A Framework for Out of Memory SVD Algorithms

Khairul Kabir[4], Azzam Haidar[1], Stanimire Tomov[1],
Aurelien Bouteiller[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, USA
[2] Oak Ridge National Laboratory, USA
[3] University of Manchester, UK,
[4] Nvidia, USA

Abstract. Many important applications – from big data analytics to information retrieval, gene expression analysis, and numerical weather prediction – require the solution of large dense singular value decompositions (SVD). In many cases the problems are too large to fit into the computer's main memory, and thus require specialized *out-of-core* algorithms that use disk storage. In this paper, we analyze the SVD communications, as related to hierarchical memories, and design a class of algorithms that minimizes them. This class includes out-of-core SVDs but can also be applied between other consecutive levels of the memory hierarchy, e.g., GPU SVD using the CPU memory for large problems. We call these *out-of-memory* (OOM) algorithms. To design OOM SVDs, we first study the communications for both classical one-stage blocked SVD and two-stage tiled SVD. We present the theoretical analysis and strategies to design, as well as implement, these communication avoiding OOM SVD algorithms. We show performance results for multicore architecture that illustrate our theoretical findings and match our performance models.

1 Introduction

The singular value decomposition (SVD) of an $m \times n$ matrix A finds two orthogonal matrices U, V, and a diagonal matrix Σ with non-negative numbers, such that $A = U\Sigma V^T$. The diagonal elements of Σ are called the singular values, and the orthogonal matrices U and V contain the left and right singular vectors of A, respectively. The SVD is typically done by a three-phase process: **1)** *Reduction phase*: orthogonal matrices Q and P are applied on both the left and the right side of A to reduce it to a bidiagonal form matrix, B; **2)** *Solver phase*: a singular value solver computes the singular values Σ, and the left and right singular vectors $\widetilde{U}$ and $\widetilde{V}$, respectively, of the bidiagonal matrix B; **3)** *Singular vectors update phase*: if required, the left and the right singular vectors of A are computed as $U = Q^T\widetilde{U}$ and $V = P\,\widetilde{V}$. In this work, we are interested in the computation of the singular values only. When the matrix A is too large and does not fit in-memory, our goal is to design efficient algorithms to perform the computation while A is out-of-memory (e.g., A could be in the hard disk drive, flash memory, or fast buffer when a CPU computation is considered, or in the CPU memory for GPU or Xeon Phi computations). The memory bottleneck of the SVD computation is the first phase (e.g., illustrated by the difference in columns 7 and 8 in Tables 4 and 5 from the experimental results section). Once A is reduced, B consists of two vectors that fit

(in general) in-memory, where the singular value solver will be able to compute the singular values of B in-memory. If the singular vectors are needed, the second phase also requires the use of OOM techniques. To reduce a general matrix to bidiagonal form we can use either the standard approach which is implemented in LAPACK (we call it one-stage algorithm since it reduces the matrix to bidiagonal in one step), or a two-stage algorithm which reduces the matrix to a bidiagonal form in two steps: first to a band, and then to the bidiagonal form.

Since A resides out-of-memory, the communications to bring parts of A in-memory and back will have a high impact on the overall run time of any OOM algorithm. Thus, to develop efficient OOM SVDs, first and foremost we must study the SVD computational processes and communication patterns, in order to successfully design next the algorithms that minimize communications, as well as overlap them with computation as much as possible.

2 Related Work

A number of dense linear algebra algorithms have been designed to solve problems that are too large to fit in the main memory of a computer at once, and are therefore stored on disks [18,6,4]. Called *out-of-core*, these algorithms mainly targeted one-sided factorizations (LU, QR, and Cholesky). Similar algorithms can be derived between other levels of the memory hierarchy, e.g., for problems that use GPUs but can not fit in the GPU's memory and therefore also use CPU memory, e.g., called non-GPU-resident in [19,20].

Similar algorithms are computationally not feasible for the standard eigensolvers or SVD problems in LAPACK, as we show in this paper, and therefore have not been developed before. Exceptions are special cases, e.g., SVD on tall-and skinny matrices, where a direct SVD computation is replaced by an out-of-core QR first, followed by an in-core SVD of the resulting small R [17].

The development of two-stage eigensolvers and SVD algorithms made it feasible to consider designing their out-of-core counterparts. A two-step reduction for the generalized symmetric eigenvalue problem was reported for the first time in the context of an out-of-core solver [8,9]. Later, the two-stage approach [14,2] was generalized to a multi-stage implementation [3] to reduce a matrix to tridiagonal, bidiagonal, and Hessenberg forms. The two-stage approach was applied to the TRD (Triangular Reduction) [12] and to SVD [13,16,15] in combination with tile algorithms and runtime scheduling based on data dependences between tasks that operate on the tiles. This resulted in very good performance but has never been used to compute the singular vectors.

We note that the principle of the two-stage approach is similar to [12], as in both cases the matrix is first reduced to condensed forms. However, the final form and the transformations used are different, e.g., [12] is for symmetric eigenvalue problems, so reduction is to block diagonal and symmetry is preserved, while the reduction for SVD is to band with shape as shown in Figure 2. The second stages are also different in terms of transformations, their application, and final matrix shape (tridiagonal vs. bidiagonal). Figure 7 shows the effect of specific strategies for retaining data in memory vs. a generic approach, e.g., that would follow the computation as coded in [12].

More recently, a new parallel, high-performance implementation of the tile reduction phase on homogeneous multicore architectures was introduced [15]. It used a two-

stage approach and a runtime scheduler that keeps track of data dependences. Algorithmically, the two-stage approach is the latest development in the field.

3 Contributions

The primary goal of this paper is to design communication avoiding OOM SVD algorithms and their efficient implementations that overlap communications with computations as much as possible. An efficient (and acceptable) OOM SVD design must perform the computation in a realistic time and hide the communication overhead to the fullest. Our main contributions towards achieving this goal are as follows:

- We developed and presented the analysis of the communication costs for the one-stage and two-stage SVD algorithms on hierarchical memories, e.g., CPU memory for main memory and disk for out-of-memory storage, or the GPU/Coprocessor memory for main memory and CPU DRAM memory for out-of-memory storage;
- We investigated different communication avoiding strategies and developed a design with optimal communication pattern;
- We created techniques, along with their theoretical analysis, to hide communication overheads for OOM SVD;
- We also designed a communication avoiding OOM SVD algorithm and developed an optimized implementation for multicore architecture. We showed performance results that illustrate its efficiency and high performance.

4 Background

The first phase of the SVD computation is called *bidiagonal reduction* or BRD, and as mentioned, is considered to be the most expensive part of the computation. In particular, when only singular values are to be computed, it takes more than 90% of the time on modern computer architectures. The BRD's computation cost in terms of floating point operations (flops) is $O(\frac{8}{3}n^3)$. The two main approaches for the BRD phase are:

- **One-stage approach**: the standard one-stage approach as implemented in LAPACK [1], applies Householder transformations in a blocked fashion to reduce the dense matrix to bidiagonal form in one step;
- **Two-stage approach**: the newly developed two-stage approach [12] reduces the general matrix to band form in a first stage, and then reduces the band matrix to bidiagonal form in a second stage.

4.1 The one-stage algorithm for SVD

The one-stage reduction of a matrix A to bidiagonal form, as is implemented in LAPACK, applies orthogonal transformation matrices on the left and right side of A. The transformations are applied from both left- and right-side of A, and therefore BRD is also called a "two-sided factorization." The blocked BRD [5] proceeds by "panel/trailing matrix update" and can be summarized as follows. The panel factorization zeroes the

entries below the subdiagonal and above the diagonal. It goes over its "*nb*" columns and rows (red portion of Figure 1) and annihilates them one after another in an alternating fashion (a column followed by a row, as shown in Figure 1). The panel computation requires two matrix-vector multiplications: one with the trailing matrix to the right of the column that is being annihilated, and a second one with the trailing matrix below the row that is being annihilated. The panel computation generates the left and right reflectors U and V, and the left and right accumulation X and Y. Once the panel is done, the trailing matrix is updated by two matrix-matrix multiplications:

$$A_{s+nb:n,s+nb:n} \leftarrow A_{s+nb:n,s+nb:n} - U \times Y^{\mathsf{T}} - X \times V^{\mathsf{T}},$$

where s denotes the step and nb denotes the panel width. The process is repeated until the whole matrix is reduced to bidiagonal form.

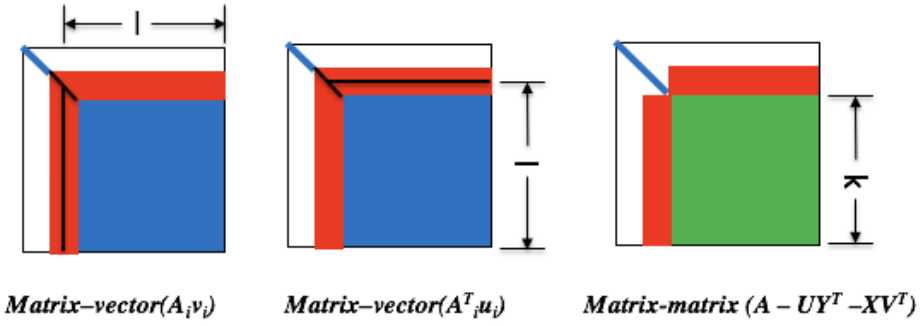


Fig. 1. LAPACK one-stage blocked algorithm: illustration of the main BLAS kernels used.

4.2 The two-stage Algorithm for SVD

Because of the cost of the reduction step, renewed research has focused on improving this step, resulting in a novel technique based on a two-stage reduction [3,5,7,10,12]. The two-stage reduction is designed to overcome the limitations of the one-stage approach by exchanging memory-bound operations for compute intensive ones. It relies heavily on compute-intensive operations so that performance scales up with CPU core count. As the name implied, the two-stage approach splits the original one-stage approach into two phases - the first stage reduces the general matrix to band form and the second stage reduces the band matrix to bidiagonal form. The first stage is compute-intensive and heavily depends on Level 3 BLAS, whereas the second stage which represents a small percentage of the flops and is memory bound, but can be implemented with cache-friendly and memory-aware kernels to make it efficient.

First Stage: Compute-Intensive The first stage reduces general matrix to band form using a sequence of blocked Householder transformations. Compared with the one-stage algorithm, this stage eliminates matrix-vector operations and replaces them with

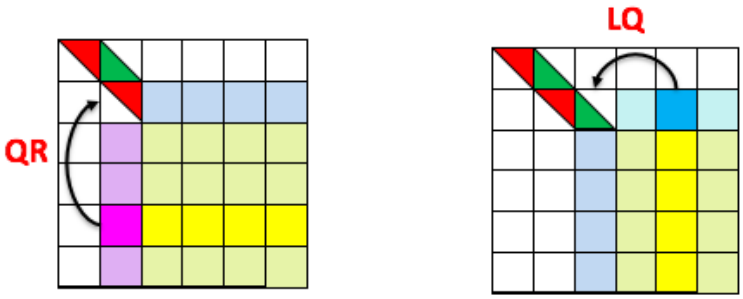
matrix-matrix multiply kernels. Conceptually the matrix of size $n \times n$ is split into $u \times u$ tiles of size nb each, where $u = n/nb$. The algorithm then proceeds as a collection of interdependent tasks that can be scheduled for execution by either static or dynamic scheduler. Algorithm 1 shows the tile algorithm for the reduction of general matrix to band form. Figure 2 shows the execution foot-print for the second step of the reduc-

```
for s = 1 to u do
    GEQRT(A(s,s));
    for j = s+1 to u do
        UNMQR(A(s,s), A(s,j));
    for k = s+1 to u do
        TSQRT(A(s,s), A(k,s));
        for j = s+1 to u do
            TSMQR(A(s,j), A(k,j), A(k,s));
    if (s < u) then
        GELQT(A(s,s+1));
        for j = s+1 to u do
            UNMLQ(A(s,s+1), A(j,s+1));
        for k = s+2 to u do
            TSLQT(A(s,s+1), A(s,k));
            for j = s+1 to u do
                TSMLQ(A(j,s+1), A(j,k), A(s,k));
```

Algorithm 1: Two-stage algorithm to reduce a general matrix to band form.

tion to band (stage 1) algorithm. The process consists of a QR sweep followed by an LQ sweep at each step. A QR factorization (GEQRT) is computed for the tile $A_{2,2}$ (the red tile). When this QR factorization is finished, all the tiles to right of $A_{2,2}$ (the light blue tiles of Figure 2a) are updated by the UNMQR function (each tile is updated by multiplying it on the left by Q^T). At the same time, all the tiles $A_{\bullet,2}$ (the magenta tiles of Figure 2a) can also be factorized using the TSQRT kernel (computing the QR factorization of a matrix built by coupling the R factor of the QR of $A_{2,2}$ and the $A_{\bullet,2}$ tiles) one after another as all of them modify the upper triangular portion of $A_{2,2}$. Once the factorization of any of the tiles $A_{i,2}$ (for example the dark magenta tile of Figure 2a), is finished, all the tiles of the block row i (the dark yellow tiles of Figure 2a) are updated by the TSMQR kernel. Moreover, when all the operation on tile $A_{2,3}$ are finished, LQ factorization (GELQT) can now proceed for this tile (the green tile of Figure 2b). Just like the QR process, all the tiles in the third column $A_{3:u,3}$ (the light blue tiles of Figure 2b) are now updated by the Householder vectors computed during the LQ factorization (UNMLQ). Note, however, that this last update has to wait until the prior QR operations have completed. Similarly, all the tiles $A_{2,4:u}$ (the blue tiles of Figure 2b) can also be factorized (TSLQT), and once any of the tiles $A_{2,i}$ (for example, the dark blue tile of Figure 2b) finish it factorization, it enables the update of the tiles in the block

column *i* (the dark yellow tiles of Figure 2b) using the TSMLQ kernel. The interleaving of QR and LQ factorization at each step, as explained above, repeats until the end of the algorithm. At the end, it generates a band matrix of band size *nb*. Such restructuring of the algorithm removes the fork-join bottleneck of LAPACK and increases the overall performance efficiency.



(a) QR factorization of tile $A_{2,2}$ (b) LQ factorization of tile $A_{2,3}$

Fig. 2. Kernel execution of the BRD algorithm during the first stage.

Second Stage: Cache-Friendly Computational Kernels The band form is further reduced to the final condensed form using the bulge chasing technique. This procedure annihilates the extra off-diagonal elements by chasing the created fill-in elements down to the bottom right side of the matrix using successive orthogonal transformations. Since the band "*nb*" is supposed to be small, we consider that the data of this phase fit into the main memory and thus an in-memory algorithm can be applied. This stage involves memory-bound operations that require irregular memory accesses throughout the band matrix. In other words, a straightforward implementation will start accumulating substantial latency overheads each time different portions of the matrix are loaded into cache memory, and the loads can not be compensated for by the low execution rate of the actual computations on that data. To overcome these critical limitations, we employ a bulge chasing technique – originally designed for symmetric eigenvalue problems (tridiagonal reductions) [12] – to extensively use cache friendly kernels combined with fine-grained, memory-aware tasks in an out-of-order scheduling technique, which considerably enhances data locality. This reduction has been designed for newest architectures, and results have shown its efficiency. It has been well optimized so that it takes between 5% to 10% of the global time of the reduction from dense to tridiagonal on modern multicore architectures. We refer the reader to [10,12] for a detailed description of the technique.

5 An analytical study of the communication cost of data movement

In this section we develop and present the communication minimization pattern for the OOM reduction to bidiagonal form. We provide the analysis for the two techniques – one-stage v.s. two-stage – that we use to design OOM SVDs minimizing the communication cost.

As described in section-4.1, the one-stage bidiagonal reduction needs two matrix-vector multiplications (GEMV) with the trailing matrix at every column and row annihilation, and two matrix-matrix multiplications (GEMM) after every panel computation. Thus, when the matrix is large and does not fit into the main memory, it must be loaded from out-of-memory once for each column and once for each row annihilation (e.g., to perform the two GEMV operations) as well as loaded and stored once after each nb columns/rows annihilation (e.g., after each panel) for the two GEMM operations. The algorithm requires $2(m \times nb + n \times nb)$ in-memory workspace to hold the panel (U and V) and the arrays X and Y of Equation (4.1). Therefore, for an $m \times n$ matrix, the amount of words to be read and written (i.e., the amount of data movement) is given by the following formula:

$$\begin{aligned}&\text{Read } A \text{ for dgemv \#1} + \text{Read } A \text{ for dgemv \#2} + \text{Read/Write } A \text{ for dgemm}\\ &= \sum_{s=0}^{n-1}(m-s)(n-s) + \sum_{s=0}^{n-1}(m-s)(n-s-1) + 2\sum_{s=1}^{n/nb}(m-s\times nb)(n-s\times nb).\end{aligned}$$

Thus, for an $n \times n$ matrix, the amount of word movements is about $\approx \frac{2}{3}n^3 + \frac{1}{nb} \times \frac{2}{3}n^3$.

On the other hand, for the two-stage approach, there is no notion of panel and trailing matrix update. We also note that, since the whole band matrix of size $\min(m,n)\times nb$ is considered to fit into the main memory, the second stage runs efficiently in-memory, and the main attention must be brought to the first stage (i.e., reduction from dense to band), which needs to be performed in on OOM fashion. Overall, if we follow the description in Section 4.2, we find that a tile that needs to be updated must be "loaded from / stored to" out-of-memory (e.g., disk) once every step. As a result, for an $m \times n$ matrix and band of width nb, the amount of data movement is given by:

$$\begin{aligned}&\text{Read/Write } A \text{ for QR} + \text{Read/Write } A \text{ for LQ}\\ &= 2\times\sum_{s=0}^{n/nb-1}(m-s\times nb)(n-s\times nb) + 2\times\sum_{s=0}^{n/nb-1}(m-s\times nb)[n-(s+1)\times nb]\\ &\approx \frac{2}{nb}(mn^2 - \frac{n^3}{3}).\end{aligned}$$

Thus, for an $n \times n$ matrix, the amount of word movements is about $\frac{4n^3}{3nb}$.

From this formulation, one can easily observe that the classical one-stage algorithm for the reduction to bidiagonal requires $O(n^3)$ more word transfers between the in-memory and the out-of-memory storage than the two-stage approach. This is a huge amount of extra communication that dramatically affects the performance. To highlight the importance of the communications, we start by giving an example. Consider a matrix of size

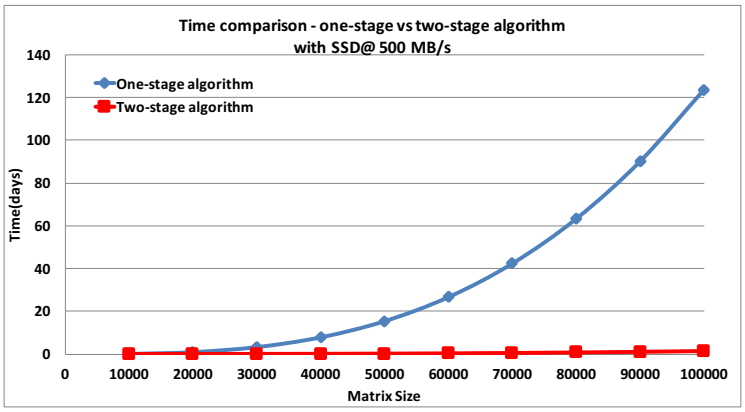


Fig. 3. OOM SVD time comparison between the one-stage and two-stage algorithms.

$n = 100,000$. The classical one-stage algorithm needs $\frac{2}{3}n^3 + \frac{1}{nb} \times \frac{2}{3}n^3$ word movements. In double precision arithmetic, for a recent Hard Drive, Solid State Drives (SSD), or out-of-GPU memory where the communication bandwidths are about 150 MB/s, 500 MB/s, and 8 GB/s, respectively, the standard one-stage technique requires 411, 123, and 7.72 days, respectively, to perform the reduction. The two-stage technique needs approximatively $\frac{1}{nb} \times \frac{4}{3}n^3$ word movements, and thus, in double precision, it necessitates 5.14, 1.54, and 0.09 days, respectively (with *nb* equal 160). Figure-3 compares the times required to reduce a general matrix to bidiagonal form, using either the one-stage or the two-stage algorithm for different matrix sizes when the matrix resides in SSD. In conclusion, these results illustrate that it is unacceptable to build an OOM algorithm based on the one-stage approach. For that, it has long been thought that an OOM SVD implementation is practically impossible.

To further emphasize the choice of the two-stage approach, consider one more example where the matrix fits in-memory, and therefore word movements are between the main memory and the cache levels. For a recent hardware, like the Intel Haswell E5-2650 v3 multicore system, achieving a bandwidth of about 60 GB/s, the one-stage takes about 24.71 hours to finish the reduction to bidiagonal form in double precision arithmetic, while the two-stage algorithm takes about 0.31 hours (with *nb* equal 160).

6 A theoretical study of the design of an OOM SVD solver

In this section, we present the theoretical analysis of the OOM algorithm and provide a detailed study of the communication pattern required by the OOM algorithm. Also, we investigate different strategies to design one that is provably optimal in term of data movement and performance. Using the conclusion from the previous section, the design path for an efficient OOM SVD must follow the two-stage approach. The reduction from dense to band form is thus the main component that needs to be studied and implemented as an OOM algorithm. The algorithm starts with A stored out-of-memory, loads in-memory parts of A by block, performs computation on the in-memory data, and sends back results in order to allow other blocks to be loaded as the algorithm proceeds to completion. For simplicity, we use the specifics for out-of-core algorithms, where the

matrix is on disk and the CPU DRAM is considered to be the main memory. However, the formulation and theorem proved here are general and applicable to OOM SVDs designs targeting other levels of the memory hierarchy.

Besides algorithmic designs to reduce communication, we create techniques to overlap the remaining communications with computation (when possible). We show that this is not always possible for SVD (and eigenvaue solvers) because of the two-sided process that must modify the whole trailing matrix in order to proceed from column to column of the (panel) reduction. We note that this is in contrast to linear solvers that use either Cholesky, LU, or QR factorizations, which do not need the trailing matrix when factorizing a panel [18,6,4,19].

6.1 A study of the ratio of communication to computation

While the main goal in designing efficient OOM algorithms is to minimize communication, as determined in Section 5, the second major objective is to overlap the remaining communications with computation (when possible). Ideally, communication is totally overlapped, in which case the OOM algorithm runs as fast as its in-core counterpart. We study and formulate a theorem that theoretically answers the question to what degree this overlap is possible for an OOM SVD. The basic principle that we apply to hide the communication overhead is: if a computation is using and operating on data of block k, we write back the data of block $k-1$ and read the data of block $k+1$. For full overlap, the communication must be in less or equal time to the computation task on the data of block k. The main and the most time consuming type of tasks in the two-stage algorithm are the update tasks (e.g., TSMQR and TSMLQ) [11]. Therefore, we focus our analysis and description on this type, as the substitution to other type can be derived easily. Figure 4 shows two scenarios for the update tasks: **(1)** All the threads are participating on the computation of a single task that we call *multi-threaded single task*. To hide communication we must write back the tile computed previously (pink color) and bring the next tile (cyan color) in memory in less time than the computation of the current tile (red color); and **(2)** Each thread works on a separate tile – called *sequential multi-task*. If there are p threads, we must write back the previously computed "p" tiles and load the next "p" tiles for the next computation while computation is happening on the current p tiles.

Theorem 1. *The OOM two-stage SVD reduction algorithm fully overlaps data communication with computation if the tile size b is at least $\frac{3.2\alpha}{\beta}$ for double precision (DP) arithmetic, where β is the communication bandwidth (in Bytes/s) and α is the computational performance capability of the system (in flops/s).*

Proof. First, we consider the case when all threads are working on a single task, as shown in Figure 4 (left). A tile of size b consists of b^2 elements, which is $8b^2$ bytes in DP arithmetic. We use the DP arithmetic representation for all the subsequent formulations. Assuming that the read and write bandwidths are similar, the time t_{read} to read (or the time t_{write} to write) a tile of size b in seconds (s) is given by:

$$t_{read} = t_{write} = \frac{8b^2}{\beta},$$

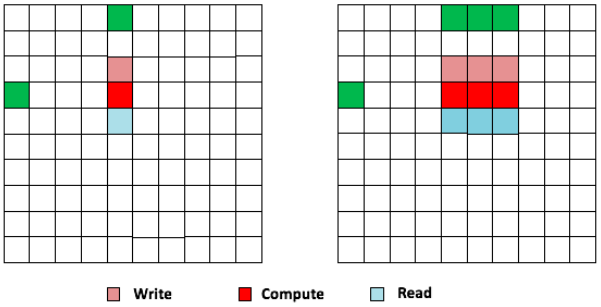


Fig. 4. Reduction of general matrix to band form – update (multi-threaded single task vs. sequential multi-task).

where β is the bandwidth of the transfer between disk and memory. The computation cost of the update task (*TSMQR* or *TSMLQ* routine) for a tile of size b is $5b^3$ flops, yielding $t_{compute} = \frac{5b^3}{\alpha}$, where α is the performance capability in flops/s for the in-memory operation that must be performed (e.g., update operation; we note that TSMQR/TSMLQ reach about 80%-85% of the machine peak). The necessary condition to hide the communication overhead is:

$$\begin{aligned} & t_{compute} \geq t_{read} + t_{write} \\ \Rightarrow & \frac{5b^3}{\alpha} \geq \frac{16b^2}{\beta} \\ \Rightarrow & b \geq \frac{3.2\alpha}{\beta}. \end{aligned}$$

Now consider the case where tasks are running in parallel (see Figure 4 (right)) and each thread is working on a separate tile. If p tasks run in parallel, p tiles are brought in-memory and sent back to disk after the computation. Thus,

$$t_{read} = t_{write} = \frac{p \times 8b^2}{\beta} \quad \text{and} \quad t_{compute} = \frac{5b^3}{\frac{\alpha}{p}} = \frac{p \times 5b^3}{\alpha}.$$

Thus, to overlap communication with the computation, we must have $b \geq \frac{3.2\alpha}{\beta}$ □

Table 1 shows the minimum tile sizes "b", required to completely overlap the communication with the computation for various systems. The higher the performance capability is, the larger the required tile size is in order to overcome the communication time. For example, a Sandy Bridge machine, having a computational performance of $\alpha = 250$ Gflop/s, connected to an HDD with bandwidth of 50 MB/s, requires the tile to be of size $16,000$. Such big tile size is however not computationally feasible for the following reason:

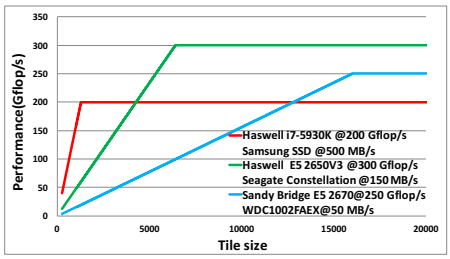
– The tile size defines the width of the reduced band matrix. Band matrix of this size ($n \times b$), may not fit in-memory for the second stage;

System	Communication bandwidth β (GB/s)	Update kernel performance (Gflop/s)	Minimum tile size to hide communication
Sandy Bridge E5-2670 WDC1002FAEX	0.05	250	16000
Haswell i7-5930K Samsung SSD EVO	0.5	200	1280
Haswell E5 2650V3 Seagate Constellation	0.15	300	6400
K40 PCI	8	960	384
P100 PCI	8	3760	1504
KNC PCI	8	768	308
KNL PCI	8	1600	640

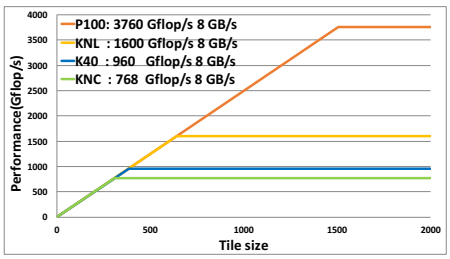
Table 1. Minimum tile sizes needed in order to overlap communication time by computation for OOM SVD solver on various systems.

– Even if the band matrix fits in-memory, the second stage of the algorithm (reduction from band to bidiagonal form) will be extremely inefficient and will adversely affect the overall run-time.

The performance of the two-stage OOM SVD can be estimated by a roofline model for the update tasks. For double precision data, the update task computes $5b^3$ flops for a tile of size b, and communicates $16b^2$ bytes of data. In short update task computes $5b^3$ flop for $16b^2$ byte data. The arithmetic intensity, i.e. flop to byte ratio for update task is $\frac{5b}{16}$. If the system has bandwidth β, performance of two-stage OOM SVD is computed multiplying arithmetic intensity by system bandwidth, i.e. $\frac{5b\times\beta}{16}$. Figure 5 shows the roofline performance model of the OOM SVD solver for different tile sizes and for different types of systems. Figure 5 shows that the peak performance is not achievable with small tile size. At the same time, big tile sizes, that are required to reach peak performance, are also not feasible. Therefore, it can be concluded that the performance of the OOM two-stage algorithm is bounded by disk bandwidth.



(a) The CPU is considered as main memory and the data resides in disk



(b) The Device (GPU/Xeon-Phi) is considered as main memory and the data resides in system DRAM memory

Fig. 5. Roofline performance model for the OOM SVD solver.

7 Algorithmic design

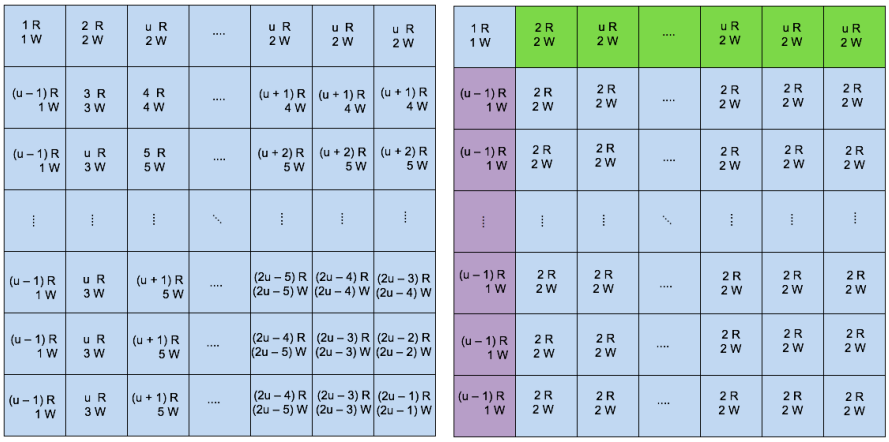
Sections 5 and 6 addressed the two main design considerations for OOM SVD. These are: **(1)** algorithms to minimize the communications between the in- and out-memory layers, and **(2)** overlapping the remaining communications with computation, respectively. In this section, since any system will have some available main physical memory, we analyze and develop strategies to further reduce the communication overhead for the two-stage OOM SVD algorithm by taking advantage of such memory holding data and reusing it as much as possible.

7.1 Proposition 1 - Global Communication Reducing Strategy

In order to minimize the communication overhead, our first strategy follows the idea to hold and keep in memory the tiles that are the most accessed during the whole reduction process which we call global access pattern. As we are reading and writing data in tile granularity, our first algorithmic design comprised of finding out the tiles that are used the most in order keep them in the main memory. If one tile of the matrix is held in memory, then, at each step of Algorithm 1, we can save one read and one write for the *QR* sweep, and similar for the *LQ* sweep till the step reach the tile index. For example if we hold the tile in the lower right corner, the amount of reads and writes that can be reduced by holding it in memory is $2(u-1)R+2(u-1)W$. The most used tiles are in the lower right corner of the matrix as those tiles are used for both the *QR* and *LQ* sweeps in each step of the algorithm. Figure 6a shows the total number of reads (R) and writes (W) required during the process for each tile of a square matrix of $u \times u$ tiles. As a results, according to the proposed strategy, the available physical memory will be used to hold tiles from the lower right corner of the matrix based on their global access number of R/W. Our strategy is implemented as a decision maker engine which decide which tile is to keep in memory, when to release it back (write it back) as well as when to read it. Based on the decision a task with the corresponding dependencies is submitted to the runtime system and this task return the pointer to the data (that has been held, copied) that the next computational kernel will need. Similarly, when a computational task is done, the decision maker decide whether to keep it in memory or to initiate a task that send it back to the disk.

7.2 Proposition 2 - Optimal Communication Reducing Strategy

We analyzed in detail the characteristic of the reduction algorithm. As mentioned above, it is composed of a *QR* followed by an *LQ* sweep or vice versa. The *QR* and the *LQ* sweeps consist of applying the Householder reflectors generated during the panel factorization at each step "*i*" to the trailing matrix. The *QR* and the *LQ* panels of a step "*i*" consists of the tiles in position A(:,*i*) and A(*i*,:) respectively, for example, the tiles highlighted in purple and green in Figure 6b corresponds to the panels of step 1). The trailing matrix is the portion on the right/bottom side of the panel for the *QR* and *LQ* update respectively). Diving into the detail of the algorithm, we can find that the Householder reflectors generated at step "*i*" are needed as input data by all the update tasks corresponding to step "*i*". Thus, these tiles are read as many times as they are needed.



(a) Total number of R/W (b) number of R/W for 1 step

Fig. 6. Snapshot of the amount of reads and writes required overall (left) and by step (right).

Moreover, we can also observe that these tiles are never accessed in the upcomings steps $> i$. For example, the tiles highlighted in purple and green in Figure 6b are read $(u-1)$ and u times for the *QR* and *LQ* sweep, respectively. This is done only in step 1 and they are never referenced after that. As consequence of our analysis, holding one tile from the purple block during the *QR* sweep, then, when the *QR* is done, using the same space to hold one tile form the green area (*LQ* sweep), saves about $2(u-2)$ reads in step 1, $2(u-3)$ reads in step 2, and so on. As a result, if we have physical memory to hold one tile, we can reduce (u^2-3u+2) reads using our proposition 2 compared to $2(u-1)R+2(u-1)W$ using proposition 1. Compared to the first strategy, we can expect a very large gain using this strategy.

Thus, if we consider that the minimum workspace of the OOM algorithm is composed of one panel (e.g., u tiles), we can find that the gain is about $\sum_{s=1}^{u-1}(u-s-1)(u-s)+\sum_{s=1}^{u-1}(u-s-1)(u-s-1)=\frac{2}{3}u^3-\frac{5}{2}u^2+\frac{17}{6}u-1$ reads. Since in practice the available space can be larger than a panel, then after holding the panel, we can start holding from the right bottom corner, since then these tiles become the most used step-wise or global-wise. Figure 7 compares the amount of the read that can be reduced by our two strategies. It is easy to notice that the optimal solution is the second strategy.

8 Runtime Estimation Model for the OOM two-stage SVD solver

In this section we define the model to estimate the run-time for reduction to band form of the OOM two-stage SVD solver. We first start by assuming that there is enough memory to store only u tiles for the panel and 4 tiles for temporary data. Since, as described in the above section, the algorithm is bound by the amount of tiles to be read/written, we compute the total number of tiles to be read/written. For a matrix of $u \times v$ tiles, we compute that the total number of tiles to be read or written is:

$$1 \times \sum_{s=0}^{v-1}(u-s)+2 \times \sum_{s=0}^{v-1}(u-s)(v-s-1) = uv^2-\frac{1}{3}v^3+\frac{1}{2}v^2-\frac{1}{6}v. \quad (1)$$

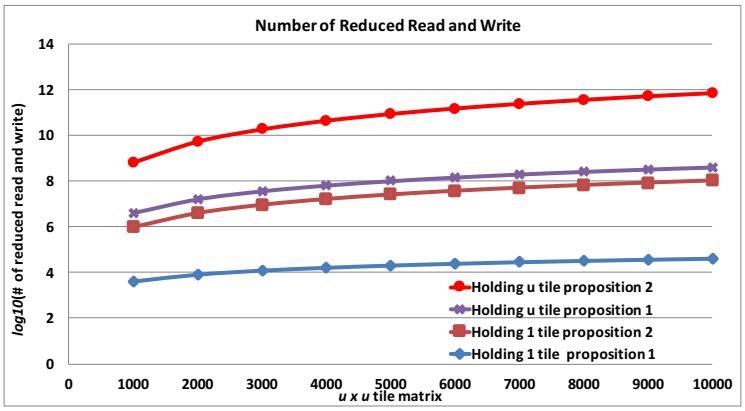


Fig. 7. Amount of reduced tile reads/writes by using the two new strategies that we propose. For a square matrix with $u \times u$ tiles, the amount of communication is $\frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$ tiles to be read and $\frac{2}{3}u^3 + \frac{u^2}{2} - \frac{u}{6}$ tiles to be written.

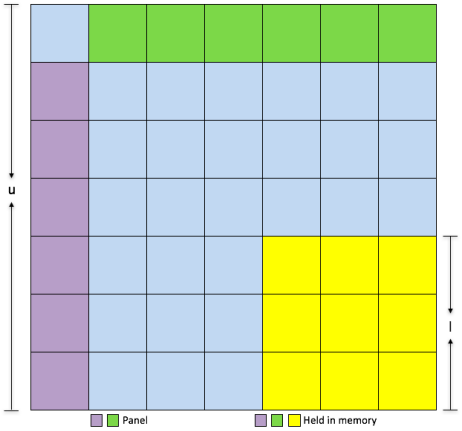


Fig. 8. Extra memory available used to hold tiles to reduce communications

Usually the system memory will have more space, and thus more tiles can be kept in memory. In order to provide a correct model for the runtime estimation, we present the possible scenarios. First, if the tile size b is larger than the minimal needed to overlap communications by computation, as described in Section 6.1, then the total time can be estimated to be the in-memory computational time, since the communication is hidden in this case. The reduction operations – $\frac{8}{3}n^3$, where $n = u \times b$ – are mostly computed by the update kernel. Thus, the estimated time is equal to:

$$T_{est} = \frac{8n^3}{3\alpha},$$

where α is the performance of the update kernel.

The other scenario, which is commonly faced in practice for the DRAM/disk case, is when b is smaller than the minimal tile size required to entirely hide the communication. In this case, the estimated run-time is given by:

$$T_{est} = T_{read} + T_{write} + T_{compute},$$

where T_{read} and T_{write} are the times required for all the read and write, respectively, and $T_{computation}$ is the computation time. $T_{computation}$ is not straightforward to estimate. For the yellow area, it is equal to the computational time of the kernel since this data is in memory, while for the white tiles, it is equal to the communication time (since the tile size b is smaller than the minimal, the communication time is larger than the computation time). So, $T_{computation}$ will be included in T_{read} and T_{write}, and therefore, $T_{computation} = l^2 \frac{5b^3}{\alpha}$.

We define T_{opti} to be the time that has been optimized by avoiding the read and the write of the yellow area. From steps 1 to $u-l$, each tile in the yellow area is read twice and written twice (see Section 7.2 and Figure 6b), meaning that at every step for each tile of the yellow area, we optimize 2 reads and 2 writes in terms of communication. Consequently, we optimize $(v-l) \times (2R+2W)$. For the steps from $l+1$ to u all the tiles are in memory and thus all the read and the write required for this area are avoided, which is $\frac{2}{3}l^3 + \frac{1}{2}l^2 - \frac{1}{6}l$ reads and $\frac{2}{3}l^3 + \frac{1}{2}l^2 - \frac{1}{6}l$ writes. Therefore:

$$T_{opti} = (2(v-l) + \frac{2}{3}l^3 + \frac{1}{2}l^2 - \frac{1}{6}l)(R+W).$$

Thus, the T_{read} or T_{write} is equal to the total amount of read needed without holding or optimizing, minus the T_{opti} for the read:

$$T_{write} = T_{read} = \frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u^2 - 2(v-l) - \frac{2}{3}l^3 - \frac{1}{2}l^2 + \frac{1}{6}l$$

Consequently, the model for estimating the time of the OOM SVD is defined by:

$$Test = \frac{4}{3}u^3 + u^2 - \frac{1}{3}u^2 - 4(v-l) - \frac{4}{3}l^3 - l^2 + \frac{1}{3}l + l^2\frac{5b^3}{\alpha}. \quad (2)$$

9 Experimental results

To evaluate the performance of the OOM two-stage algorithm, we have done a set of experiments. This section presents the results and analysis of the experimental data collected. We run our experiment on both Haswell i7-5930K and Haswell E5 2650 V3 machines. We use a few different systems to run our experiments. The details of the machines we used are given in Table-2. We first studied the effect of the bandwidth size on the performance. A low bandwidth predestines low performance for the OOM solver since communication will be dominant. The maximum performance in this case will be bound somewhere in the lower portion of the roofline model, since the block size must be small. Consequently, the generation of many small Read/Write tasks further increases disk traffic, and affects bandwidth of the disk negatively. Even though

	SSD System: Haswell i7-5930K	**Spindle System:** Haswell Xeon E5 2650V3
Clock	3.5 GHz	2.3 GHz
Core	6	10
Memory	32GB	32GB
L2 Cache	15 MB	25 MB
Peak performance	336 Gflop/s	368 Gflops
Disk	Samsung SSD EVO 465 GB	Seagate ES.3 1000 GB

Table 2. Machine configurations.

Samsung SSD and Seagate Constellation HDD have high theoretical R/W bandwidth, we are unable to achieve it because of complex access order and small tile sizes. Big tile sizes help to have less tasks and overcome some of these short comings, but at the same time increase run-time for the second stage (reduction of band matrix to bidiagonal form) of the two-stage algorithm. Table 3 shows the effect of tile size for the OOM SVD solver for $100k \times 60k$ matrix when we run it on the Spindle System. Basically, the second stage of the SVD solver is memory bound, and its performance depends on the memory bandwidth. Table 3 shows the execution time for both stages for two tile size. Big tile size, e.g., 512, improves the performance of the first stage compared to tiles of size 128, but it requires longer time for the second stage. However, since the second stage runs in-memory, its time remain negligible compared to the first stage.

Tile size	Obtained disk bandwidth (GB/s)	Update kernel performance (Gflop/s)	First stage of two-stage SVD algorithm		Second stage of two-stage SVD algorithm time(hour)
			Estimated time(hour)	Obtained time(hour)	
128	80	300	13.42	13.90	0.06
512	110	300	3.94	3.68	0.53

Table 3. Effect of tile size on the runtime of the two-stage OOM SVD algorithm on the Spindle System, with a matrix of size $100k \times 60k$.

Tables 4 and 5 present details of our experiments. We report the obtained bandwidth (e.g., the average of the measured bandwidth) as well as the performance of the main kernel (e.g., the update kernel) since our roofline performance model depends on it. We also report the estimated time to compute the first stage using Equation-2 and the actual time obtained during our runs. Moreover, we report the time to performs the second stage, and thus the total time for the OOM reduction to bidiagonal using the two-stage algorithm. To give the reader a clear view about the benefit and the efficiency of our proposed OOM algorithm, we show the estimated runtime for an OOM reduction to bidiagonal using the standard one-stage algorithm. our OOM SVD solver uses the maximum amount of memory that the system allows us to use (32 GB on these systems). From Tables 4 and 5 we can observe that the estimated execution time for the first stage is close to the observed run-time. This also highlights the importance of the performance

analysis discussed above, and shows that our performance model is good enough. For example, in the Spindle System (e.g., Haswell E5 2650V3 machine), for $100k \times 100k$ matrix, the actual run time for the first stage is 19.04 hours, whereas the estimated run-time using Equation-2 is 19.70 hours.

			Benchmarking Bandwidth & Perf		two-stage BRD reduction				one-stage BRD reduction	
Matrix	Size	Tile size	Obtained bandwidth (MB/s)	Obtained Update kernel performance (Gflop/s)	Estimated OOM first stage time(h)	Obtained OOM first stage time(h)	Obtained second stage time(h)	Obtained OOM two-stage time(h)	Estimated OOM one-stage time(day)	Obtained OOM one-stage time
100k x 20k	16GB	128	180	160	0.32	0.33	0.05	0.38	1.50h	1.48h
100k x 40k	32GB	128	180	160	1.21	1.20	0.20	1.40	5.25h	5.2h
100k x 60k	48GB	512	145	160	4.50	4.36	0.48	4.84	184	N/A
100k x 80k	64GB	512	145	160	10.19	9.90	0.84	10.74	300	N/A
100k x 100k	80GB	512	145	160	17.73	17.21	1.30	18.51	426	N/A

Table 4. Obtained and estimated runtime of the two-/one- stage algorithms on the SSD System.

			Benchmarking Bandwidth & Perf		two-stage BRD reduction				one-stage BRD reduction	
Matrix	Size	Tile size	Obtained bandwidth (MB/s)	Obtained Update kernel performance (Gflop/s)	Estimated OOM first stage time(h)	Obtained OOM first stage time(h)	Obtained second stage time(h)	Obtained OOM two-stage time(h)	Estimated OOM one-stage time(day)	Obtained OOM one-stage time
100k x 20k	16GB	128	130	300	0.174	0.171	0.005	0.176	1.4h	1.3h
100k x 40k	32GB	128	130	300	0.64	0.58	0.02	0.60	5.1h	5.0h
100k x 60k	48GB	512	110	300	3.94	3.68	0.53	4.22	242	N/A
100k x 80k	64GB	512	110	300	10.52	10.28	0.96	11.24	395	N/A
100k x 100k	80GB	512	110	300	19.70	19.04	1.54	20.57	562	N/A

Table 5. Obtained and estimated runtime of the two-/one- stage algorithms on the Spindle System.

Tables 4 and 5 show the overall runtime for the two-stage OOM SVD solver for both systems. We compare it to the obtained/estimated time for an OOM SVD solver by using the standard one-stage algorithm since both methods reduce a general matrix to bidiagonal form. To make the estimation as accurate as possible, we did not use the manufacturer data for the bandwidth and performance, instead, we used α and β obtained from benchmarking the bandwidth and the update kernel performance. As seen in the tables, and as expected based on our theoretical study presented in Section 5, the OOM two-stage algorithm is much faster than the one-stage algorithm, and can be used in practice, versus the non-practical use of the one-stage algorithm. For example, for $100k \times 100k$ matrix the two-stage OOM SVD algorithm is taking only $20.57 hours$, whereas the one-stage algorithm takes 562 days. Also, we notice that our optimized two-stage OOM SVD can solve big problems that are not possible to solve using the traditional SVD algorithm in limited time. This is because the two-stage OOM SVD reduces disk traffic significantly, using all the strategies and techniques explained above. In addition to what has been described and showed above in term of importance and efficiency of our proposed OOM SVD solver, we note that the $100k \times 20k$ and $100k \times 40k$ test case fits into the main memory, and thus, all algorithms run in-memory. We can see

here that even for in-memory, our two-stage approach remains about 3-5 times faster than the standard one-stage approach. Last, we also note that simply using swap space in a memory constrained environment is not a viable option. We performed SWAP experiment, we force the algorithm to execute in a memory constrained environment by locking away 90% of physical memory from the application. The memory management is thus delegated to the operating system and inactive pages are sent to a disk-backed swap space. The observed disk bandwidth sampled during the execution of the algorithm is lower than 5MB/s. The execution time of the two-stage reduction algorithm using disk SWAP was about 580 times more expensive for a small test case of size $10k \times 10k$, while the one using the one-stage algorithm cannot complete after a full two days of execution. For that, we consider that using the swap disk is not a acceptable option at all.

10 Conclusion

We developed and presented the analysis of the communication costs for the one-stage and two-stage SVD algorithms on hierarchical memories. Different communication avoiding strategies were investigated and a design with optimal communication pattern was developed. Moreover, techniques to hide communication overheads for the OOM SVD were created. Optimized implementations of the algorithms developed now enable us to solve efficiently SVD problems where the matrix is too large and does not fit into the system memory, and for which traditional SVD algorithms can not be used. We provided a clear picture about the possible optimizations and improvements. Future work includes efforts to further improve the performance of the OOM SVD by developing OOM QR factorization for tall matrices. The idea here is to precede the SVD by an OOM *QR* decomposition, and then perform an in-memory SVD on the small upper triangular matrix *R*. In case *R* does not fit in-memory, our OOM SVD can be applied to it to still benefit from *R*'s smaller size.

Acknowledgments

This material is based upon work supported by the US Department of Energy, Nvidia Corporation, and Intel Corporation.

References

1. E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. http://www.netlib.org/lapack/lug/.
2. Christian Bischof, Bruno Lang, and Xiaobai Sun. Parallel tridiagonalization through two-step band reduction. In *In Proceedings of the Scalable High-Performance Computing Conference*, pages 23–27. IEEE Computer Society Press, 1994.
3. Christian H. Bischof, Bruno Lang, and Xiaobai Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM TOMS*, 26(4):602–616, 2000.

4. Eduardo F. D'Azevedo and Jack Dongarra. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurrency - Practice and Experience*, 12(15):1481–1493, 2000.
5. Jack J. Dongarra, Danny C. Sorensen, and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215 – 227, 1989.
6. J.J. Dongarra, S. Hammarling, and D.W. Walker. Key concepts for parallel out-of-core lu factorization. *Computers & Mathematics with Applications*, 35(7):13 – 31, 1998.
7. WilfriedN. Gansterer, DieterF. Kvasnicka, and ChristophW. Ueberhuber. Multi-sweep algorithms for the symmetric eigenproblem. In *Vector and Parallel Processing - VECPAR'98*, volume 1573 of *Lecture Notes in Computer Science*, pages 20–28. Springer, 1999.
8. Roger Grimes, Henry Krakauer, John Lewis, Horst Simon, and Su-Hai Wei. The solution of large dense generalized eigenvalue problems on the cray X-MP/24 with SSD. *J. Comput. Phys.*, 69:471–481, April 1987.
9. Roger G. Grimes and Horst D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14:241–256, September 1988.
10. A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications*, September 2012. (accepted).
11. Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. *SC '12: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.
12. Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of SC '11*, pages 8:1–8:11, New York, NY, USA, 2011. ACM.
13. Azzam Haidar, Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 21-25 2012. ISBN 978-1-4673-0975-2.
14. Bruno Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14:1320–1338, November 1993.
15. Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. *ACM TOMS*, 39(3), 2013. In publication.
16. Hatem Ltaief, Piotr Luszczek, Azzam Haidar, and Jack Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Proceedings of 9th International Conference, PPAM 2011*, volume 7203, pages 661–670, Torun, Poland, 2012.
17. Eran Rabani and Sivan Toledo. Out-of-Core SVD and QR Decompositions. In *PPSC*, 2001.
18. Sivan Toledo and Fred G. Gustavson. The Design and Implementation of SOLAR, a Portable Library for Scalable Out-of-core Linear Algebra Computations. In *Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems: Part of the Federated Computing Research Conference*, IOPADS '96, pages 28–40, New York, NY, USA, 1996. ACM.
19. Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. One-sided dense matrix factorizations on a multicore with multiple gpu accelerators*. *Procedia Computer Science*, 9:37 – 46, 2012.
20. Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. Non-GPU-resident Dense Symmetric Indefinite Factorization. *Concurrency and Computation: Practice and Experience*, 11-2016 2016.