Out Of Memory SVD Solver for Big Data

Azzam Haidar*, Khairul Kabir¶, Diana Fayad§, Stanimire Tomov*, Jack Dongarra*†‡
{haidar|fayad|tomov|dongarra}@icl.utk.edu,
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA
†Oak Ridge National Laboratory, USA
‡University of Manchester, UK
§Universite of Versailles Saint-Quentin-En-Yvelines, France
¶Nvidia, USA

Abstract— **Many applications – from data compression to numerical weather prediction and information retrieval – need to compute large dense singular value decompositions (SVD). When the problems are too large to fit into the computer's main memory, specialized *out-of-core* algorithms that use disk storage are required. A typical example is when trying to analyze a large data set through tools like MATLAB or Octave, but the data is just too large to be loaded. To overcome this, we designed a class of *out-of-memory* (OOM) algorithms to reduce, as well as overlap communication with computation. Of particular interest is OOM algorithms for matrices of size $m \times n$, where $m >> n$ or $m << n$, e.g., corresponding to cases of too many variables, or too many observations. To design OOM SVDs, we first study the communications cost for the SVD techniques as well as for the QR/LQ factorization followed by SVD. We present the theoretical analysis about the data movement cost and strategies to design OOM SVD algorithms. We show performance results for multicore architecture that illustrate our theoretical findings and match our performance models. Moreover, our experimental results show the feasibility and superiority of the OOM SVD.**

I. 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 BRD*: 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 (BRD). 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.

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.

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 [12], [5], [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 [13], [14].

Similar algorithms are computationally not feasible for the standard eigensolvers or SVD problems in LAPACK, as we showed recently [7], and therefore have not been developed before. Exceptions are special cases that we target here, when $m >> n$ or $m << n$, where a direct SVD computation can be replaced by an out-of-core QR (or LQ) first, followed by an in-core SVD of the resulting small R (or L, respectively).

II. Contributions

The primary goal of this paper is to design OOM SVD algorithms and efficient implementations that reduce, as well as overlap communications with computations as much as possible. Our main contributions towards this goal are:

- We developed and presented the analysis of the communication costs for the SVD algorithms as well as for QR factorization on hierarchical memories, e.g., CPU memory for main memory and disk for out-of-memory storage, or the GPU/Coprocessor for main memory and CPU DRAM for out-of-memory storage;
- We discussed techniques, along with their theoretical analysis, to hide communication overheads for OOM QR;
- We also designed a OOM SVD algorithm and developed an optimized implementation for multicore architecture based on OOM QR + SVD on R. We showed performance

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

results that illustrate its efficiency and high performance which highlight the feasibility to compute whatever was not possible to compute in a near past.

III. Study of the BRD algorithm

The reduction of a matrix A to a bidiagonal form, as implemented in LAPACK, applies orthogonal transformation matrices on the left and right side of A to reduce A to bidiagonal, for that it is called a "two-sided factorization." The blocked BRD [6] proceeds by steps of "panel/trailing matrix update" and can be summarized as follows. At every step, the panel factorization zeroes the entries below the subdiagonal and above the diagonal. It goes over its "*nb*" columns and rows (red portion in 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 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:

$$A_{s+nb:n,s+nb:n} \leftarrow A_{s+nb:n,s+nb:n} - U \times Y^{\top} - X \times V^{\top}, \qquad (1)$$

where s denotes the step and nb denotes the panel width. The process is repeated until the whole matrix is reduced to a bidiagonal form. The total cost of such algorithm is

$$BRD = 4mn^2 - 4/3n^3 + 3n^2 - mn + 25/3n \qquad (2)$$

IV. Theoretical study of the communication cost of data movement for the OOM BRD Reduction

In this section we develop and present the communication pattern for the OOM reduction to bidiagonal form. The bidiagonal reduction needs two matrix-vector multiplications (dgemv) with the trailing matrix at every column and row annihilation, and two matrix-matrix multiplications (dgemm) 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 dgemv as well as loaded and stored back once after each panel to perform the two dgemm 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 (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:

$$\begin{aligned}
&\mathrm{R}(A)\ \texttt{dgemv}\ \#1 + \mathrm{R}(A)\ \texttt{dgemv}\ \#2 + \mathrm{R/W}(A)\ \texttt{dgemm} \\
&= \sum_{s=0}^{min(m,n)-1} (m-s)(n-s) + \sum_{s=0}^{min(m,n)-1} (m-s)(n-s-1) \\
&\quad + 2\sum_{s=1}^{n/nb} (m - s \times nb)(n - s \times nb). \\
&= (mn^2 - \frac{n^3}{3})(1 + \frac{1}{nb}) + \frac{n^2}{2} - 3mn + \frac{n}{3}(\frac{5}{2} + nb) \\
&\approx \quad mn^2 - \frac{n^3}{3} + n^2/2 - 3mn \qquad (3)
\end{aligned}$$

From this formulation, one can easily observe that the reduction to bidiagonal requires a large amount of data movement compared to the total cost in term of number of operations depicted in Equation (2).

Tricks to hide the communication with the computation cannot be used here since the whole matrix need to be loaded or stored at each step and the cost of reading/storing is more expensive than the computation. To highlight the importance of the communications, we start by giving an example. Figure 2 show the time required to perform the OOM bidiagonal reduction using a recent hardware, Solid State Drives (SSD), or out-of-GPU memory where the communication bandwidths are about 150 MB/s, 500 MB/s, respectively, for both square and tall-skinny matrices. We mention that for the one-sided factorisation (such as Cholesky, LU or QR) the matrix do not need to be loaded/stored at each step, thus some tricks and algorithmic variant (e.g., left looking) can be used to hide the cost of communication with the computation. We discuss this techniques later in the section related to the OOM QR.

In conclusion, these results illustrate that it is unacceptable to build an OOM bidiagonal reduction algorithm. For that, it has been thought that the OOM SVD implementation is practically impossible. However, for big data analysis most of the matrices are tall-skinny matrices. In this case, there is an alternative technique that minimizes the cost in both number of operations and data movement. Such technique is well known for tall-skinny matrices and it consists of performing a QR factorisation first and then a bidiagonal reduction on the R of the QR which is small and can be performed in memory. In this paper we are interested in this type of computation.

V. The OOM QR+SVD Algorithm

The QR factorization of an $m \times n$ real matrix A is the decomposition of A as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is a $n \times n$ real upper triangular matrix. QR factorization generates a smaller $n \times n$ upper triangular matrix R when $m \gg n$. Instead of reducing A to bidiagonal form directly for SVD, the following two-step approach can be adopted.

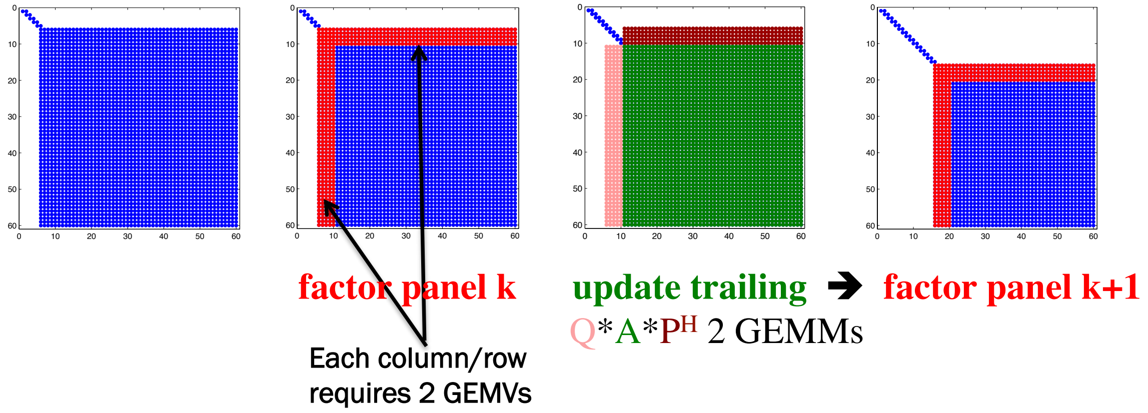


Fig. 1. LAPACK one-stage blocked algorithm: illustration of the main BLAS kernels used.

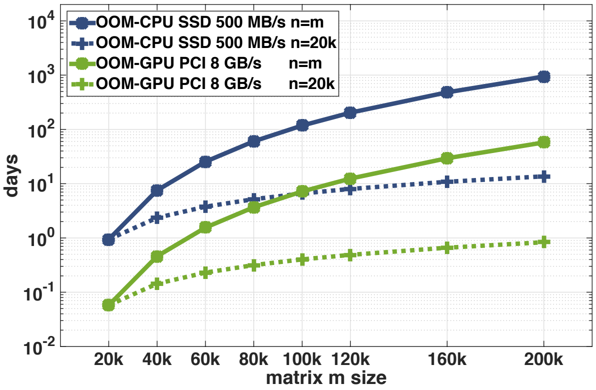


Fig. 2. Estimated OOM SVD time based on the amount of R/W.

- Compute QR factorization of the general matrix A, as $A = QR$.
- Compute singular value decomposition of the matrix R, as $R = U_1 \Sigma V_1^T$.

Both A and R matrices have the same singular values. The computational complexity for computing QR + BRD for a matrix A of size (m,n) where $m \gg n$ is as follows:

$$QR\ for\ A(m,n) + BRD\ for\ R(n,n)$$
$$2n^2(m - \frac{n}{3}) + \frac{8}{3}n^3$$
$$2n^2(m+n)$$

The computation complexity to reduce directly the $m \times n$ matrix A to bidiagonal form is $(4mn^2 - \frac{4}{3}n^3)$ flop, while the two-step approach requires $2n^2(m+n)$ flop only. The advantages of two step approach are as follows:

- One-sided factorization i.e., QR is faster than two-sided factorization, as transformations are applied from one side.
- For a $m \times n$ matrix A, if $m \gg n$, the upper triangular matrix, R, may fit in main memory and an in-memory SVD algorithm will be used. Otherwise, the OOM SVD of R must be faster than OOM SVD of A, since R is much smaller than A. This later is not discussed in this paper.

A. The QR Algorithm

In LAPACK, QR factorization is performed as a blocked algorithm by the dgeqrf [1] routine for double precision. The QR factorization algorithm is a two-phases process:

- Panel factorization — for a panel of size nb, nb columns are factored using Householder transformations as shown in Figure 3.
- Update trailing matrix — in the update phase, the nb transformations that are generated during the panel factorization are applied all at once to the rest of the trailing sub-matrix by Level-3 BLAS operations (dlarfb).

The process is repeated until all columns have been factored. The panel factorization process is rich in Level-2 BLAS operations and does not scale well on a multicore system, as Level-2 BLAS cannot be efficiently parallelized. The execution flow of a block factorization algorithm represents a fork-join model between the panel factorization and the updates of the trailing sub-matrix. The problem of fork-join bottleneck in block algorithms has been overcome in [2], [3], [8], [9], [11] where panel factorization and trailing submatrix updates are broken into smaller task that operates on tile of size b and that can be represented as a DAG. This technique is called tile algorithm. In the DAG, nodes represent tasks and edges represent the dependencies among them. Execution of the algorithm is performed by out-of-order asynchronous execution of the tasks without violating the dependencies, which helps to hide sequential tasks behind fast, parallel ones.

B. The Tile QR Algorithm

High-performance implementation of tile QR factorization is presented in [2], [3] for multicore architecture. The algorithm processes square tile instead of rectangular panel as in an LAPACK blocked algorithm [1]. The tile QR algorithm is presented in Algorithm 1 . It consists of the following four basic computational kernels:

TABLE I
COMPUTATION COST FOR TILE QR KERNEL

Kernel	Computation cost (flop)	Total cost for $u \times u$ tile matrix
geqrt	$2b^3$	$O(u) \times 2b^3 = O(m) \times b^2$
ormqr	$3b^3$	$O(u^2) \times 3b^3 = O(m^2) \times b$
tsqrt	$\frac{10}{3}b^3$	$O(u^2) \times \frac{10}{3}b^3 = O(m^2) \times b$
tsmqr	$5b^3$	$O(u^3) \times 5b^3 = O(m^3)$

- dgeqrt performs the QR factorization of a diagonal tile and generates an upper triangular matrix R and a unit lower triangular matrix V. The lower triangular matrix V contains the Householder reflectors.
- dtsqrt performs the QR factorization of a tile below the diagonal coupled with the R of the diagonal tile produced by either dgeqrt or by the previous dtsqrt. Thus it update the R and generates a square matrix V for Householder reflectors.
- dormqr applies the orthogonal transformations computed by dgeqrt to the right of the diagonal tile.
- dtsmqr applies the orthogonal transformations computed by dtsqrt to the right of the tiles factorized by dtsqrt.

The kernels are represented by a DAG and executed each as a task using a dynamic runtime scheduler. In our example we used *QUARK* [15] but it could be easily replaced by OpenMP [10].

Algorithm 1 Tile QR algorithm.

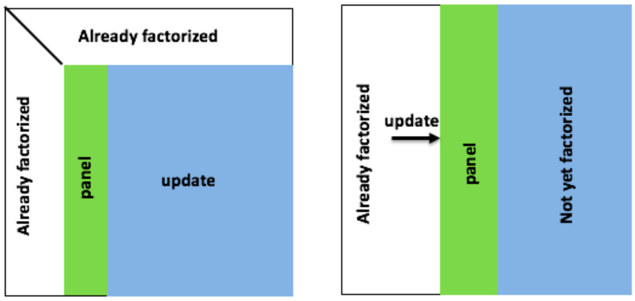
```
  $A.mt = m/nb$;   $A.nt = n/nb$
  for $k \in \{0 \dots min(A.mt, A.nt)\}$ do

    {Panel factorization}
    $A_{kk,kk} \leftarrow GEQRT(A_{k,k})$
    for $m \in \{k+1 \,..\, A.mt\}$ do
      $A_{m,k} \leftarrow TSQRT(A_{k,k}, A_{m,k})$
    end for

    {Trailing matrix update}
    for $n \in \{k+1 \,..\, A.nt\}$ do
      $A_{k,n} \leftarrow UNMQR(A_{k,k}, A_{k,n})$
    end for
    for $m \in \{k+1 \,..\, A.mt\}$ do
      for $n \in \{k+1 \,..\, A.nt\}$ do
        $A_{m,n} \leftarrow TSMQR(A_{k,n}, A_{m,n}, A_{m,k})$
      end for
    end for

  end for
```

For an $m \times m$ matrix with tile size b and, if $u = \frac{m}{b}$, we present in Table I the computation cost for the kernels used in tile QR. The most expensive kernel is the update kernel (tsmqr) and it requires $O(m^3)$ flop. For all our theoretical analysis, we consider the computation cost of the tsmqr routine.



(a) Right-looking (b) Left-looking

Fig. 3. QR factorization, right looking vs. left looking variants.

C. Algorithmic Variants

There exist two algorithmic variants for the *QR* factorization, the left looking one and the right looking one. The right looking variant factorize then update meaning it factorize the current panel and once done it applies the corresponding updates to the right as shown in Figure 3a. The right-looking variant requires access to the trailing matrix for each panel it processes and therefore reads and writes the whole trailing matrix. Meanwhile the left-looking variant update and then factorize meaning it applies all the updates coming from the left side to the current panel as shown in Figure 3b and then it factorize it. Therefore delays subsequent updates of the remaining parts of the matrix and thus need only to read the portion on the left instead of R/W the portion on the right. As a result, the left looking variant was always advantageous for Out Of Memory techniques and this will be our choice for our OOM QR. For that reason we decided to develop our OOM QR algorithm based on the left looking implementation of the tile algorithm.

D. A theoretical study of the design of an OOM QR

The efficiency of the OOM QR algorithm depends on its ability to hide the data communication with the computation. We used the left looking implementation. In order to update a current panel (see the green portion of Figure 4a) the data on the left need to be loaded. It is loaded by chunk into a local in memory workspace (the dark red portion of Figure 4a) we want to investigate the circumstances in which the updating of the panel may be overlapped with the reading of the tiles from the left needed by the update. More specifically, we want to know whether the updating of the green tiles (size - $m \times w$) can be overlapped with the reading of the red tiles into a local temporary workspace as as shown in Figure 4a.

Assume the panel is already loaded into memory. If u is the number of tiles in the vertical direction, then, $u = \frac{m}{b}$ where b is the size of the tile. Let define $w = k \times b$. The computation cost for the update kernel (tsmqr) is $5b^3$ flops. Let's define α the computational performance that can be achieved by the update kernel and β the bandwidth that can be reached by the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

read/write of the data from the OOM storage. Thus the time to update the panel t_{update}, is given by:

$$t_{update} = \frac{(u-1) \times k \times 5b^3}{\alpha} \approx \frac{m \times w \times 5b}{\alpha}$$

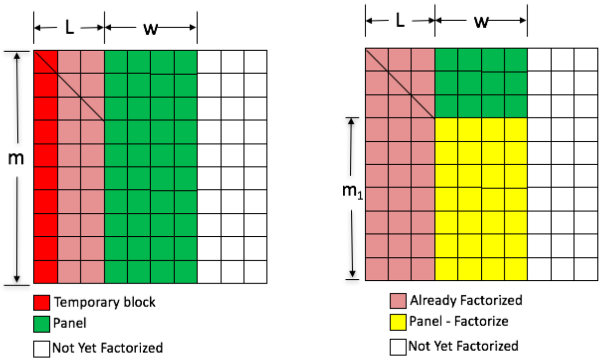
The time to read the red tiles into the temporary local memory storage, t_{read} is:

$$t_{read} = \frac{u \times 8b^2}{\beta \times 10^6} = \frac{m \times 8b}{\beta}$$

In order to overlap the communication with the update:

$$\begin{aligned} & t_{update} \geq t_{read} \\ & => \frac{m \times w \times 5b}{\alpha} \geq \frac{m \times 8b}{\beta} \\ & => w \geq \frac{1.6\alpha}{\beta} \end{aligned}$$

Hence, if the panel width, w, is at least $\frac{1.6\alpha}{\beta}$, the updating of the panel, overlaps with the reading of the tiles into the temporary storage. For example, for a Haswell E5 2650V3 machine where $\alpha = 300 Gflop/s$ and $\beta = 150 MB/s$, if the panel width is 3200, the panel updating overlaps with the reading of the tiles into the "*in memory*" temporary storage, meaning that the cost of reading from the out of Memory can be hidden with the computation and thus the OOM QR factorization will perform at the same speed as the *in-memory* QR. One can see that overlapping the updating of the panel with reading depends only from the panel width, not from the panel height. Thus, we do not need to wait for the entire column of tiles to be loaded into the temporary block in order to start the computtaion. This is another advantage of using tile algorithms.



(a) Updating the panel.

(b) Factorizing the panel.

Fig. 4. Left-looking tile QR — updating and factorizing

VI. Experimental Results of the OOM solver

To evaluate the performance of the OOM QR tile algorithm, we have conducted a number of experiments and collected execution traces to show how it overlaps with tile reading/writing from/to the disk.

We note that there is many other optimization challenges that are implemented in order to increase the performance. We will give brief description of these optimization. First of all, the way the tiles are read from the Out of Memory storage is important since it define the achievable bandwidth. For example a row-wise tile reading is slower than a column-wise reading for that it is beneficial to submit the tasks in a way to prioritize the column-wise reading. Second, if too many "reading tasks" are submitted and since they are independent, these tasks get scheduled into the queue which will delay the computation tasks, and thus a careful order of submission might help. Third, As the reading and writing from the disk is kind of a sequential process (meaning that 1 thread is enough to achieve the bandwidth), thus having more threads to read will not help rather it might slowdown. For that, we force all the Read/Write task to be executed by 1 particular thread. When the panel get updated, the only tile that need to be factorized are the one from the diagonal and below (yellow tiles of Figure 4b) thus the above diagonal one can be written back to the disk while the factorization is happening. This also require a careful attention in the submitting order of the task to the scheduler.

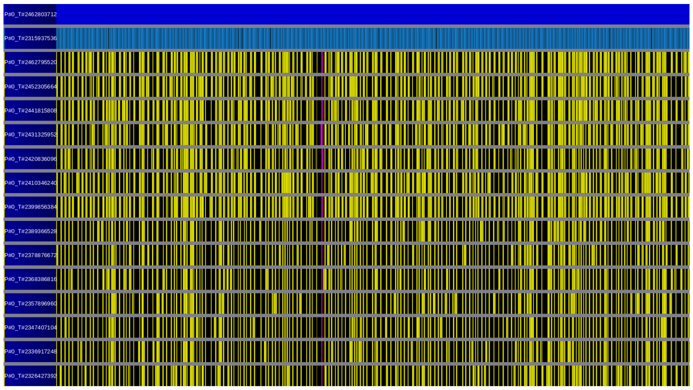


Fig. 5. OOM Tile QR algorithm: Trace of the left looking panel update and factorize.

First we used a panel width less than $\frac{1.6\alpha}{\beta}$ and showed in Figure 5 the execution trace of the OOM QR. The read tasks are represented in blue, and the computational task in yellow and some red. As we can see, the yellow portion is sparse meaning that the computation is waiting for data to arrive. The communication overlap is partial. In Figure 6 we illustrate the trace when the panel width is larger than $\frac{1.6\alpha}{\beta}$ and where the optimization cited above are applied. We can easily see that the trace is more condensed, the blue task (Read task) are completely overlapped with the computation (yellow tasks) as well as the writing back (the write tasks in purple) start as soon as the data concerned is done (meaning either it is in the above diagonal portion and so when its update finish, it is ready to be written back, or it has to be factorized and it is written back as soon as its factorization finish).

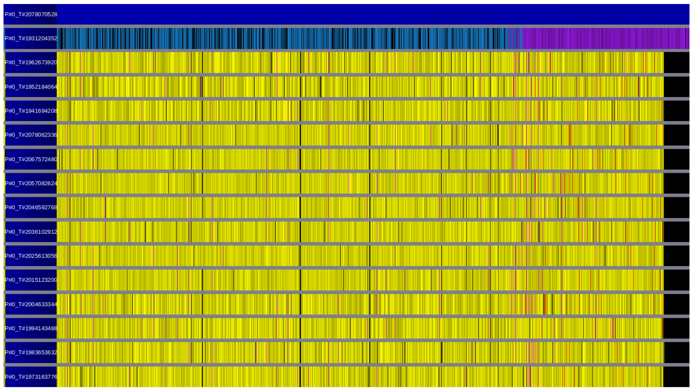


Fig. 6. OOM Tile QR algorithm: Trace of the optimized left looking panel update and factorize.

	System A:SSD Haswell i7-5930K	**System B:Spindle** Haswell Xeon E5 2650V3
Core	6	10
Memory	32GB	32GB
in memory QR peak performance	170 Gflop/s	300 Gflops
Achievable bandwidth	430 MB/s	150 MB/s

TABLE II
MACHINE CONFIGURATIONS.

A. OOM QR Performance

In this section we present the performance of our OOM QR when the matrix does not fit in the main memory. We have run our experiment on both Haswell i7-5930K and Haswell E5 2650 V3 machines. The details of the machines we used are given in Table-II.

We compare the performance of our OOM QR with the extrapolated performance of in-memory QR. To compute the extrapolated performance of the *in memory* QR, we take the peak that the "*in memory* QR can achieve for large matrices that fit into the main memory (meaning the asymptotic peak) and extrapolate. This is considered to be an upper bound of what the *in memory* algorithm can achieve. The peak performance value is summarized in Table II. Obviously, we consider that the original matrix is in the disk and so it need to be read/written back even for the *in memory* algorithm. Table III and Table IV show the performance of our OOM QR for the Haswell i7-5930K and Haswell E5 2650V3 machines respectively. We have also presented the "*in-memory* QR performance that we extrapolated using Table II. We would like to highlight the attractive performance of our OOM QR that is close to the extrapolated performance of the in-memory QR for most of the test cases. This mean that our design is overlapping the data transfer with the computation and behave as if the data is in memory. Note that, for System B, the OOM QR can be slightly faster than the *in memory* QR because the OOM algorithm can start the computation while still reading whereas the *in memory* has to wait that the whole matrix is in memory to start the computation.

TABLE III
OUR OOM QR PERFORMANCE ON SYSTEM A

Matrix	Read +Write time(s)	Extrapolated in-memory QR time(s)	Extrapolated in-memory QR time(s) (with read write)	Our OOM QR(s)
100k x 40k	148	1631	1779	1802
100k x 60k	222	3388	3610	3694
100k x 80k	296	5521	5817	5932
100k x 100k	372	7843	8215	8309

TABLE IV
OUR OOM QR PERFORMANCE ON SYSTEM B

Matrix	Read +Write time(s)	Extrapolated in-memory QR time(s)	Extrapolated in-memory QR time(s) (with read write)	Our OOM QR(s)
100k x 40k	426	924	1350	1341
100k x 60k	640	1920	2560	2527
100k x 80k	852	3128	3980	3959
100k x 100k	1066	4444	5510	5485

B. OOM SVD using OOM QR

In this section we present performance of our OOM SVD solver for tall-skinny matrices using OOM-QR and *in memory* SVD. In the first step, we factorized the original matrix A using our OOM QR and then since the matrix R fit in memory we used *in memory* SVD solver for the SVD computation.

Table V presents the performance of our OOM SVD for tall-skinny matrices. The column "Our OOM SVD" is the sum of the first two columns of Table V. The column "extrapolated OOM SVD(A)" is the extrapolation of the time for computing Out Of Memory SVD directly on A based on Equation (3). As mentioned in Section IV, an OOM SVD on A for large A is not feasible since it requires a lot of time to finish. This is also seen in the fourth column of Table V.

Consequently, when comparing our OOM SVD (that computes an OOM QR first, followed by an *in memory* SVD) to a hypothetical OOM SVD on A directly, our algorithm is about $4,000\times$ faster. As many applications require SVD for tall-skinny matrices, our OOM SVD for tall-skinny matrices can solve such big problems in a brief time.

TABLE V
OUR OOM SVD PERFORMANCE ON SYSTEM B

Matrix	Our
	OOM QR(A)
	time(s)
		In memory
		SVD(R)
		time(s)
			Our OOM SVD
			OOM QR(A)+SVD(R)
			hours
				extrapolated
				OOM SVD(A)
				days

100k x 40k	1331	908	0.62	81
200k x 40k	2804	908	1.03	175
300k x 40k	4278	908	1.44	270
400k x 40k	5752	908	1.85	364

VII. CONCLUSION

We developed and presented the analysis of the communication costs for an OOM SVD algorithms on hierarchical memories. We discussed techniques to improve the performance of the OOM SVD for tall-skinny matrices by developing OOM QR factorization. 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*. We presented tile algorithm for the OOM QR implementation as well as we analyzed techniques to hide communication overheads.

Optimized implementations of the proposed algorithm 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. Future work includes efforts to further improve our solver to the case where *R* does not fit in memory. In this case an OOM SVD solver for *R* will be developed.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The work was also partially supported by Nvidia and NSF under grant No. 1514406.

REFERENCES

[1] 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, 3rd edition, 1999.

[2] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[3] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

[4] E. F. D'Azevedo and J. 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] J. Dongarra, S. Hammarling, and D. Walker. Key concepts for parallel out-of-core lu factorization. *Computers & Mathematics with Applications*, 35(7):13 – 31, 1998.

[6] 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.

[7] K. Kabir, A. Haidar, S. Tomov, A. Bouteiller, and J. Dongarra. A Framework for Out of Memory SVD Algorithms. In *ISC High Performance 2017*, Frankfurt, Germany, 06-2017 2017. Springer, Springers Lecture Notes in Computer Science (LNCS).

[8] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, Sept. 2008.

[9] J. Kurzak and J. Dongarra. QR Factorization for the CELL Processor. LAPACK Working Note 201, May 2008.

[10] OpenMP application program interface, July 2013. Version 4.0.

[11] E. S. Quintana-Ortí and R. A. van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. Math. Softw.*, 35(2):11, 2008. DOI: 10.1145/1377612.1377615.

[12] S. Toledo and F. 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.

[13] I. Yamazaki, S. Tomov, and J. Dongarra. One-sided dense matrix factorizations on a multicore with multiple gpu accelerators*. *Procedia Computer Science*, 9:37 – 46, 2012.

[14] I. Yamazaki, S. Tomov, and J. Dongarra. Non-gpu-resident symmetric indefinite factorization. *Concurrency and Computation: Practice and Experience*, 29(5):e4012–n/a, 2017. e4012 cpe.4012.

[15] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011.