluation and TLB miss analysis of the one-stage LAPACK TRD algorithm with optimized Intel MKL BLAS, on a dual-socket quad-core Intel Xeon (8 cores total).

The idea is then to try to overcome those bottlenecks by tackling the problem using a different strategy mostly defined by: a) the tile two-stage approach, b) exposing and bringing the parallelism to the fore, c) fine-grained optimized computational kernels and d) efficient dynamic runtime system for task scheduling. This strategy is explained in the following Sections.

5. THE TILE TWO-STAGE TRD APPROACH

The two-stage approach permits us to cast expensive memory operations occurring during the panel factorization into faster compute intensive ones. This results in splitting the original one-stage approach into a compute-intensive phase (first stage) and a memory-bound phase (second stage). Figure 2 shows how the two-stage approach proceeds. Since the matrix is symmetric, the lower or upper part is not referenced (gray area in Figure 2). The first stage reduces the original symmetric dense matrix to a symmetric band form. The second stage applies the bulge chasing procedure, where all the extra off-diagonal entries are annihilated element-wise.

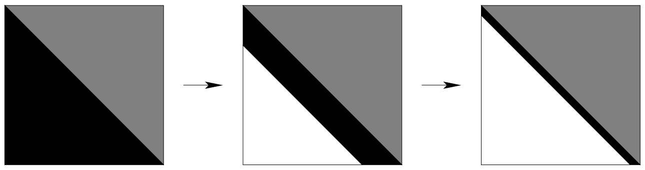


Figure 2: Reduction of a symmetric dense matrix to tridiagonal form using a two-stage approach.

The next milestone is to break those stages into small granularity tasks in order to expose and to bring to the fore the parallelism residing within the BLAS library. The concepts of tile algorithms for one-sided factorizations [3] have been extended to address the two-sided reduction algorithms, and in particular, the two-stage TRD algorithm. The matrix is basically split into square tiles, where each data entry within a tile is contiguous in memory. This requires implementations of new computational kernels to be able to operate on the tiled structure. The general algorithm is then broken into tasks and proceeds on top of tile data layout, which eventually generates a directed acyclic graph (DAG), where nodes represent tasks and edges describe the data dependencies between them. Those kernels will ultimately need to be optimized for each stage they will be running on. Moreover, an efficient dynamic runtime system named QUARK (available within the PLASMA library and described in Section 8.1) is used to schedule the different tasks from both stages in an out-of-order fashion, as long as data dependencies are not violated for numerical correctness purposes. Therefore, the top left matrix corner may have reached the final tridiagonal form (second stage) while the bottom right corner of the matrix is still being reduced to the symmetric band form (first stage).

Last but not least, the degree of parallelism i.e., the number of concurrent tasks, is fairly high for the first stage but very low for the second stage. Indeed, the bulge chasing procedure generates many overlapping regions of computations, which may considerably impede the overall parallel scaling of the tile TRD algorithm. Therefore, the goal is (1) to obtain after the first reduction stage a matrix bandwidth small enough such that, the element-wise bulge chasing operations do not become a bottleneck and (2) to develop fine-grained memory-aware computational kernels to speed up this memory-bound stage through caching improvement mechanisms e.g., by enforcing data and thread locality with the dynamic runtime system support.

The next Section presents the new numerical kernels implemented for the tile two-stage TRD algorithm.

6. HIGH PERFORMANCE FINE-GRAINED AND MEMORY-AWARE KERNELS

This Section gives detailed information about the high performance fine-grained and cache-friendly computational kernels involved in the first and second stage, respectively.

6.1 First Stage: Efficient Fine-grained Kernels

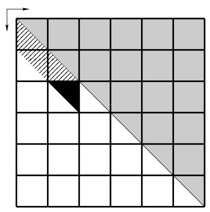
Figure 3 highlights the execution breakdown at the second step of the first stage reduction. Since the matrix is symmetric, only the lower part is referenced and the upper part (gray color) stays untouched. We reuse some of the QR factorization kernels [3] to compute the panel factorizations and to generate the corresponding Householder reflectors. We recall that xGEQRT computes a QR factorization of a single sub-diagonal tile, as presented in Figure 3(a). The left and right applications of a Householder reflector block on a symmetric diagonal tile is done by the new xSYRFB kernel, as shown in Figure 3(b). The right applications on the single tiles proceed then along the same tile column using the xORMQR kernel, as depicted in Figure 3(c). Figure 3(d) shows how xTSQRT computes a QR factorization of a matrix composed by the sub-diagonal tile and a square tile located below it, on the same tile column. Once the Householder reflectors have been calculated, they need to be applied to the trailing submatrix. We developed a new kernel to appropriately handle the symmetric property of the trailing matrix. Indeed, the xTSMQRLR kernel in Figure 3(e) loads three tiles together – two of them are symmetric diagonal tiles – and carefully applies left and right orthogonal transformations on them. This new kernel is a fusion of four distinct kernels previously introduced in Luszczek et al. [19]. Not only does this fused kernel improve the readability of the code, but it also may eventually enhance the overall performance by improving data reuse. Once the special treatment for the symmetric structure has completed, we then apply the same transformations to the left side of the matrix as well as to the right side (down to the bottom of the matrix) using the xTSMQR kernel with the corresponding side variants, as displayed in Figure 3(f) and 3(g), respectively. It is necessary to further explain the left variant, as it should require the tile colored in light gray located in position (3,4) from Figure 3(f). In fact, this tile is not referenced since only the lower part of the matrix is being operated. Therefore, by taking the transpose of the tile located in position (4,3) from the same Figure, we are able to compute the right operations. Finally, Figure 3(h) represents the matrix structure at the end of the second step of the reduction. The symmetric band structure starts to appear at the top left corner of the matrix (the dashed area).

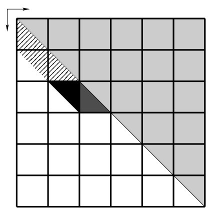
Algorithm 1 describes the different steps of the general tile band TRD algorithm (first stage) for the lower case, using the double precision naming convention for the computational kernels. The reduction to symmetric band tridiagonal form can be easily derived for the upper case. All the operations will be then based on the LQ factorization numerical kernels, as described in Ltaief et al. [18]. Most of the kernels from the first stage are compute-intensive and rely on Level 3 BLAS operations (i.e., matrix-matrix multiplication) to achieve high performance. Therefore, it is critical to supply a large enough tile size to run them close to the theoretical peak performance of the machine.

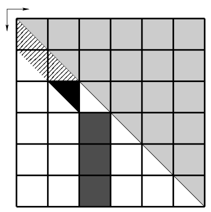
The next Section presents the kernels of the bulge chasing procedure (the second stage) used to annihilate the extra off-diagonal elements.

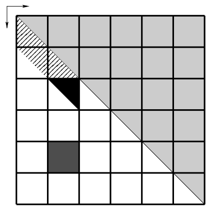
6.2 Second Stage: Cache-Friendly Kernels

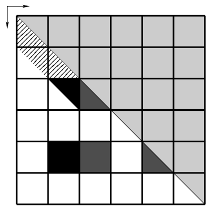
It is very important to understand the shortcomings of the standard bulge chasing procedure before explaining the cache-friendly numerical kernels. Figure 4 shows how the annihilation of the off-diagonal elements proceeds, as implemented in the LAPACK

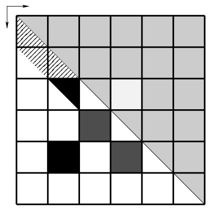

(a) xGEQRT: QR factorization of a single tile.

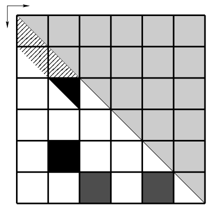

(b) xSYRFB: Left/right updates on a symmetric diagonal tile.


(c) xORMQR: Right updates of a single tile down to the bottom.


(d) xTSQRT: QR factorization of a triangular tile on top a square tile.


(e) xTSMQRLR: Left/right applications on the diagonal region.


(f) xTSMQR: Left application using the transpose of the light gray upper tile.


(g) xTSMQR: Right application down to the bottom.

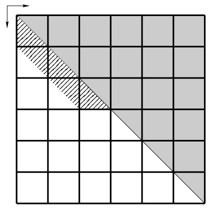

(h) Matrix structure at the end of the reduction step 2.

Figure 3: Kernel execution breakdown of the tile TRD algorithm during the first stage.

Algorithm 1 First stage: reduction to symmetric band tridiagonal form with Householder reflectors.

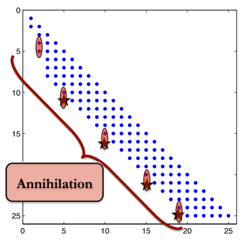
```
 1: for $step = 1, 2$ to $\mathrm{NT}-1$ do
 2:   DGEQRT($A_{step+1,step}$)
 3:   {Left/right updates of a symmetric tile}
 4:   DSYRFB($A_{step+1,step}$, $A_{step+1,step+1}$)
 5:   for $i = step+2$ to NT do
 6:     {Right updates}
 7:     DORMQR($A_{step+1,step}$, $A_{i,step+1}$)
 8:   end for
 9:   for $k = step+2$ to NT do
10:     DTSQRT($A_{step+1,step}$, $A_{k,step}$)
11:     for $j = step+2$ to k-1 do
12:       {Left updates (transposed)}
13:       DTSMQR($A_{j,step+1}$, $A_{k,j}$)
14:     end for
15:     for $m = k+1$ to NT do
16:       {Right updates}
17:       DTSMQR($A_{m,step+1}$, $A_{k,m}$)
18:     end for
19:     {Left/right updates on the diagonal symmetric structure}
20:     DTSMQRLR($A_{step+1,step+1}$, $A_{m,step+1}$, $A_{m,m}$)
21:   end for
22: end for
```

routine xSBTRD, following Schwarz [22] and Kaufman [16] algorithms. Multiple bulges are successively chased down (Figure 4(a)) followed by the corresponding left updates (Figure 4(b)), symmetric updates of the two-by-two diagonal elements (Figure 4(c)) and right updates leading to the creation of bulges (Figure 4(d)). This present algorithm requires the matrix to be accessed from multiple disjoint locations in order to apply the corresponding left and right orthogonal transformations. In other words, there is an accumulation of substantial latency overhead each time the different portions of the matrix are loaded into the cache memory, which is not yet compensated for by the low execution rate of the actual computations (the so-called *surface to volume* effect).

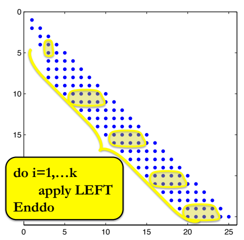
To overcome those critical limitations, we have designed a novel bulge chasing algorithm based on three new kernels, which allow us to considerably enhance the data locality:

- The **xSBELR** kernel: this kernel triggers the beginning of each sweep by successive element-wise annihilations of the extra non-zero entries within a single column, as shown in Figure 5(a). It then applies *all* the left and right updates on the corresponding data block loaded into the cache memory and cautiously handles the symmetric property of the block.
- The **xSBRCE** kernel: this kernel successively applies *all* the right updates coming from the previous kernel, either xSBELR or xSBLRX (described below). This subsequently generates single bulges, which have to be immediately annihilated by appropriate left transformations in order to eventually avoid an expansion of the fill-in structure (Figure 5(b)) by subsequent orthogonal transformations.
- The **xSBLRX** kernel: this kernel successively applies *all* the left updates (coming from the xSBRCE kernel) and the corresponding right updates on a symmetric data block, as depicted in (Figure 5(c)).

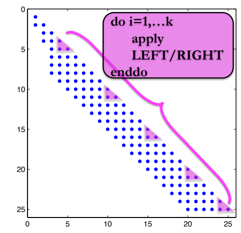
Those kernels share a fundamental principle. The sequences of interleaved left and right updates embedded in each kernel have been decoupled and reordered such that fusing some update operations is possible, as long as the data dependencies are not violated. Moreover, those kernels run on top of tile data layout and perfectly control the various situations, especially in the case where a particular data block spans over several tiles (see Figure 5(d)). Here, the golden rule of thumb is that, once a data block is loaded into the cache memory, *all* possible computational operations have to be applied at once. This prevents going back and forth to request the data from main memory and therefore, permits reuse of the data block already residing within the cache memory. Also, each instance of those functions is given a unique identification number within a sweep to identify the proper kernel to be executed. Let us then introduce T_{me}^{sw} a task T from the sweep *sw* with the identification number *me*. The identifier *me* will also be used as a new way to track the complex data dependencies, as explained later in Section 8.2. The data locality can be further enhanced by interleaving tasks form different sweeps, as seen in Figure 6, producing a *zigzag* pattern. For example, the task T_1^2 can start as soon as the task T_3^1 returns and will actually benefit from the previous tasks still being in the cache memory. Algorithm 2 presents the step-by-step bulge chasing procedure, reducing a symmetric band tridiagonal to condensed form based on element-wise annihilation.



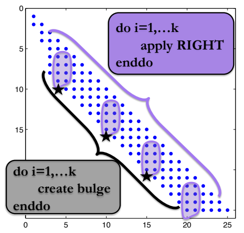
(a) Loop for elimination.



(b) Loop over the left updates.



(c) Loop over the updates on the symmetric diagonal blocks.



(d) Loop over the right updates and the created bulges.

Figure 4: The LAPACK bulge chasing algorithm: xSBTRD (on top of column-major data layout).

As opposed to the first stage kernels, the kernels of the second stage are clearly memory-bound and rely on Level 1 BLAS operations. Their performances will be solely guided by how much data can fit in the actual cache memory. Thus, if the tile size chosen during the first step is too large, the second stage may encounter high difficulties to cope with the memory bus latency. The challenging problem is the following: on the one hand, the tile size needs to be large enough to extract high performance from the first stage and on the other hand, it has to be small enough to extract high performance (thanks to the cache speed up) from the second stage.

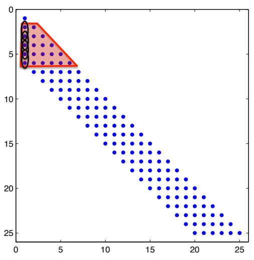
This trade-off between the kernel granularities and performance has been tackled using a grouping technique, which is the subject of the next Section.

7. THE GROUPING TECHNIQUE

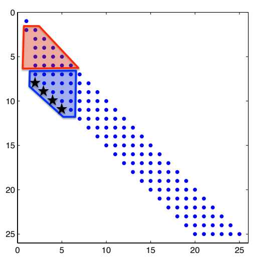
The concept of the grouping technique is straightforward. This technique has been applied to both stages, as presented in the next sections.

7.1 Grouping the Kernels From the First Stage

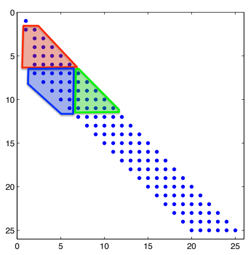
In the first stage, the grouping technique consists of aggregating different data tiles as well as the computational kernels operating on them, in order to build a *super* tile where the data locality may be further enhanced. Figure 7 shows the super tiles of size 2 during the third step of the reduction to symmetric band form. The kernels operating on those super tiles are actually a combination of the kernels previously introduced in Section 6.1. For instance, in Figure 7(a), the single call to the xTSMQRLR-s kernel on the dark gray super tiles will be internally decoupled by multiple calls to the xTSMQR and xTSMQRLR kernels. In the same manner, in Figures 7(b) and 7(c), the single calls to the xTSMQR-s kernel variants (i.e., left transposed and right updates) on the dark gray super tiles will be internally decoupled by multiple calls to the corresponding xTSMQR kernel variants. Therefore, applying the grouping



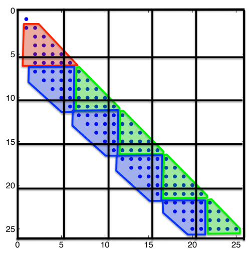
(a) xSBELR.



(b) xSBRCE.



(c) xSBLRX.



(d) Unrolling the kernel calls within one sweep.

Figure 5: First sweep on top of tile data layout of our bulge chasing procedure using the new memory-aware kernels.

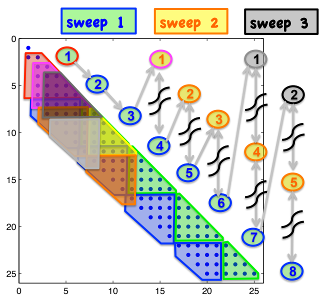


Figure 6: Data dependencies between subsequent sweeps.

technique during the first stage permits us to obtain a small matrix bandwidth size, while increasing at the same time the data reuse in order to compensate the eventual performance losses by running finer-grained kernels.

7.2 Grouping the Kernels From the Second Stage

Following the same principles, Figure 8 describes the grouping technique applied during the second stage to further improve the data locality and the cache reuse. This can actually be considered as a second level of grouping, the first one already being nested within the kernel implementations, by aggregating and fusing the different orthogonal transformations within a single data block (as explained in Section 6.2). The group size then becomes a critical parameter to tune and it is specific for each stage. It is obvious to envision that a small group size will reduce the impact of the data locality and increase the parallelism, while a larger group size will increase the data locality at the expense of decreasing the degree of parallelism. A detailed analysis of the impact of the grouping technique for the first and second stages is presented later in Sections 9.2 and 9.3, with performance comparisons against the equivalent functions in the state-of-the-art numerical libraries.

Algorithm 2 Second stage: reduction from symmetric band tridiagonal to condensed form using the bulge chasing procedure based on element-wise annihilation.

```
1: for $j = 1, 2$ to N$-1$ do
2:   {Loop over the sweeps}
3:   last_sweep = j;
4:   for $m = 1$ to 3 do
5:     for $k = 1, 2$ to last_sweep do
6:       {I am at column $k$, generate my task identifier}
7:       me = (j-k) * 3 + m ;
8:       {Set the pointer (matrix row/col position) for kernels}
9:       p2 = floor((me+1)/2) * NB + k;
10:      p1 = p2 - NB +1;
11:      if (id == 1) then
12:        {the first red task at column $k$}
13:        DSBELR($A_{p1:p2,p1-1:p2}$);
14:      else if (mod(id,2) == 0) then
15:        {a blue task at column $k$}
16:        DSBRCE($A_{p2+1:p2+NB,p1:p2}$);
17:      else
18:        {a green task at column $k$}
19:        DSBLRX($A_{p1:p2,p1:p2}$);
20:      end if
21:    end for
22:  end for
23: end for
```

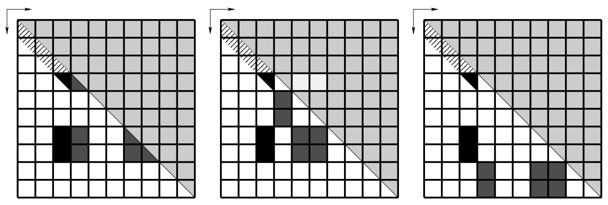

(a) xTSMQRLR-s. (b) xTSMQR-s: left transposed updates. (c) xTSMQR-s: right updates.

Figure 7: Grouping technique during the first stage using super tiles of size 2.

As a consequence, along with the data locality aspect, thread locality is yet another feature which needs to be taken into account to achieve parallel performance, and is described in the next section.

8. PARALLEL IMPLEMENTATION DETAILS

This Section first recalls the dynamic runtime system environment QUARK used to schedule the different kernels from the first and second stage using a thread locality mechanism. It also explains how the new function dependency tracking helps to simplify the actual data dependencies of our tile two-stage TRD (PLASMA-DSYTRD).

8.1 The Dynamic Runtime System Environment QUARK

Restructuring linear algebra algorithms as a sequence of tasks that operate on tiles of data can remove the fork-join bottlenecks seen in block algorithms (as seen in Section 4). This is accomplished by enabling out-of-order execution of tasks, which can hide the work done by the bottleneck (sequential) tasks. In order for the scheduler to be able to determine dependencies between the tasks, it needs to know how each task is using its arguments. Arguments can be `VALUE`, which are copied to the task, or they can be `INPUT`, `OUTPUT`, or `INOUT`, which have the expected meanings. Given the sequential order that the tasks are added to the scheduler, and the way that the arguments are used, we can infer the relationships between the tasks. A task can read a data item that is written by a previous task (read-after-write RAW dependency); or a task can write a data item that is written by previous task (write-after-write WAW dependency); a task can write a data time that is read by a previous task (write-after-read WAR dependency). The dependencies between the tasks form an implicit DAG, however this DAG is never explicitly realized in the scheduler. The structure is maintained in the way that tasks are queued on data items, waiting for the appropriate access to the data. The tasks are inserted into the scheduler, which stores them to be executed when all the dependencies are satisfied. That is, a task is ready to be executed when all parent tasks have completed. Since we are working with tiles of data that should fit in the local caches on each core, we have the ability to hint the cache locality behavior. A parameter in a call can be decorated with the LOCALITY flag in order to tell the scheduler that the data item (parameter) should be kept in cache if possible. After a computational core (worker) executes that task, the scheduler will assign by default any future task using that data item to the same core. Note that the work stealing can disrupt the by-default assignment of tasks to cores.

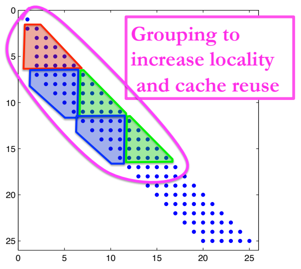


Figure 8: Grouping technique during the second stage.

8.2 Translating Data Dependencies to Functions Dependencies

For the first stage (reduction from dense to band form), the data dependencies are tracked using QUARK, which analyzes the parameter directions (i.e., `VALUE | INPUT | OUTPUT | INOUT`) provided for each task and appropriately determines the order of task scheduling. The second stage (reduction from band to tridiagonal form) is probably one of the main challenging algorithms, when it comes to tracking data dependencies. Indeed, the bulge chasing algorithm proceeds to the next step column-by-column rather than tile-by-tile like in the first stage (Figure 3). As opposed to the first stage, the kernels may thus span over portions of several tiles (one, two or three). Moreover, the subsequent sweeps will generate tasks, which will partially overlap tasks from previous sweeps and this is where the challenge is inherent. To our knowledge, there are no runtime frameworks available which properly handle overlapping region hazards. Luszczek et al. [19] proposed a data dependency layer (DTL) to handle those data dependencies and to provide crucial information to the runtime to achieve the correct scheduling. This layer basically will lock a tile as soon as a single task tries to access it in order to prevent other tasks from acquiring it at the same time, even though the region requested by the latter task may be disjoint from the one operated by the original task. DTL provides a systematic mechanism to handle data dependencies in presence of overlapping regions, it may thus unnecessarily lock data tiles in situations where it should not. Therefore, we have investigated a new tracking technique based rather on function dependencies, which is very suitable for the problem we are trying to solve. Not only does it allow several kernels to access the same data tile but also it drastically simplifies the tracking of the data dependencies and reduce the corresponding analysis overhead of the runtime. To understand how the function dependency technique works, let us recall the previous notation T_{me}^{sw} from Section 6.2, where *me* is the task identifier within the sweep *sw*. Then, we should differentiate two types of dependencies:

- **Dependencies within a sweep:**
  The analysis of the bulge chasing algorithm highlights the fact that the tasks of a single sweep are purely sequential. Thus, within a single sweep j, a task T_{me}^{j} has to return before the task T_{me+1}^{j} can get scheduled. For example, from Figure 5, a green task can not start before the previous blue task finishes.
- **Dependencies between sweeps:**
  Now, when can a task from the subsequent sweeps start? From Figure 6, T_1^2 overlaps portions of T_1^1, T_2^1 and T_3^1 and can run once the latter tasks are finished. Therefore, the task T_{me}^{j} can be scheduled only when the tasks T_{me}^{j-1}, T_{me+1}^{j-1}, T_{me+2}^{j-1} have been completed.

From the different dependencies described above, we can combine them and induce a general definition as follows:

DEFINITION 8.1. *The complete elimination of a column at a sweep "j" of the symmetric band matrix, can be viewed as a set of tasks where each task represents one of the three kernels introduced in Section 6.2. Each task is assigned an identifier "me" following an ascending order starting from "1" at each beginning of a sweep. The data dependencies of a particular task "me" can be translated to function dependencies, such that the task T_{me}^{j} will only depend on* ***two*** *tasks, T_{me-1}^{j} and T_{me+2}^{j-1}.*

Once generated by each task, the **two** function dependencies will be turned over to QUARK, which will consider them in return as simple dependencies and will accordingly schedule the different tasks. Definition 8.1 actually also helps to distinguish potential parallelism opportunities in the second stage. Indeed, interleaving tasks from subsequent sweeps may not only increase the data locality but also the degree of parallelism at the same time. For instance, from Figure 6, the set of tasks T_1^3, T_4^2 and T_7^1 are completely disjoint and can concurrently run, although the impact on the overall performance may be moderate.

However, there exists a discrepancy between the dependency policies of the first stage (data dependencies) and the second stage (function dependencies) and the next section explains how the issue has been resolved.

8.3 Tracking Data Dependencies Between the Two Stages

Since different dependency tracking mechanisms have been used for the first and the second stage, a sort of dependency homogenization needs to be set up in order to be able to run tasks from both stages in an out-of-order fashion without a barrier or a synchronization point between them. From Definition 8.1, it turns out that the only dependency necessary to bridge both stages is to create function dependencies between the tasks of the first sweep "*only*" with the corresponding tasks of the first stage. One can easily derive the following dependency formula using the task identifier "*me*" of the first sweep tasks (from the second stage) to determine the coordinates of the data tile (from the first stage) it is linked to:

```
1: if (I am from the first sweep ($T_{:}^{1}$)) then
2:     if ($mod(me,2) == 0$) then
3:         related_step = me/2
4:     else
5:         related_step = (me+1)/2
6:     end if
7: end if
```

As a result, all the tasks of the first sweep ($T_{:}^{1}$) in the bulge chasing procedure depend on their corresponding tiles $A_{related\_step+1,related\_step}$ from the first stage. This crucial information will be handed to QUARK, which will then ensure tasks form both stages can actually overlap whenever possible.

9. PERFORMANCE RESULTS

This Section presents the performance comparisons of our tile two-stage TRD against the state-of-the-art numerical linear algebra libraries.

9.1 Experimental Environment

9.1.1 Hardware Description

Our experiments have been performed on the largest shared-memory system we could access at the time of writing this paper. It is representative of a vast class of servers and workstations commonly used for computationally intensive workloads. It clearly shows the industry's transition from chips with few cores to tens of cores; from compute nodes with order O(10) cores to O(100) designs, and from Front Side Bus memory interconnect (Intel's NetBurst and Core Architectures) to Non-Uniform Memory Access (NUMA) and cache coherent NUMA hardware (AMD's HyperTransport). It is composed of eight AMD Opteron(tm) Processor 8439 SE of six cores (48 cores total) each running at 2.81 GHz with 128 GB. The total number of cores is evenly spread among two physical boards. The cache size per core is 512 KB. All the computations are done in double precision arithmetics. The theoretical peak for this architecture in double precision is 539.5 Gflop/s (11.2 Gflop/s per core).

9.1.2 Numerical Libraries

There are a number of software packages that implement the tridiagonal reduction. For comparison, we used as many as we were aware of, and here we briefly describe each one in turn. LAPACK [4] is a library of Fortran 77 subroutines for solving the most commonly occurring problems in dense matrix computations. LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. The equivalent routine name performing the tridiagonal reduction is DSYTRD. LAPACK has been linked with the optimized Intel MKL BLAS V10.3.2 to support parallelism. The reference implementation of the SBR Toolbox [5] provides an equivalent routine for the tridiagonal reduction, named DSYRDD. This routine is actually a driver, which automatically tunes the successive reductions. It is also linked with the optimized MKL BLAS V10.3.2 to achieve parallelism. MKL (Math Kernel Library) [1] is a commercial software from Intel that is a highly optimized programming library. It includes a comprehensive set of mathematical routines implemented to run well on multicore processors. In particular, MKL includes a LAPACK-equivalent interface that allows for easy swapping of the LAPACK reference implementation for the MKL by simply changing the linking parameters. The SBR Toolbox interface is also available in MKL, under the name DSYRDB. We recall that the algorithmic complexity of the standard full tridiagonal reduction is $\frac{4}{3}N^3$.

9.2 Tuning the Group Size

One of the key parameters to tune is the group size. It corresponds to a trade-off between the degree of parallelism and the amount of data reuse. Figure 9 shows the effect of the group size for each computational stage for a matrix size N$=$ 12000 using 48 cores. For the first stage, a large group size impedes the task-based parallelism, and thus significantly increases the elapsed time. However, it does not really affect the second stage since this stage does not expose parallelism enough anyway. For this particular matrix size, the optimized super tile size for the first stage is 2 and the optimized group size for the second stage is 8, respectively.

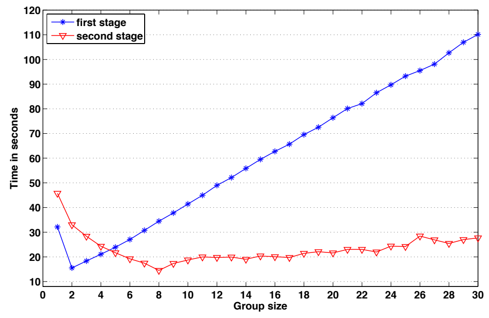


Figure 9: Effect of the group size on both stages for a matrix size N$=$ 12000 **running using eight socket hexa-core AMD Opteron** (48 **cores total).**

9.3 Tuning the Tile Size

The tile size is yet another critical parameter to tune. We present below how different the optimized tile sizes for each stage are.

9.3.1 Performance Comparisons of The First Stage

Table 1 shows timing comparisons in seconds of the first stage (reduction from dense to band) using 48 cores, with and without the task grouping technique, while varying the tile size. The elapsed time dramatically increases as the tile size gets smaller when the grouping technique is disabled. Enabling the grouping technique enhances the cache reuse, minimizes the overhead of the scheduler by tracking less data dependencies and thus, significantly decreases the elapsed time of the first stage. Figure 10 highlights the scalability of the first stage (PLASMA-DSYRBT) when the number of cores increases. We compare our first stage procedure against the equivalent function from SBR toolbox using the optimized tile size reported in Table 1 i.e., NB$=$ 160. Our implementation of the first stage scales reasonably, thanks to the increased parallelism degree brought by tile algorithms. Although this stage is very compute-intensive, the equivalent SBR function to reduce the symmetric dense to band form does not scale at all, surprisingly, mainly due to the overhead of the nested-parallelism within the fork-join paradigm.

First stage	without grouping					with grouping				
	4000	8000	16000	20000	24000	4000	8000	16000	20000	24000
NB=160	1.6	6.9	32	54	88	1.6	6.8	32	55	88
NB=80	2.6	10.1	54	92	125	1.3	5.6	32	56	92
NB=40	14.2	243	780	>2000	>2500	2.0	8.4	46	84	132

Table 1: Impact in seconds of the grouping technique on the first stage using eight socket hexa-core AMD Opteron (48 **cores total).**

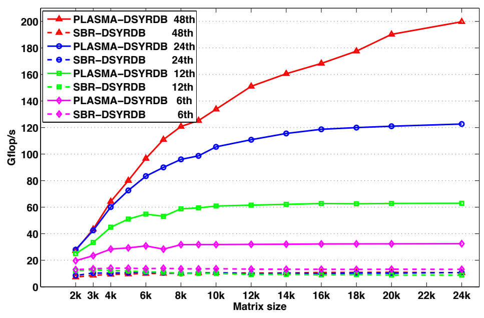


Figure 10: Performance comparisons of the first stage with a matrix bandwidth size NB$=$ 160**.**

9.3.2 Performance Comparisons of The Second Stage

Table 2 presents timing comparisons in seconds of the second stage (reduction from band to tridiagonal form), with and without the grouping technique, using different tile sizes. The elapsed time of the second stage without the grouping technique is roughly divided into 12 when the grouping technique is enabled. However, extracting parallelism from this the bulge chasing is very challenging because of its memory-bound and strong sequential nature. Figure 11 shows the scalability of the second stage (PLASMA-DSBTRD) against the equivalent routines from the SBR toolbox and MKL using 6 and 48 cores. While our implementation shows some limitations in scaling, the SBR and MKL implementations clearly present extremely poor scaling.

Second stage	without grouping					with grouping				
	4000	8000	16000	20000	24000	4000	8000	16000	20000	24000
NB= 160	6.6	15	54	123	161	5.1	11	30	46	61
NB= 80	5.2	20	148	124	214	3.4	7.3	23	35	50
NB= 40	12	69	162	434	357	2.8	7.3	23	36	53

Table 2: Impact in seconds of the grouping technique on the second stage using eight socket hexa-core AMD Opteron (48 **cores total).**

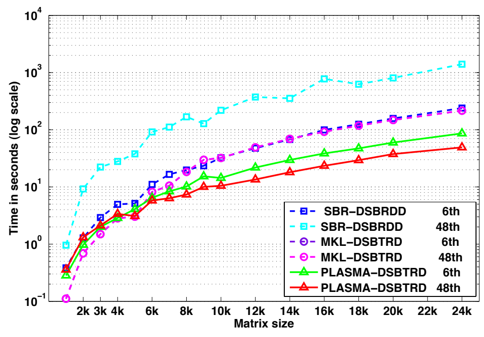


Figure 11: Scalability limitations of the second stage.

9.4 Performance Comparisons of The Overall Tile Two-Stage TRD

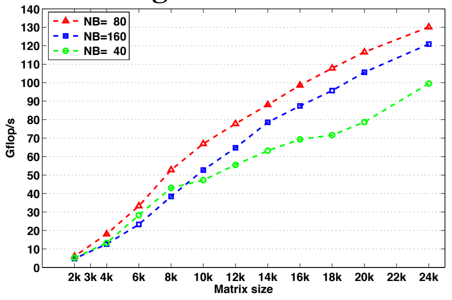


Figure 12: Effect of the tile size on the overall performance of PLASMA-DSYTRD using eight socket hexa-core AMD Opteron (48 cores total).

Figure12 shows the performance of the overall tile two-stage TRD (PLASMA-DSYTRD) when using different NB = 160 | 80 | 40. It turns out that the optimized empirical tile size for the whole reduction is NB = 80, which is similar to the one from the second stage. This confirms the importance of the bulge chasing procedure in the overall algorithm. In other words, close attention to the second stage with an appropriate tuning of the tile size is necessary to extract high performance from the overall algorithm. Figure 13 describes the performance comparisons of our tile two-stage TRD against the equivalent functions, optimized accordingly, from the state-of-the-art numerical libraries. Namely, (1) the DSYTRD routine from LAPACK reference and (2) the DSYRDD function from SBR toolbox, both linked with Intel MKL BLAS, (3) the DSYTRD routine from the vendor Intel MKL library and finally, (4) the Intel MKL implementation of the DSYRDB routine from the reference SBR toolbox. The best grouping sizes for the first and the second stages, as well as the tile size have been empirically selected for each matrix size. For a 24000×24000 matrix size, the group sizes for the two stages are equal to 2 and 8, respectively and the optimized tile size is NB = 80, as mentioned above. Our implementation asymptotically achieves a 2-fold speedup against the Intel MKL implementation of the SBR toolbox routine, using only the new fine-grained and memory-aware kernels (w/o grouping). After enabling the grouping technique, PLASMA-DSYTRD asymptotically scores a 50-fold speedup against the LAPACK reference routine, a 12-fold speedup against the Intel MKL DSYTRD and the SBR reference implementation and a 7-fold speedup against the Intel MKL implementation of the SBR toolbox routine. Calculating the eigenvalues (e.g., with a divide-and-conquer approach) from the tridiagonal form is an $O(N^2)$ complexity and therefore, the performance curves in Figure 13 would slightly drop (up to 5%).

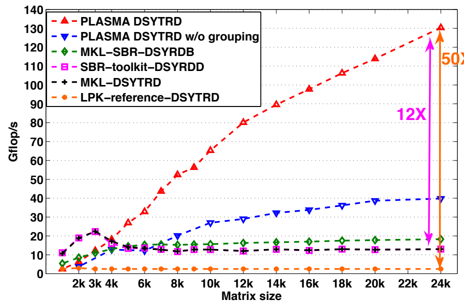


Figure 13: Performance comparisons of PLASMA-DSYTRD against the state-of-the-art numerical libraries using eight socket hexa-core AMD Opteron (48 cores total).

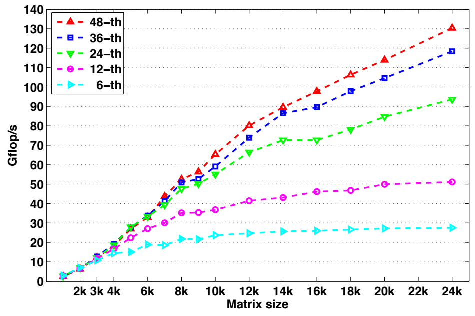


Figure 14: Performance scalability of PLASMA-DSYTRD.

9.5 Scalability

Figure 14 shows the performance scalability of PLASMA-DSYTRD on 48 cores. Our implementation perfectly scales up to 24 cores and the performance then starts to deteriorate. This is mainly due to (1) the scalability limitations of the second stage, as pointed out earlier in Figure 11 and (2) to the NUMA design, especially when the cores located on one physical board initiate data requests to the other remote physical board; and those data requests consistently happen during the second stage.

10. SUMMARY AND FUTURE WORK

This paper describes an efficient implementation of PLASMA-xSYTRD on the latest generation of shared-memory systems we could access. The symmetric dense matrix is reduced using a two-stage approach on top of tile data layout, where the tile matrix is first reduced to symmetric band form prior to the final condensed structure. The authors have developed high performance new fine-grained and memory-aware kernels, tightly coupled with the dynamic runtime environment system QUARK. They have further leveraged the overall performance thanks to the grouping technique as well as the use of function dependencies, which simplifies the complex data dependencies between the different tasks from both stages. The PLASMA-xSYTRD performance reported in this paper is unprecedented. Our implementation results in up to 50-fold improvement compared to the equivalent routine from LAPACK V3.2 and Intel MKL V10.2 on an eight socket hexa-core AMD Opteron multicore shared-memory system (48 cores total) with a matrix size of 24000×24000. It is noteworthy to mention that the overall performance of PLASMA-xSYTRD (130 Gflop/s) represents only a small fraction of the machine theoretical peak, roughly 25%. But considering the memory-bound nature of the second stage, the performance obtained is actually very encouraging and exceeds the expectation for this type of algorithm. The computation of the eigenvectors is the next milestone the authors will be looking at, along with the natural extension to the bidiagonal and Hessenberg reductions, which are the first processing step toward computing the singular value decompositions and the eigenvalues of a non-symmetric matrix, respectively. Last but not least, the authors believe that understanding the challenges involved when implementing such an application on a shared-memory system is critical before tackling the distributed environment, although the last generation of shared-memory machines, like the one used in our experiments, emulates to some extent the distributed context. A future distributed implementation will eventually be integrated within the DPLASMA library [6].

11. REFERENCES

[1] http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[2] Performance Application Programming Interface (PAPI). Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/papi/.

[3] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[4] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[5] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The sbr toolbox—software for successive band reduction. *ACM Trans. Math. Softw.*, 26(4):602–616, 2000.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, J. K. Thomas Herault, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *Accepted at the 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*, Anchorage, AK, USA, May 2011. ACM.

[7] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2006.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20(13):1573–1590, 2008.

[9] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[10] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125, New York, NY, USA, 2007. ACM.

[11] T. A. Davis and S. Rajamanickam. PIRO BAND, Pipelined Plane Rotations for Blocked Band Reduction.

[12] J. Dongarra and P. Beckman. The International Exascale Software Roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.

[13] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215 – 227, 1989. Special Issue on Parallel Algorithms for Numerical Linear Algebra.

[14] G. H. Golub and C. F. Van Loan. *Matrix Computation*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[15] R. G. Grimes and H. D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Trans. Math. Softw.*, 14:241–256, September 1988.

[16] L. Kaufman. Banded eigenvalue solvers on vector machines. *ACM Transaction on Math Software*, 10, No. 1:73–85, Mar. 1984.

[17] B. Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25(7):845–860, 1999.

[18] H. Ltaief, J. Kurzak, and J. Dongarra. Parallel band two-sided matrix bidiagonalization for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), April 2010.

[19] P. Luszczek, H. Ltaief, and J. Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. *IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[20] E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2), July 2008.

[21] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *PDP*, pages 301–310. IEEE Computer Society, 2008.

[22] H. Schwartz. Tridiagonalization of a symmetric band matrix. 12:231–241, 1968. Also in [**?**, pages 273–283].

[23] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[24] University of Tennessee Knoxville. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*, November 2010.

[25] University of Texas Austin. *The FLAME Project*, April 2010.