5-2017

Analysis and Design of Communication Avoiding Algorithms for Out of Memory(OOM) SVD

Khairul Kabir
University of Tennessee, Knoxville, kkabir@vols.utk.edu

Recommended Citation

Kabir, Khairul, "Analysis and Design of Communication Avoiding Algorithms for Out of Memory(OOM) SVD. " PhD diss., University of Tennessee, 2017.
https://trace.tennessee.edu/utk_graddiss/4472

To the Graduate Council:

I am submitting herewith a dissertation written by Khairul Kabir entitled "Analysis and Design of Communication Avoiding Algorithms for Out of Memory(OOM) SVD." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Michael Berry, Gregory Peterson, Bruce Ralston

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Analysis and Design of Communication Avoiding Algorithms for Out of Memory(OOM) SVD

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Khairul Kabir

May 2017

This dissertation is dedicated to my family:

my wife, Tasneem, for her love and suppor;

my parents, brother and sisters, and uncles, who inspired me on this path;

my mother-in-law, for her support and encouragement; and

my son, Farzin, for bringing so much joy to my life.

Acknowledgements

I would like to thank my adviser, Dr. Jack Dongarra, for his guidance and support during my graduate study at the University of Tennessee. Without his help I could not have completed this work. Thanks also to Dr. Michael Berry, Dr. Gregory Peterson, and Dr. Bruce Ralston for serving on my doctoral committee. I greatly appreciate the time and feedback they contributed to my dissertation and research.

I will be forever grateful to my colleagues and mentors, Dr. Azzam Haidar and Dr. Stanimire Tomov, for their guidance and help throughout my time at ICL. I would also like to thank Dr. Piotr Luszczek, Dr. Mark Gates, and Dr. Ichitaro Yamazaki for their valuable suggestions and help on my research. In addition, I want to express my appreciation to my fellow students at the Innovative Computing Laboratory for their friendship and many pleasant times.

I am also grateful to my family and friends; to my wonderful wife, Tasneem Halim, for always being supportive; to my parents for all their hard work and patience, as well as to all my brothers and sisters, and uncles and aunts for being so supportive my entire life; and to my mother-in-law, without whose support I would not have completed the writing of this dissertation.

Lastly, special thanks to my son, Frazin Kabir, who has brought so much fun and excitement to my life.

Abstract

Many applications — including big data analytics, information retrieval, gene expression analysis, and numerical weather prediction – require the solution of large, dense singular value decomposition (SVD). The size of matrices used in many of these applications is becoming too large to fit into into a computer's main memory at one time, and the traditional SVD algorithms that require all the matrix components to be loaded into memory before computation starts cannot be used directly. Moving data (communication) between levels of memory hierarchy and the disk exposes extra challenges to design SVD for such big matrices because of the exponential growth in the gap between floating-point arithmetic rate and bandwidth for many different storage devices on modern high performance computers. In this dissertation, we have analyzed communication overhead on hierarchical memory systems and disks for SVD algorithms and designed communication-avoiding (CA) Out of Memory (OOM) SVD algorithms. By *Out of Memory* we mean that the matrix is too big to fit in the main memory and therefore must reside in external or internal storage. We have studied communication overhead for classical one-stage blocked SVD and two-stage tiled SVD algorithms and proposed our OOM SVD algorithm, which reduces the communication cost. We have presented theoretical analysis and strategies to design CA OOM SVD algorithms, developed optimized implementation of CA OOM SVD for multicore architecture, and presented its performance results.

When matrices are tall, performance of OOM SVD can be improved significantly by carrying out QR decomposition on the original matrix in the first place. The upper triangular matrix generated by QR decomposition may fit in the main memory, and in-core SVD can be used efficiently. Even if the upper triangular matrix does not fit in the main memory, OOM SVD will work on a smaller matrix. That is why we have analyzed communication reduction for OOM QR algorithm, implemented optimized OOM tiled QR for multicore systems and showed performance improvement of OOM SVD algorithms for tall matrices.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Singular value decomposition (SVD) problems are fundamental for many computational science and engineering applications. For example, in statistics SVD is directly related to the principal component analysis method [38, 39]; in signal processing and pattern recognition, it is used as an essential filtering tool, and for analysis of control systems [52]. The SVD also plays a very important role in linear algebra. It has applications in such areas as least squares problems [27, 25, 47], computing the pseudoinverse [25], and computing the Jordan canonical form [28]. In addition, SVD is used in information retrieval [41] for filtering and rank reduction of the term-by-document matrix to minimize cost and improve efficiency of the retrieval, in solving integral equations [37], in digital image processing [5], and in gene expression analysis, in seismic reflection tomography [21, 8], and in optimization [6]. Some of these applications require SVD for matrices that are too big to fit in a computer's main memory. The traditional SVD algorithms that require matrix data must be loaded into main memory all at once before the computation begins either cannot solve these problem or is not fast enough to solve it in limited time. In this dissertation we explore methods to solve SVD problems that are too large to fit in main memory and therefore reside in external or internal storage.

The SVD problem [26] for a given $m \times n$ matrix A finds a diagonal matrix Σ of size $m \times n$ and orthogonal (or unitary) matrices U and V of sizes $m \times m$ and $n \times n$, respectively, such that $A = U\Sigma V^\top$ (or $A = U\Sigma V^H$). The diagonal elements of Σ are called singular values of A, and the columns of U and V are called its left and right singular vectors, respectively. SVD decomposition of a dense matrix is computed in a classical three-phase process [54]. 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 condensed form matrix; hence, these are called two-sided factorization. Note that the use of two-sided orthogonal transformations guarantees that A has the same singular values as the reduced matrix, and the singular vectors of A can be easily derived from those of the reduced matrix. 2. **Solution phase:** a singular value solver computes the singular values and the left and right vectors $\tilde{U}$ and $\tilde{V^T}$ of the condensed form matrix. 3. **Back transformation phase:** if required, the left and right singular vectors of A are computed by multiplying $\tilde{U}$ and $\tilde{V^T}$ by the orthogonal matrices used in the reduction phase. This reduction step is called *bidiagonal reduction (BRD)*, and it has always been the most expensive phase of the three. In multicore architecture, the reduction phase consumes 90% of the overall run time when singular values are computed [49] and required approximately 70% of the total run time when both singular values and singular vectors are computed [49]. Because the reduction phase is the expensive one in this dissertation, we have studied classical one-stage and two-stage algorithms that reduce the general matrix to bidiagonal form and analyzed the communication overhead in a heterogeneous memory system to design communication-avoiding Out of Memory (CA OOM) SVD algorithms.

1.2 Thesis statement and contribution

The primary goal of this dissertation is to design CA algorithms for OOM SVD. By *OOM* we mean that the matrix is too large to fit in memory and therefore must reside in external or internal storage. Since the whole matrix can not be loaded into the computer memory all at once, parts of it must be loaded and sent back to storage throughout the algorithm. Efficient algorithms need to be designed to hide this communication overhead. The main contributions of this dissertation are as follows:

CA OOM SVD:

- Analyzed communication cost on heterogeneous levels of memory system — for example, CPU memory for in-memory computation and disk for OOM storage, GPU/Coprocessor memory for in-memory computation, and CPU memory for OOM storage; and investigated communication reduction from one-stage to two-stage OOM SVD algorithms.
- Presented theoretical analysis and strategies to hide communication overhead for OOM SVD, and provided the necessary conditions to develop CA algorithms.
- Designed CA OOM SVD algorithms and developed an optimized implementation for multicore architecture.

OOM SVD through OOM QR factorization:

When the matrix is tall and does not fit in the main memory, the original matrix is factorized by QR factorization and SVD is computed from the R matrix, as both the original matrix and R matrix have the same singular values. The R matrix may fit in the main memory and classical SVD algorithm can be used directly; otherwise, OOM SVD will handle the smaller matrix.

- Analyzed communication overhead for OOM tiled QR and implemented left-looking tiled QR for multicore architecture.

- Improved the performance of OOM SVD through OOM QR.

1.3 Outline of the dissertation

This dissertation is organized as follows:

- Chapter 2 introduces SVD and its application. It also represents classical SVD algorithms and their complexity.
- Chapter 3 presents the theoretical analysis of SVD communication on the heterogeneous levels of memory system and strategies to develop CA algorithms for OOM SVD.
- Chapter 4 presents analysis to reduce communication for OOM tile QR and performance improvement of OOM SVD through OOM tile QR for tall matrices.
- Chapter 5 concludes the dissertation and discusses possible future extensions.

Chapter 2

Background

2.1 Introduction

The **singular value decomposition**, or SVD, is a very powerful technique used to deal with matrix problems in general. In recent years, the SVD has become an essential tool for solving a wide variety of problems that arise in many practical applications. The use of the SVD in these applications provides information about the rank of a matrix, or approximates a matrix using a lower rank approximation, or forms orthogonal bases for the row and column spaces of a matrix.

The SVD of an $m \times n$ matrix A is a factorization of the following form:

$$A = U\Sigma V^T \ (A = U\Sigma V^H \text{ in the complex case}),$$

where U is an $m \times m$ and V is an $n \times n$ real or complex unitary matrix and Σ is an m-by-n rectangular diagonal matrix with real elements, σ_i, such that:

$$\sigma_1 \geq \sigma_2 \geq \ldots \sigma_{\min(m,n)} \geq 0.$$

The diagonal entries σ_i of Σ are the **singular values** of A and the first $\min(m, n)$ columns of U and V are the **left** and **right singular vectors** of A.

The SVD is computed very effectively in the following three classical phases as presented by Golub and Kahan in 1965 [24]:

1. The matrix A is reduced to bidiagonal form by applying successive distinct transformations from the left (U_1) as well as from the right (V_1) as : $A = U_1 B V_1^T$ if A is real ($A = U_1 B V_1^H$ if A is complex), where U_1 and V_1 are orthogonal (unitary if A is complex), and B is real and upper-bidiagonal when $m \geq n$ or lower bidiagonal when $m < n$.

2. The SVD of the bidiagonal matrix B is computed: $B = U_2 \Sigma V_2^T$, where U_2 and V_2 are orthogonal and Σ is diagonal as described above. There are several algorithms to compute singular values from bidiagonal matrix but originally the QR iteration is used.

3. If desired, the singular vectors of A are then computed as $U = U_1 U_2$ and $V = V_1 V_2$.

The fist stage, which reduces the general matrix to bidiagonal form is called *bidiagonal reduction* (BRD for short) and considered to be the most expensive phase. The second phase computes singular values of bidiagonal matrix using the *divide-and-conquer iteration*. Finally, the third phase computes corresponding singular vectors from the reduced form using either the dqds algorithm [22] or Cuppen's divide-and-conquer algorithm [40, 29] of *divide-and-conquer back-transformation*. The QR iteration [16, 15] is no longer used to compute singular vectors because it consumes roughly 50% more time than the methods mentioned earlier.

The computation cost for bidiagonal reduction is $O(\frac{8}{3}n^3)$, which makes it difficult to design an efficient algorithm and develop an optimized implementation. Two main approaches to solve these problems are as follows:

- One-stage approach: the standard one-stage approach as implemented in LAPACK [3] applies Householder transformations in a blocked fashion to reduce the dense matrix to bidiagonal form directly.

- Two-stage approach: the two-stage approach applies blocked Householder transformations [33] to reduce general matrix to band matrix in the first stage and reduces band matrix to bidiagonal form using a bulge chasing technique in the second stage.

2.2 One-stage bidiagonal reduction

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 sides of A. As opposed to the one-sided factorizations (i.e., LU, Cholesky, QR/LQ), the computed transformations are applied from both the left and right sides of A; therefore, it is called "two-sided factorization." The blocked bidiagonal reduction algorithm as described in [20], can be summarized as follows:

To reduce a matrix A of size $m \times n$ to bidiagonal form, two orthogonal matrices U_1 and V_1 are applied on the left and the right sides, respectively, $B = U_1^T A V_1$. The matrices U_1 and V_1 are represented as products of elementary reflectors:

$$U_1 = H_1 H_2 \ldots H_n \qquad \text{and} \qquad V_1 = G_1 G_2 \ldots G_{n-1}.$$

Each H_i and G_i has the form

$$H_i = I - \tau_{i,u} u_i u_i^T \qquad \text{and} \qquad G_i = I - \tau_{i,v} v_i v_i^T,$$

where $\tau_{i,u}$ and $\tau_{i,v}$ are scalars, and u_i and v_i are vectors. To block the computation one can observe the computation for step i - $H_i^T A_i G_i$, where A_i is the reduced matrix A before step i. So,

$$(I - \tau_{i,u} u_i u_i^T) A_i (I - \tau_{i,v} v_i v_i^T) = A_i - u_i y^T - x v_i^T$$

Here $x = \tau_{i,v} A_i v_i$, $z = \tau_{i,u} A_i^T u_i$, and $y = z - \tau_{i,u}(u_i^T x) v_i$. Note that it is possible to update only the current panel made of A_i's leading block of columns and rows in order to proceed with the computation and the application of the H_{i+1} and G_{i+1}. This is done by updating the remainder of the panel by x and y vectors. Thus, the update is delayed, but at each step, two matrix-vector products ($A_i v_i$ and $A_i^T u_i$) that require the access of the entire trailing matrix A_i are computed.

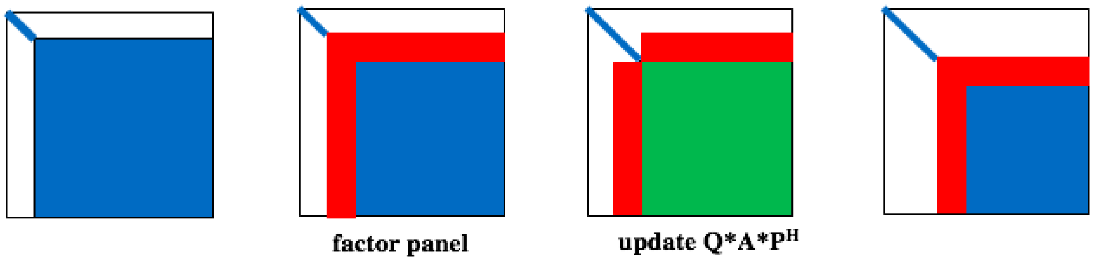


Figure 2.1: LAPACK one-stage blocked algorithm.

So, the LAPACK blocked algorithm has two computational steps: (1) the panel factorization and (2) the update of the trailing submatrix. First, the panel factorization processes a single block of columns and rows. The process annihilates columns/rows of a panel by Householder reflectors that introduce zeros to the entries below the subdiagonal. The corresponding left and right reflectors are saved in the original matrix A, and the accumulation of the left and right transformations are saved in two temporary storages X and Y. The accumulation requires two matrix-vector operations and thus loads the whole unreduced trailing matrix into memory at each step of the reduction algorithm. After factorizing the panel, the trailing matrix is updated by two matrix-matrix multiplications. One multiplication requires the left reflectors (V) and the accumulated transformations X, and the other multiplication uses the right reflectors (U) and the accumulated transformations Y. The process is repeated until the whole matrix is reduced to bidiagonal form.

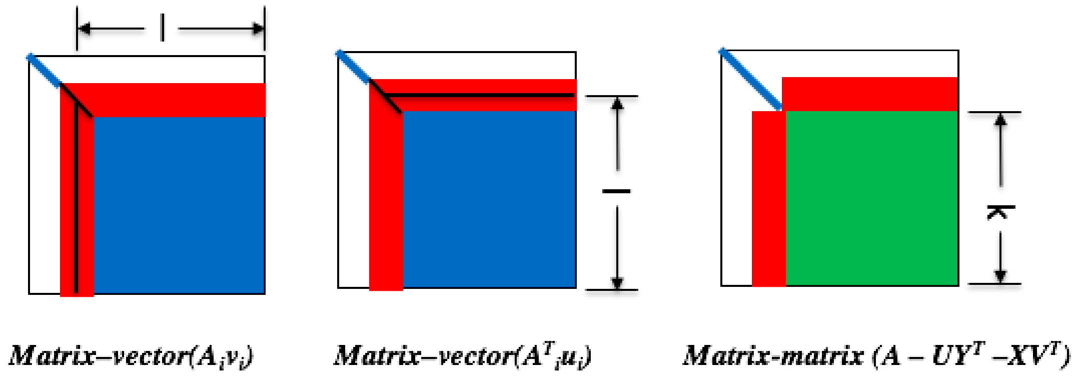


Figure 2.2: LAPACK one-stage blocked algorithm - used BLAS kernel.

In particular, for a square matrix of size n by n with a block size nb (for simplicity, nb divides n) there are n/nb steps in the algorithm. At each step, the algorithm needs $2 \times nb$ matrix-vector operations to accumulate X and Y. If l is the size of trailing matrix at step s, then the cost of this operation is $2l^2$. As there will be a $2 \times nb$ such operation, the total cost is $2nb \times 2l^2$. The update of the trailing matrix at step s is,

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

If k is the size of the trailing matrix at step s. The cost of this update is the cost of two matrix-matrix products using the gemm routine - $2 \times 2\ nb\ k^2$. Thus the total cost for the n/nb steps is:

$$\begin{aligned} &\approx\ 4nb \sum_{nb}^{n/nb} l^2 + 4nb \sum_{2nb}^{\frac{n-nb}{n_b}} k^2 \\ &\approx\ \tfrac{4}{3} n^3_{\mathsf{gemv}} + \tfrac{4}{3} n^3_{\mathsf{gemm}} \\ &\approx\ \tfrac{8}{3} n^3. \end{aligned}$$

So, for $m = n$, half of the operations are in Level 2 BLAS (matrix-vector products), while the other half are in Level 3 BLAS. In conclusion, the bidiagonal reduction based on blocked Householder transformations is expected to be about $2\times$ faster than a non blocked Level 2 BLAS factorization — provided the Level 3 BLAS is significantly faster than the Level 2 BLAS. This is the case for current accelerators and many-core processors, where the ratio of Level 3 to Level 2 BLAS performance is about 30×, and current trends show this ratio increasing for the foreseeable future.

The one-stage reduction to bidiagonal form described in section-2.2 has poor efficiency. The panel factorization, which introduces zeros to the entries below the sub-diagonal within a single block of columns, requires two matrix-vector multiplications with the trailing sub matrix for each reflector and is thus memory bound. This step is critical and time-consuming, as the whole trailing matrix is loaded into memory twice for each column/row of the matrix. The performance is bounded by memory bandwidth and does not scale up with the number of cores. Moreover, one may get tremendous amount of cache and TLB misses for large matrices, as the matrix will not fit in cache. The trailing sub matrix is then updated by the blocked reflectors using Level 3 BLAS — matrix-matrix multiplications (GEMM). This is the only computational step in one-stage reduction algorithms that is computation intensive and rich in parallelism. Unfortunately, update of the trailing sub matrix is synchronized with panel factorization, which prevents asynchronous execution of memory-bound and compute-bound steps. Figure 2.3 shows the percentage of the total time spent for each of the three phases of the SVD algorithm using the standard one-stage reduction approach when all the singular vectors are computed. From Figure 2.3 it is clear that the reduction to the bidiagonal form requires more than 70% of the total execution time when all the singular vectors are computed and consumes 90% of the total execution time when only singular values are computed.

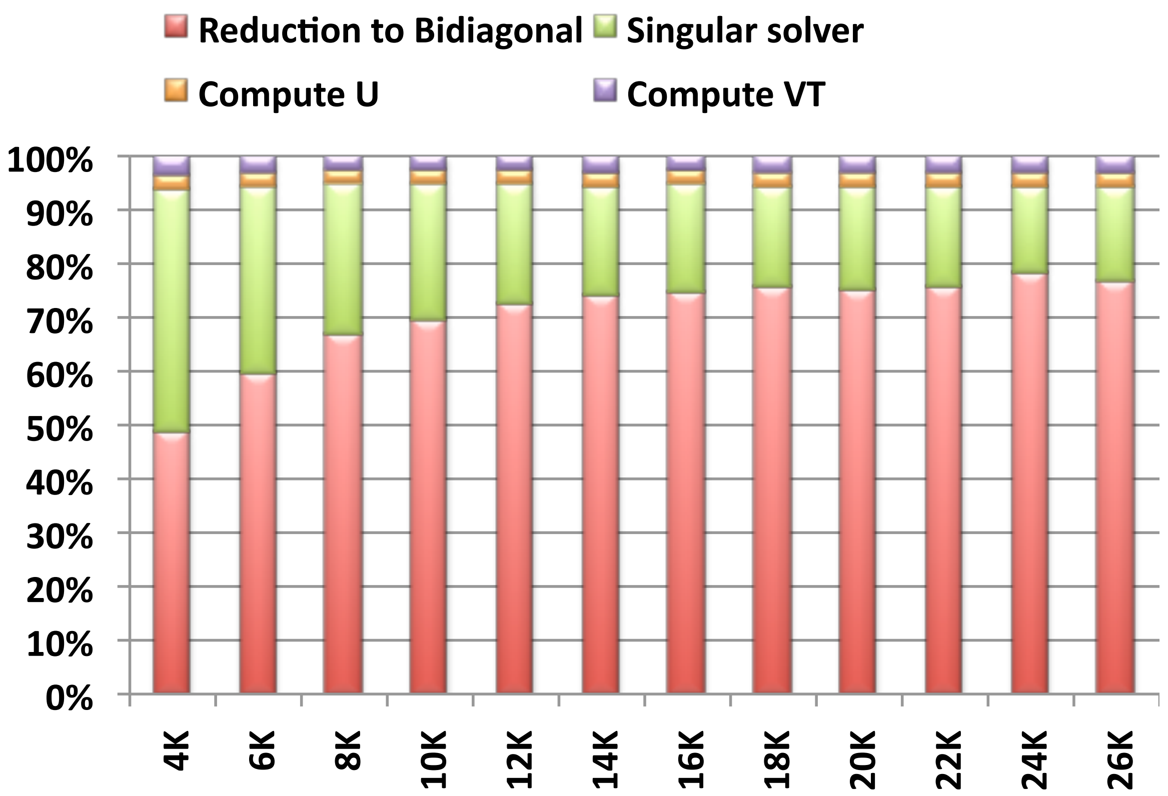


Figure 2.3: The percentage of the time spent in each kernel of the DGESDD solver using the standard one-stage approach to compute the bidiagonal form.

2.3 Two-stage bidiagonal reduction

The two-stage reduction is designed to overcome the limitations of the one-stage approach and reduces memory-bound operations. It now relies heavily on compute-intensive operations so that performance scales up with CPU core count. As the name implies, the two-stage approach splits the original one-stage approach into two phases — the first phase(first stage) reduces the general matrix to band form and the second phase (second stage) reduces the band matrix to bidiagonal form as shown in Figure 2.4. The first stage is compute intensive and heavily depends on Level 3 BLAS, whereas the second stage is memory bound and depends on Level 2 BLAS. The idea behind the first stage, which reduces the general matrix to band form, is based on the tile algorithm. Before moving to the details, we will talk about tile algorithms.

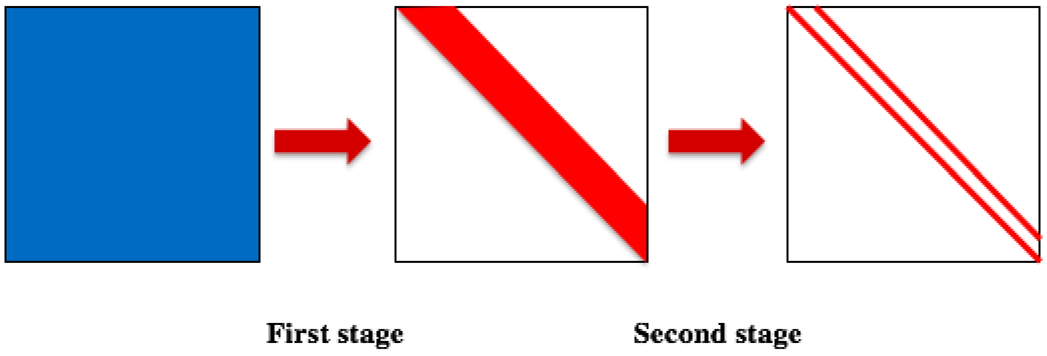


Figure 2.4: Two-stage bidiagonal reduction

2.3.1 Tile algorithms

Tile algorithms are based on the idea of dividing the entire matrix into square tiles of relatively small sizes and process the matrix tile by tile. The rationale is that the few tiles (one, two, or three) that are involved in a particular matrix operation fit entirely in some level of the cache hierarchy so that *capacity* cache misses can be mostly eliminated. The motivation for tile algorithms came from the desire to extend the performance benefit matrix multiplication gets by tiling the multiplication algorithm. The great advantage of tile algorithm is it allows the expression of the algorithm in the form of a task graph, *Direct Acyclic Graph* (DAG), that can be scheduled by dynamic scheduler using dataflow principles [10, 46, 35].

The benefits of tile algorithms on multicore processors were initially demonstrated for the one-sided factorization (Cholesky, LU and QR in [42, 11, 13, 45]) and later extended for bidiagonal and tridiagonal reduction algorithms for the solution of the singular value and the symmetric eigenvalue problem. The application of Householder transformations by tiles reduces the general matrix to band form, but successive elimination of the sub diagonal entries by a series of Householder transformations [48, 51, 33, 34, 50, 49] is required to generate the proper form.

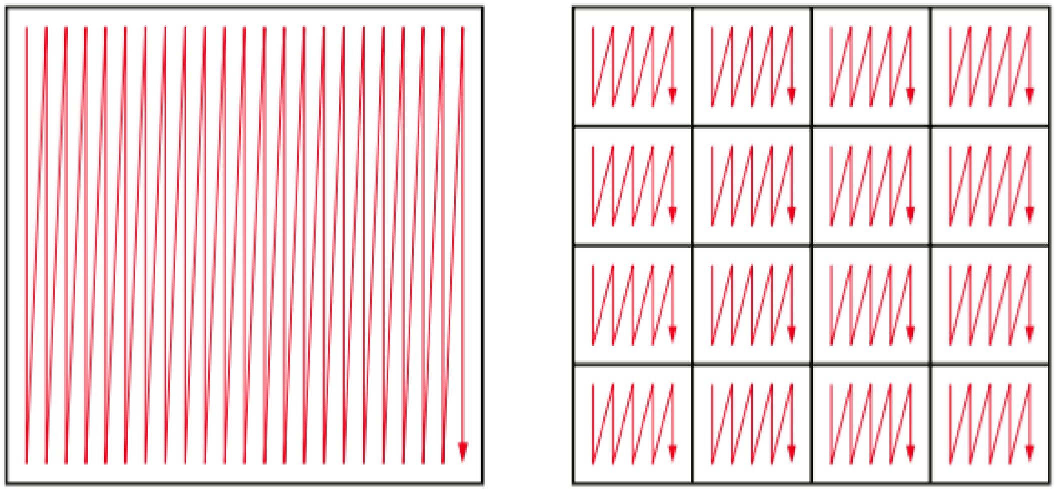


Figure 2.5: Translation from LAPACK layout (column major) to tile data layout.

The fundamentals behind tile algorithms are to transform the original matrix to tile data layout (TDL) [30] format as shown in Figure 2.5. The entire matrix is divided into square sub matrices called tiles. Each tile of the matrix resides in contiguous region of memory so that translation between tile layout and FORTRAN 77 layout can be done in place. Gustavson et al. [31] developed a collection of routines to translate FORTRAN 77 layout to tile layout and vice versa that are distributed as part of the PLASMA library[55].

The dramatic fine-grained parallelism exposed by tile algorithms can be exploited by designing an efficient scheduler that will maintain data dependencies while scheduling them in parallel fashion. This has been done for both one-sided factorizations in [11, 13, 2, 46, 1, 35, 17, 18, 19], as well as the more complicated two-sided ones in [48, 51, 33, 34, 50, 49]. But constructing such schedules by manipulating loop indexes and maintaining dependencies using progress tables are tedious and error prone. The *QUeuing And Runtime for Kernels* (QUARK) [56] system, developed at the University of Tennessee, overcomes these problems by constructing a virtual DAG, or task graph, of the problem and exploring the tasks in the order organized in DAG.

2.3.2 First stage — compute intensive

The first stage reduces general matrix to band form using a sequence of blocked Householder transformations. This stage eliminates matrix-vector operations from the one-stage and depends on matrix-matrix multiply kernels. That is why the first stage is compute intensive and can be run in parallel [7, 20, 23, 36]. Conceptually, the matrix is split into $nt \times nt$ tiles for a matrix of size $n \times n$ with tile size nb where $nt = n/nb$. As the tiles are small in size, the entire tile fits in cache and is stored contiguously in memory. The algorithm then proceeds as a collection of interdependent tasks that can be scheduled by both static and dynamic scheduler. In Algorithm-1 we have shown the tile algorithm for the reduction of general matrix to band form. Eight compute-intensive kernels are used in the algorithm.

– DGEQRT/DGELQT perform a QR factorization of diagonal tile and an LQ factorization of a sub or super diagonal tile, respectively.

– DORMQR/DORMLQ apply the orthogonal transformations computed from DGEQRT/DGELQT to the left/right sides, respectively.

– DTSQRT/DTSLQT compute a QR and an LQ factorization by coupling a triangular tile (upper if QR, lower if LQ) with a corresponding full square tile.

– DTSMQR/DTSMLQ apply the orthogonal transformations computed from DTSQRT/DTSLQT to the left/right sides, respectively, of the entire trailing matrix.

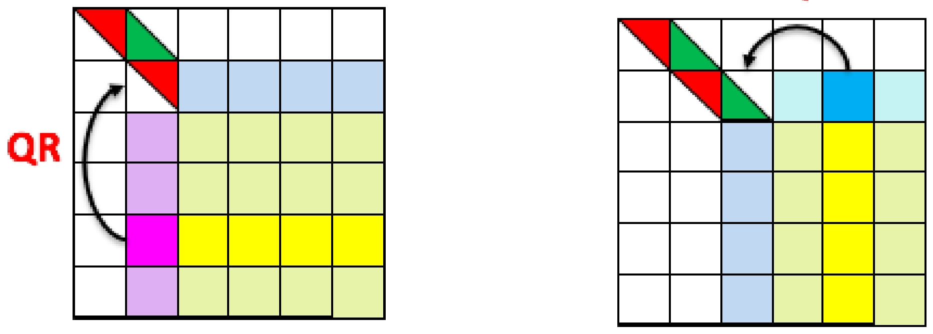
Algorithm 1: Tiled algorithm to reduce general matrix to band form

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

For a tile of size b and $n \times n$ tiled matrix, Table 2.1 below shows the computation cost for reducing general matrix to band form. In terms of flop count, $TSMQR$ routine is the most expensive kernel and consumes the most flops.

Table 2.1: Computation cost for kernels use to reduce a general matrix to band form

Kernel	Computation cost	Total cost for $n \times n$ tile matrix
GEQRT/GELQT	$2b^3$	$O(n) \times 2b^3$
ORMQR/ORMLQ	$3b^3$	$O(n^2) \times 3b^3$
TSQRT/TSLQT	$\frac{10}{3}b^3$	$O(n^2) \times \frac{10}{3}b^3$
TSMQR/TSMLQ	$5b^3$	$O(n^3) \times 5b^3$



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

Figure 2.6: Kernel execution of the BRD algorithm during the first stage.

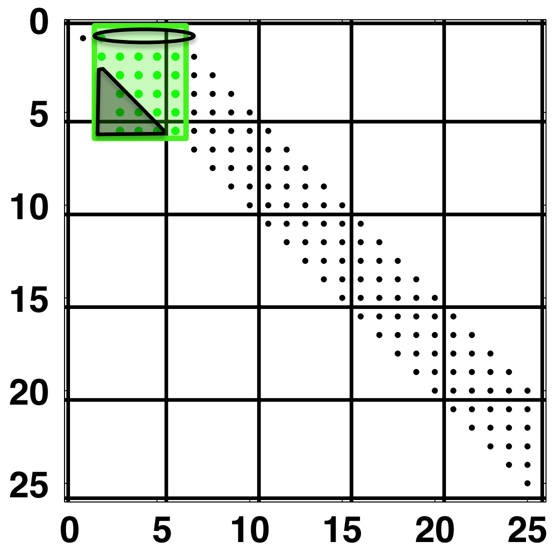
Figure 2.6 shows the execution foot print for the second step of the first stage of reduction algorithm. A QR factorization is computed for the tile $A_{2,2}$ (the red tile). When this QR factorization is finished, all the tiles right to $A_(2,2)$ are updated in parallel. Update of $A_{2,\bullet}$ (the grey tiles of Figure 2.6a) are performed by applying the Householder transformations that are generated by the QR factorization of $A_{2,2}$. Simultaneously, all the tiles $A_{\bullet,2}$ (the magenta tiles of Figure 2.6a) can also be factorized independently one after another, as all of them require the use of R of $A_{2,2}$. After the QR factorization of tile $A_{i,2}$ (the dark magenta tile of Figure 2.6a), all the tiles of the block row i (the dark yellow tiles of Figure 2.6a) are updated by a set of parallel tasks. Update of tiles from row i also requires to access tiles from the second row. Moreover, when $A_{2,3}$ is updated, LQ factorization can now proceed for this tile (the green tile of Figure 2.6b). Just like the QR process, after LQ factorization, all the tiles in the third column ($A_{3:nt,3}$ [the grey tiles of Figure 2.6b]) are now independently updated by the Householder vectors computed during LQ factorization, provided that updates of these tiles are done for QR factorization. Similarly, all the tiles ($A_{2,4:nt}$ [the light cyan tiles of Figure 2.6b]) can also be factorized, and annihilation of $A_{1,i}$ (the dark blue tile of Figure 2.6b) enables update of the tiles from block column i (the dark yellow tiles of Figure 2.6b).

The interleaving of QR and LQ factorization at each step as explained above for the execution flow repeats until the end of the algorithm. At the end, it generates a band matrix of band size nb. It must be noted that the tile formulation of the algorithm creates many small tasks that can be executed in parallel. Usually the tasks tasks are organized into a DAG [9, 14] where the nodes represent the computational tasks and the edges represent the data dependencies among them. The tasks are then executed in parallel without violating their dependencies. Such restructuring of the algorithm as a sequence of tasks that operate on tiles of data removes the fork-join bottleneck of LAPACK and increases the overall performance efficiency.

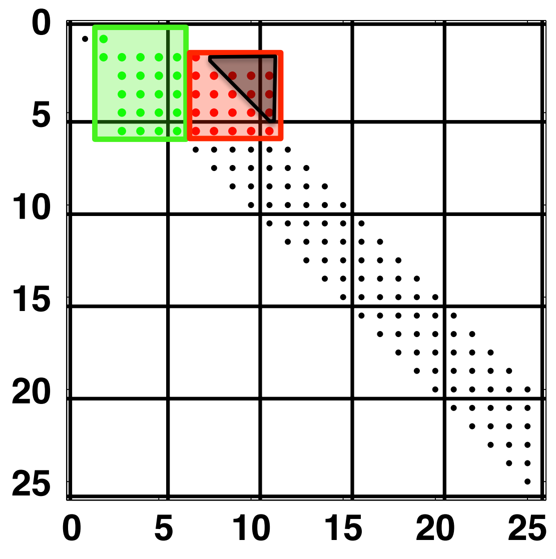
2.3.3 Second stage

In the second stage, the band form is further reduced to bidiagonal form using the bulge chasing technique. This procedure chases the fill-in elements created during the annihilation process of the extra off-diagonal element and annihilates them using orthogonal transformation at each sweep of the algorithm. This step is memory bound and accesses the band matrix from multiple disjoint locations, which creates substantial latency overhead as different portions of the matrix are loaded into the cache. Unfortunately, there is too little computation to overcome this latency overhead. A novel bulge chasing algorithm described in [32] overcomes these critical limitations. The bulge chasing technique is similar to the one used for symmetric eigenvalue problems in [33] but differs from it in using a column-wise elimination instead of an element-wise elimination. When singular vectors are computed a column-wise elimination method has great advantage over element-wise elimination. In particular, singular vectors are updated by element-wise Householder reflectors based on BLAS 1 operations, which presents a serious bottleneck for performance improvement. On the other hand, a column-wise elimination method accumulates the transformations and updates the singular vectors using Level 3 BLAS. As a result, the update is faster and more efficient.

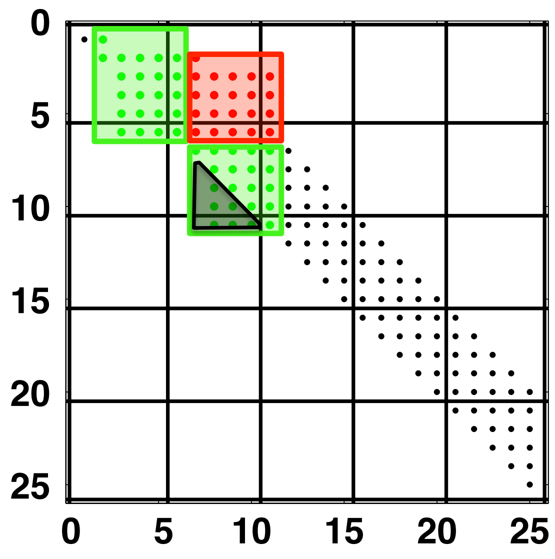
The bulge chasing algorithm has three cache-efficient kernels. The main goal of the kernel is to load the block of the matrix in cache and apply all the possible computations before being replaced by another block. The first kernel is called xGBCW1, which manipulates the green block of data as shown in Figure 2.7a. It annihilates the extra non zero entries within a single row and applies the computed elementary Householder reflector from the right. This annihilation process triggers a new bulge (triangular bulges as shown in Figure 2.7a [the black block]) that is chased in a subsequent sweep. One can notice that a bulge (lower triangular portion of the green block in Figure 2.7d) created in one sweep overlaps with the bulge (lower triangular portion of the blue block in Figure 2.7d) created in the next sweep. Instead of eliminating the whole triangular bulge elimination of the overlapped portion is delayed for later sweep and the non overlapped portion is eliminated in the current sweep. The second kernel is xGBCW2, which loads the next block and applies the necessary left updates derived from the first kernel. It also generates triangular bulges as shown in Figure 2.7b. Finally, the third kernel is xGBCW3. It loads the next block (the third green block of Figure 2.7c) and applies the right updates derived from kernel 2. Like kernel 1, kernel 3 generates a bulge that is removed and updated correspondingly from the left. So the single sweep of the bulge chasing process can be described as a single call of kernel 1 followed by repetitive call to a cycle of kernel 2 and kernel 3.



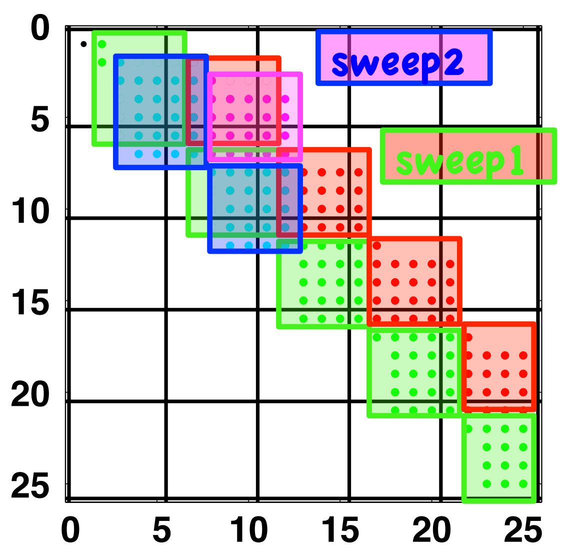
(a) xGBCW1 (green).



(b) xGBCW2 (red).



(c) xGBCW3 (green).



(d) bulge overlap.

Figure 2.7: Kernel execution of the BRD algorithm during the second stage.

The main challenge for this stage is to track dependencies among the tasks. Tasks from one sweep use partial data from the previous sweep. Dependencies among the computation tasks from subsequent sweeps are tracked using the data translation layer (DTL) and functional dependencies described in [51, 33]. To reduce memory traffic, the same thread is assigned the subsequent tasks that involve the same region of data. Scheduler ensures maximum reuse of data by distributing the tasks according to their data location.

2.4 System and disk information

We have used a few different systems to run our experiment. In Table 2.2 we have shown the details of the machine we used.

Table 2.2: Machine configuration

	Sandy Bridge Xeon 5-2670, Western Digital	Haswell i7-5930K, Samsung SSD	Haswell Xeon E5 2650V3, Seagate Constellation
Clock	2.6 GHz	3.5 GHz	2.3 GHz
Core	16	6	10
Memory	52 GB	32GB	32GB
Cache	20 MB	15 MB	25 MB
Peak performance	330 Gflop/s	336 Gflop/s	368 - 480 Gflops (with boost)
Disk	Western Digital WDC1002FAEX 931G	Samsung SSD EVO 465G	Seagate Constellation ES.3 1000G

We have also considered the following accelerators (GPU) and coprocessors for our theoretical analysis.

Table 2.3: Accelerator and coprocessor

	NVIDIA K40, PCIe 8x	NVIDIA P100, PCIe 8x	Xeon Phi KNC, PCIe 8x	Xeon Phi KNL, PCIe 8x
Clock	745 MHz	3.5 GHz	1.2 GHz	1.30 GHz
Core	15(SMX)	56(SMX)	61	64
Memory	12 GB	16GB	16 GB	16 GB
L2 Cache	1536 KB	4096 KB	30.5 MB	32 MB
Peak performance	1430 Gflop/s	5300 Gflop/s	1208 Gflop/s	3000 Gflop/s

Hard disk drives (HDD) have been used for data storage in high-performance systems for decade. Recently, the flash-memory-based Solid State Drive (SSD) has become an emerging technology and started to gain prominence for faster read access, low power consumption, small size, and reliability compared with hard disks. That's why we consider both HDD and SDD for the experiment and theoretical analysis of our OOM SVD solver. Table 2.4 shows detailed information about the disks we used for our experiment. As disk bandwidth is extremely important to design and implement OOM algorithms, we have benchmarked HDD/SDD's sequential read/write bandwidth using both the dd and hdparm utilities from Linux. We have also benchmarked random read/write bandwidth of the disks. To do that, we accessed random tiles of a $u \times v$ tile matrix residing in the disk. We generated a random number, r, between 1 and $u * v$ and accessed ($r \bmod u$, $\frac{r}{u}$) tile of the matrix for read/write. In Table 2.4 we have shown the bandwidth we are supposed to achieve for both sequential and random disk access to the disk.

Table 2.4: Disk bandwidth information

	WDC1002FAEX	Samsung SSD EVO	Seagate Constellation ES.3 ST1000NM0033
Size	931B	465G	1000G
Peak sequential read/write bandwidth	150 MB/s	540 MB/s	175 MB/s
Sequential read/write bandwidth	50 MB/s	450-470MB/s	150 MB/s
Random read bandwidth	12 MB/s	200 MB/s	70 MB/s
Random write bandwidth	12 MB/s	90 MB/s	70 MB/s

Chapter 3

OOM SVD

The SVD for a $m \times n$ 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 matrix U and V contains the left and right singular vectors of A. As described above, SVD is solved 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: then the singular value solver computes the singular values Σ, and the left and right singular vectors $\widetilde{U}$ and $\widetilde{V}^T$ of the bidiagonal matrix B. 3) Singular vector update phase: if required, the left and the right singular vectors of A are computed by multiplying $\widetilde{U}$ and $\widetilde{V}^T$ by the orthogonal matrices Q and P used to reduce the general matrix to bidiagonal form in the reduction phase. In this work, we are interested in the computation of the singular value only.

3.1 Introduction

When the matrix A is too large and does not fit into the system memory, we have to find a technique to perform the computation while A is out of memory (A could be in the Hard Drive disk, flash memory, fast buffer, CPU memory when the GPU or the XeonPhi is considered to be the system etc...), that's what we call OOM algorithm. The bottleneck of the SVD computation is the first phase where we have to reduce the dense matrix A to bidiagonal form. Once it is bidiagonal, it consists of two vectors, and thus it will fit into the memory and the singular value solver will be able to compute its singular value in memory. If the singular vectors are needed, they will require an OOM technique, but this case is not studied here, we focus on the computation of the singular values. As a consequence, the main focus should be on the reduction phase. To reduce a general matrix to bidiagonal form we can use either the standard approach, which is implemented in LAPACK (we call it a one-stage algorithm since it reduces the matrix from dense to bidiagonal in 1 step), or the two-stage algorithm implemented in PLASMA, which reduces the matrix in two steps, first to band form then to bidiagonal form.

Since, A reside out of memory, communication between disk and memory, and bandwidth of the disk will have high impact on the overall run time of any OOM algorithm. Thus, a careful understanding and study of the computational process and the communication pattern is required in order to propose a successful and optimized design. Below, we will explain the details of each algorithm, as well as evaluate and prove the optimal design in order to implement it in an OOM fashion.

3.2 An analytical study of the communication cost of data movement

In this section we studied the communication pattern for the OOM reduction to bidiagonal form. We will provide analysis for the two techniques (one-stage vs, two-stage) and propose and discuss our design decision that minimizes the communication cost.

As described in section 2.2 and detailed in Equation (2.2), the one-stage bidiagonal reduction needs two matrix-vector operation (GEMV) with the trailing matrix at every column/row annihilation and one matrix-matrix operation (GEMM) after every panel reduction. Thus, when the matrix is large and does not fit into the main memory, it will need to be loaded from the disk two times for every column/row annihilation for the GEMV operation and two times after each nb column (e.g., after each panel) for the GEMM operation. In every case the matrix is sent back to disk. The algorithm will requires $2(m \times nb + n \times nb)$ as in memory workspace to hold the panel (U and V) and the arrays X and Y of Equation (2.2). Therefore, for a $m \times n$ matrix the amount of words to be read and written (e.g., the amount of data movement) is given by the following formula:

$$\begin{aligned}
&\text{Read } A \text{ for dgemv } 1 + \text{Read } A \text{ for dgemv } 2 + \text{Read/Write } A \text{ for dgemm} \\
&= \sum_{s=0}^{n-1}(m-s)(n-s) + \sum_{s=0}^{n-1}(m-s)(n-s-1) + 2\sum_{s=1}^{n/nb}(m - s\times nb)(n - s \times nb) \\
&= (2\sum_{s=0}^{n-1}(m-s)(n-s) - \sum_{s=0}^{n-1}(m-s)) + 2\sum_{s=1}^{n/nb}(m - s\times nb)(n - s \times nb) \\
&= mn^2 - \frac{n^3}{3} + \frac{n^2}{2} - m + \frac{5n}{6} - 1 + \frac{mn^2}{nb} - mn - \frac{n^3}{3nb} + \frac{n}{3} \times nb
\end{aligned}$$

For a $m \times m$ matrix, the amount of word movement is:

$$\begin{aligned}
&\frac{2}{3}m^3 + \frac{m^2}{2} - \frac{m}{6} - 1 + \frac{2m^3}{3nb} - m^2 + \frac{m}{3} \times nb \\
&\approx \frac{2}{3}m^3 + \frac{1}{nb} \times \frac{2}{3}m^3
\end{aligned}$$

On the other hand, PLASMA uses a two-stage approach: (1) In the first stage it reduces the general $m \times n$ matrix to a band form of size $\min(m,n) \times nb$. (2) In the second stage, it reduce the band to bidiagonal form. Note that for a small nb the whole band matrix of size $min(m,n) \times nb$ will fit into the memory and thus the second stage can run efficiently in memory. Thus the first stage (e.g., reduction from dense to band) need to be performed in OOM fashion. As a result, for a $m \times n$ matrix and band size nb, the amount of data movement is given by:

$$\begin{aligned}
&\text{Read/Write } A \text{ for QR} + \text{Read/Write } A \text{ for LQ} \\
&= 2 \times \sum_{s=0}^{n/nb-1} (m - s \times nb)(n - s \times nb) \\
&+ 2 \times \sum_{s=0}^{n/nb-1} (m - s \times nb)[n - (s+1) \times nb] \\
&= 2nb^2 \times (\frac{mn^2}{nb^3} - \frac{n^3}{3nb^3} + \frac{n^2}{2nb^2} - \frac{m}{nb} + \frac{5n}{6nb} - 1) \\
&\approx \frac{2}{nb}(mn^2 - \frac{n^3}{3})
\end{aligned}$$

For $m \times m$ matrix amount of data movement is given by:

$$\begin{aligned}
&\frac{2}{nb} \times \frac{2}{3}m^3 \\
&= \frac{1}{nb} \times \frac{4}{3}m^3
\end{aligned}$$

From this formulation, one can easily observe that the classical one-stage algorithm for the reduction to bidiagonal requires $O(m^3)$ more word transfer between the system memory and the disk than two-stage approach. This is a huge amount of extra communications that will dramatically affect the performance. To highlight the importance of the communications, let's start by giving an example: for a matrix of size $m = 100,000$, the classical one-stage algorithm will need $\frac{2}{3}m^3 + \frac{1}{nb} \times \frac{2}{3}m^3$ words movement. In double-precision arithmetic, for recent hardware such as Hard Drive, Solid State Drives (SSD), or out of GPU memory where the communication bandwidth is about 150 MB/s, 500 MB/s, and 8 GB/s, respectively, the standard one-stage technique will require 411, 123, and 7.72 days, respectively to perform the reduction. The two-stage technique will need approximatively $\frac{1}{nb} \times \frac{4}{3}m^3$ words movement and thus in double precision it necessitates 5.14, 1.54 and 0.09 days, respectively, for nb equal 160. We mention that the one-stage approach requires the communication of $\frac{2}{3}10^{15}$ extra words, and for that we can easily expect this huge difference.

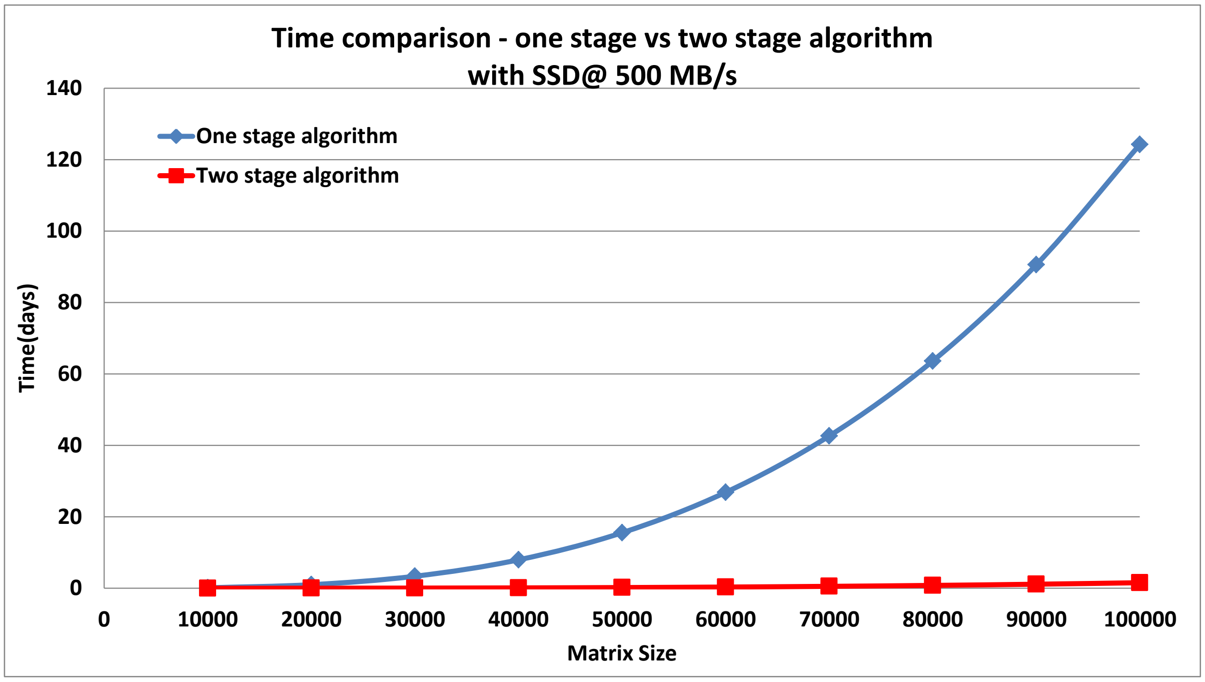


Figure 3.1: Time comparison between one-stage and two-stage algorithms.

As consequence, it is unacceptable to propose the one stage as an OOM algorithm. For that reason as well, it has always been known not to be practically possible to have an OOM SVD implementation. Moreover, to emphasize the choice of the two-stage approach, let's consider that the matrix will fit into the main memory. Then a one-stage approach will require approximately $\frac{2}{3}m^3 + \frac{1}{nb} \times \frac{2}{3}m^3$ words movement 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 60GB/s, about 24.71 hours are necessary to finish the reduction to bidiagonal form in double-precision arithmetic, while the two-stage algorithm needs approximately $\frac{1}{nb} \times \frac{4}{3}m^3$ words movement and thus requires about 0.31 hours for nb equal 160. If the matrix is read from SSD or HDD, more time is needed as compared with data read from memory because of poor bandwidth. In Figure 3.1 below we have compared the time required to reduce a general matrix to bidiagonal form between one-stage and two-stage algorithms for different matrix sizes when the matrix reside in SSD. For example, for a 100000×100000 matrix one stage will take 124.22 days compared with 1.54 days by a two-stage algorithm.

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

In this section, we will proceed with the theoretical analysis of the OOM algorithm and we will provide a detailed study of the communication pattern required by the OOM algorithm, as well as discuss and propose design strategies proving its optimality in terms of data movement and performance. In this work, we decided to comply with the proof of the previous section, which states that the only possible path for an OOM SVD solver is 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. An OOM algorithm, mean that the data on which the computation should happen is out of the main memory (e.g., either on disk, fast buffer, or out of the device memory in the case of when we consider the GPU as the main memory) and thus need to be loaded into the main memory by block, performing some computation and then sent back in order to allow another block to be loaded. For simplicity of description, our terms will follow the well-known historical OOM description where the matrix is on disk (OOM storage) and the CPU DRAM is considered to be the main memory. However, the formulation and theorem proved here can be applied to any OOM design, such as when the CPU DRAM is the main memory and the fast buffer is the OOM storage, or when a GPU/Xeon Phi is considered as the main memory and the CPU is the OOM. That is why the overall performance of the OOM reduction of general matrix to band form depends on the efficiency of minimizing or possibly hiding the communication overhead between the disk and the main memory. The widely used linear solver consisting of either cholesky, LU, or QR factorization can be implemented in a left-looking fashion, which means that data can be modified only once during factorization, and thus we can consider overlapping communication with computation. In contrast, we will see below that this is not possible for eigenvalue and singular value solver since they involve a two-sided process that modifies all the data of the trailing matrix at every step of the reduction.

3.3.1 A study of the communication/computation ratio

We will study and formulate theorem to answer one main question for any OOM algorithm, which is: in what circumstances, if there is any, we can hide communication overhead? and what is it's impact on the design of an out-of-memory algorithm? The idea here is to analyze the possibility of hiding the data transfer with the computations. To hide the communication overhead, the technique is that, if the computation is happening on data of block k, we need to write back the data of block $k-1$ and read the data of block $k+1$ in less or equal time to the computation task on the data of block k. As the two-stage algorithm works on tiles [32], our consideration is what tile size can be used in order to hide communication overhead between disk and memory. The main and the most time-consuming type of task of the two-stage algorithm is the update task (e.g., the TSMQR). Let's focus the description on this type and the substitution to other type will be implicitly easily derived. Figure 3.2 shows two scenarios for the TSMQR tasks: (1) All the threads are participating in the computation of a single task — call it multi-threaded single task. To hide communication, we have to 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). (2) Each thread works on a separate tile — sequential multi-task. If there are p threads, we have to 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.

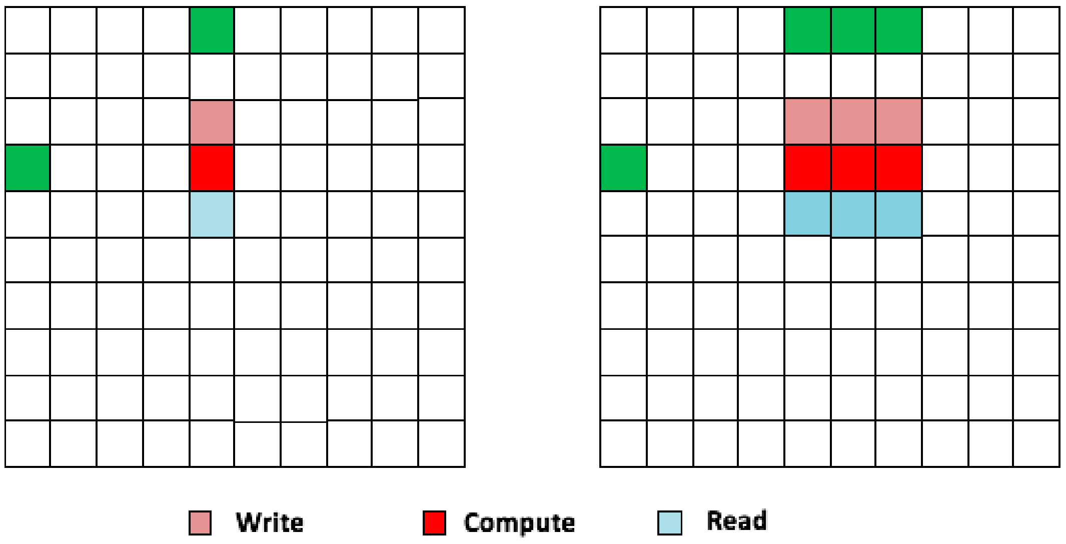


Figure 3.2: Reduction of general matrix to band form — update (multithreaded single task vs single-threaded multitask)

Theorem 3.1. *For the OOM SVD two-stage reduction algorithm, in order to overlap data communication with computation, the tile size b should be at least $\frac{3.2\alpha}{BW}$, .i.e. $b \geq \frac{3.2\alpha}{BW}$, where BW is the communication bandwidth and α is the computational performance efficiency of the system.*

Proof. First, let's consider the case when all the threads are working on a single task as shown in Figure 3.2 (left). A tile of size b consists b^2 elements, $8b^2$ bytes in double precision arithmetic. We will use the DP arithmetic representation for all the subsequent formulations. Assuming that the write bandwidth is similar to the read one, the time to read, t_{read}, or to write, t_{write}, a tile of size b is given by:

$$t_{read} = t_{write} = \frac{8b^2}{BW}s$$

where BW is the bandwidth of the transfer between disk and memory. The computation cost, which is the update cost (the *TSMQR* routine), for a tile of size b

is $5b^3$ flops. The time to compute, $t_{compute}$, is given by,

$$t_{compute} = t_{update} = \frac{5b^3}{\alpha}$$

where α is the performance efficiency in flops of the operation that has to be performed (the TSMQR is the case that reaches about 80%-85% of the machine peak). To hide the communication overhead, the necessary condition is as follows:

$$\begin{aligned} & t_{compute} \geq t_{read} + t_{write} \\ & => \frac{5b^3}{\alpha} \geq \frac{16b^2}{BW} \\ & => b \geq \frac{3.2\alpha}{BW} \end{aligned}$$

Now consider the case where tasks are running in parallel (Figure 3.2 [right]) and each thread is working on a separate tile. If p tasks run in parallel, p tiles are brought to memory and sent back to disk after computation. So,

$$t_{read} = t_{write} = \frac{p \times 8b^2}{BW} s$$

The time for computation, $t_{compute}$ is given by

$$t_{compute} = \frac{5b^3}{\frac{\alpha}{p}} = \frac{p \times 5b^3}{\alpha}$$

To overlap computation with read/write,

$$\begin{aligned} & t_{compute} \geq t_{read} + t_{write} \\ & => \frac{p \times 5b^3}{\alpha} \geq \frac{p \times 16b^2}{BW} \\ & => b \geq \frac{3.2\alpha}{BW} \end{aligned}$$

□

Table 3.1: Tile size for hiding communication time by computation for an OOM SVD solver

System	Communication bandwidth BW (GB/s)	DGEMM performance (Gflop/s)	Update kernel performance (Gflop/s)	Minimum tile size to hide communication
Sandy Bridge Xeon E5-2670 WDC1002FAEX	0.05	300	250	16000
Haswell i7-5930K Samsung SSD EVO	0.5	280	200	1280
Haswell Xeon E5 2650V3 Seagate Constellation ES.3 ST1000NM0033	0.15	440	300	6400
Tesla K40 PCIe 8x	8	1200	960	384
Tesla P100 PCIe 8x	8	4700	3760	1504
KNC 7120P PCIe 8x	8	960	768	308
KNL 7290 PCIe 8x	8	2000	1600	640

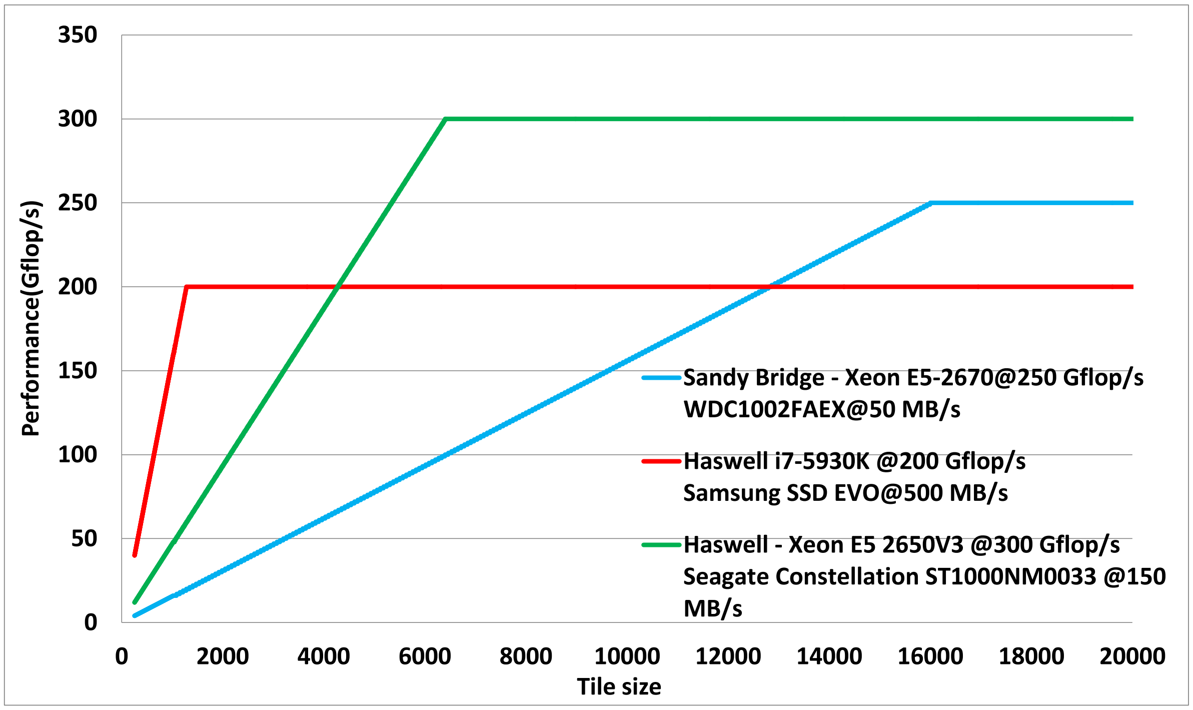
Table 3.1 shows the minimum tile size, "b", required to completely hide the communication overhead with the computation time for the system we outline in Table 2.2. The higher the ratio of computation, the larger the required tile size to overcome the communication time. For example, a Sandy Bridge machine having a computational performance $\alpha = 250$ Gflop/s connected to an HDD with a bandwidth of 50 MB/s requires the tile to be of size 16000. Such a big tile size is not reasonable because of the following:

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

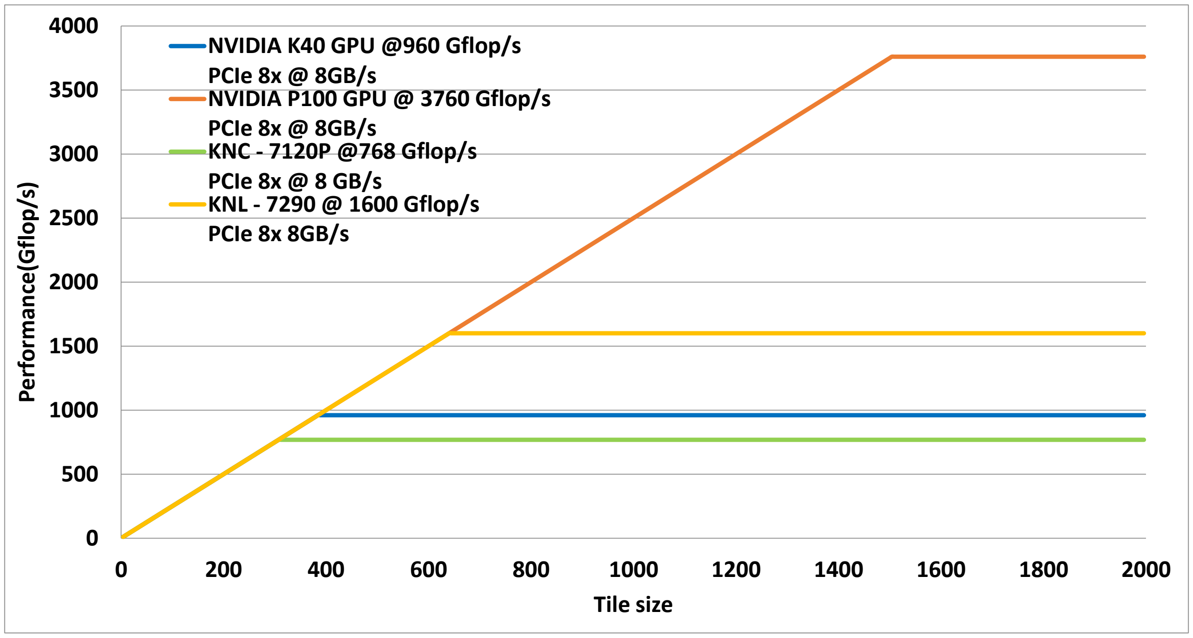
- Even if the band matrix fits in memory, the second stage (reduction from band to bidiagonal form) of the algorithm will be extremely inefficient and adversely affect the overall run time.

Performance of two-stage OOM SVD can be estimated by the roofline model of the $TSMQR$ routine, assuming the tile is read directly from and written back to disk. For double-precision data the $DTSMQR$ routine computes $5b^3$ flops for a tile of size b, and communicates $16b^2$ bytes of data for read and write. In short, the $DTSMQR$ routine computes $5b^3$ flop for $16b^2$ byte data. The arithmetic intensity (i.e. the flop-to-byte ratio for the $DTSMQR$ routine is $\frac{5b}{16}$. If the system has bandwidth BW, performance of two-stage OOM SVD is computed by multiplying arithmetic intensity by system bandwidth (i.e., $\frac{5b \times BW}{16}$). Figure 3.3 shows the performance of an OOM SVD solver for a different tile size when tile is accessed directly from the disk.

Figure 3.3 shows peak performance is not achievable with a small tile size. At the same time, the big tile size that is required to reach peak performance is not affordable. So, it can be concluded that performance of an OOM two-stage algorithm will be bounded by disk bandwidth if data is accessed directly from the disk.



(a) Performance roofline model for the SVD computation when the CPU is considered as the main memory and the data resides in the disk.



(b) Performance roofline model for the SVD computation when the Device (GPU/Xeon Phi) is considered as the main memory and the data resides in system DRAM memory.

Figure 3.3: Achievable performance for an OOM SVD solver.

3.3.2 A study to utilize main memory to hide communication overhead

The entire matrix may be too large to fit in memory, but some tiles definitely fit in there. When tiles are in memory, execution is faster than reading from disk. Some tiles might be loaded into memory at the beginning of the algorithm and other tiles are communicated back and forth between memory and disk as shown in Figure 3.4. We want to study whether data movement time for green tiles (in Figure 3.4 can be hidden by computation for both green and read tiles and in what circumstances it will be feasible.

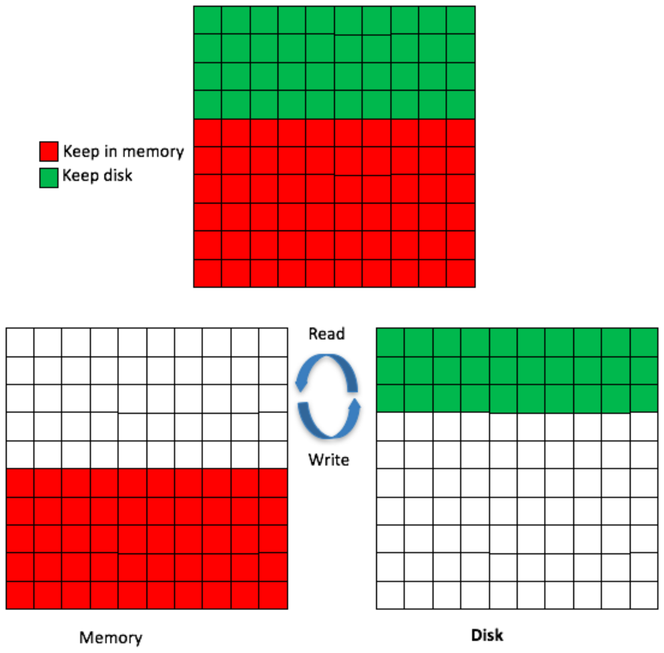


Figure 3.4: Dividing the matrix into memory and disk to hide communication overhead

Theorem 3.2. *For the OOM SVD two-stage reduction algorithm, in order to overlap data communication with computation, the ratio of tiles in the disk, nt_1, to the tiles in memory, nt_2, should be $\frac{1}{\frac{3.2\alpha}{b\times BW}-1}$, where nt, $nt = nt_1 + nt_2$, size of the matrix in number of tiles for tile size b, BW is the communication bandwidth and α is the computational performance efficiency of the system.*

Proof. Time for computation, $t_{compute}$ is:

$$\begin{aligned} t_{compute} &= computation\ for\ nt_1\ tiles\ +\ computation\ for\ nt_2\ tilse \\ &= \frac{nt_1 \times 5b^3 + nt_2 \times 5b^3}{\alpha} \\ &= \frac{nt \times 5b^3}{\alpha} \end{aligned}$$

nt_1 tiles are communicated between disk and main memory. Time for communication, $t_{read+write}$ is:

$$t_{read+write} = \frac{nt_1 \times 16b^2}{BW}$$

To hide communication overhead:

$$\begin{aligned} t_{read+write} &= t_{compute} \\ => \frac{nt_1 \times 16b^2}{BW} &= \frac{nt \times 5b^3}{\alpha} \\ => nt_1 &= \frac{b \times BW \times nt}{3.2\alpha} \end{aligned}$$

And,

$$\begin{aligned} nt_2 &= nt - nt_1 \\ => nt_2 &= \frac{3.2\alpha - b \times BW}{3.2\alpha} \times nt \end{aligned}$$

So,

$$\frac{nt_1}{nt_2} = \frac{1}{\frac{3.2\alpha}{b \times BW} - 1}$$

□

If $\alpha \gg BW$, $\frac{3.2\alpha}{b \times BW} \gg 1$ always for small tile size. For a Haswell E5 2650 machine having 150MB/s HDD bandwidth, ratio of tiles in disk to tiles in memory, $\frac{nt_1}{nt_2} = \frac{1}{49}$ for tile size 128, to hide data communication with computation. That means 98% of the matrix must be in memory to hide data movement cost for the tiles that are in disk. For big matrices, 98% of the matrix might be too large to fit in memory. So, it might not be possible to hide data movement cost completely. Holding tiles in memory will help to overlap some portion of the communication time with execution time of the tiles in memory but not completely.

Theorem 3.3. *For the OOM SVD two-stage reduction algorithm, computation of nt_2 tiles in memory overlaps communication of $nt_2 \times \dfrac{b \times BW}{3.2\alpha}$ tiles back and forth between CPU memory and disk, where b is the tile size, BW is the communication bandwidth and α is the computational performance efficiency of the system.*

Proof. Computation time for nt_2 tiles, $t_{compute}$ is:

$$t_{compute} = \frac{nt_2 \times 5b^3}{\alpha}$$

Time to read and write, $t_{read+write}$, of a tile of size b is:

$$t_{read+write} = \frac{16b^2}{BW}$$

So, number of tiles that can be brought to memory and sent back to disk:

$$\begin{aligned} &\frac{t_{compute}}{t_{read+write}} \\ &= \frac{\frac{nt_2 \times 5b^3}{\alpha}}{\frac{16b^2}{BW}} \\ &= nt_2 \times \frac{b \times BW}{3.2\alpha} \end{aligned}$$

□

For the Haswell E5 2650 machine, the number of tiles that can be brought to memory and sent back to disk is $nt_2 \times \frac{1}{50}$ for tile size 128. That means we can only read and write one tile while computing on 500 tiles. If we have 10000 tiles for a matrix and keep 1000 tiles in memory and 9000 tiles in disk then, while computing on 1000 tiles we can read and write only 20 tiles. For the rest of the 8880 tiles we have to pay the cost of reading and writing.

From theoretical analysis we can conclude that,

1. An OOM two-stage reduction algorithm requires a big tile size to hide the communication cost with the computation completely. Such big tile size is not possible to use because tile size defines the band of the reduced matrix. Performance of the second stage (reduction of band matrix to bidiagonal form) heavily depends on band size, which is b in this case, because of its memory-bound operation and potential to negatively affects overall performance.

2. Computation on tiles loaded into memory hides a very small portion of communication cost and the streaming of tiles for subsequent computation is not possible. Thus, the overall performance is bounded by the amount of reading and writing of tiles from the disk.

3.4 Algorithmic design

From section 3.3 we know that we can hide a very small portion of the communication cost by the computation. The OOM linear solver algorithms (such as Cholesky, LU factorization, and QR decomposition) involve on-sided factorization and are implemented in an OOM left-looking fashion, which will allow the modification of only block of data at each step of the process. Thus, their communication can be overlapped with the computation. In contrast, the reduction algorithm is from the two-sided factorization family (of which the tridiagonal, bidiagonal, and Hessenberg reductions are members). The reduction algorithms need to modify all the data of the trailing matrix at every step of the process, meaning they are bound by the number of reading and writing of tiles from the disk. So, in this section, our goal is to analyze and study the possibility of optimizing amount reading and writing, which in turn will reduce the communication overhead. If main memory can not hold more than four tiles, then no optimization is possible. The algorithm needs to use a maximum of four tiles or thus the data will fly back and forth from the disk during execution. If enough space exists for more than four tiles — which is the practical and realistic case, since it is unrealistic to expose a system that has only space for four tiles[only about 1.2MB of memory] then careful attention to the design is required to reach an optimal solution time.

As we are reading and writing data in tile granularity, our algorithm design determines which tiles are used most and holds them in memory until they are not used any more. The number of times tiles are requested in the reduction process depends on the order they are accessed and processed. For example we can process the algorithm in row-wise or column-wise data flow fashion. This can be viewed as something similar to the left and right-looking process used in one-sided factorization.

3.4.1 Proposition 1 — imposing parallel data flow

Our first algorithmic design follows a data flow fashion that increases the number of parallel tasks by prioritizing parallel task flow to locality. Algorithm 2 gives the details of its implementation, and Figure 3.5 illustrates the fingerprint of the dataflow pattern during one step. The reduction process consists of a QR sweep followed by an LQ sweep at each step of the process. Once the QR (the task modifying the green tile of Figure 3.5) is terminated, it enables all the magenta tiles to be updated in parallel. Thus, an algorithm that prioritizes all the tasks applied to left (tasks affecting the magenta tiles) are submitted, as well as the QR factorization (TSQRT) of the red tiles. For every TSQRT, the algorithm enables all the yellow tasks to run in parallel, and so our first proposition will submit all of these tasks to run in parallel. Similarly to the QR sweep, the LQ sweep wil allow all the tasks touching the magenta tiles to run in parallel, as well as the ones touching the yellow one. One can notice that in the QR sweep, the tiles of row "step" (magenta or yellow of the top row typed with "M") are modified by all the tasks. Let's call them the master tiles for the QR of sweep "step". Similarly, for the LQ sweep for the tiles of column "step+1", they are the master tiles for the LQ sweep "step". Thus, on a restricted memory system, the algorithm might be obliged to write back some of the master tiles in order to load the other master tiles to continue the same operation. As a result, for the next yellow update, the master tiles will be loaded/stored again and so on. Now, if we count the number of times tiles are used for reading and writing in Algorithm 2, Table 3.2 belows shows it for a $u \times v$ tile matrix. Each tile is used an equal number of times for reading and writing in Algorithm 2.

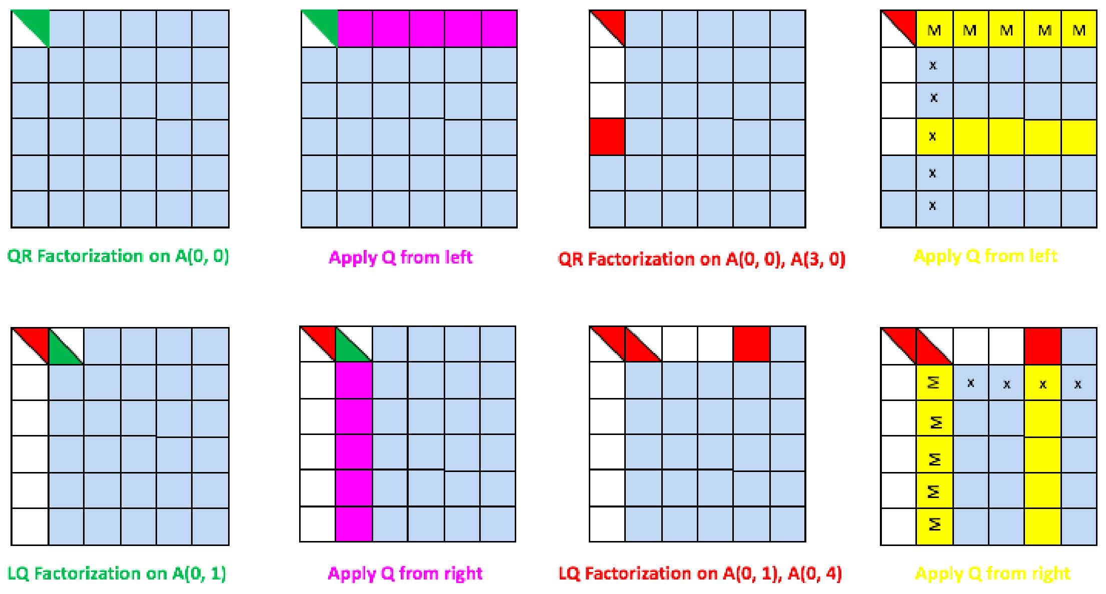

QR Factorization on A(0, 0)
Apply Q from left
QR Factorization on A(0, 0), A(3, 0)
Apply Q from left
LQ Factorization on A(0, 1)
Apply Q from right
LQ Factorization on A(0, 1), A(0, 4)
Apply Q from right
M
x

Figure 3.5: Algorithm 1 — an OOM reduction of a general matrix to band form (prioritize parallel task flow to locality).

Algorithm 2: OOM reduction of general matrix to band form (prioritize parallel task flow to locality)

```
for $s = 1$ to $nbtiles$ do
    $READ\ A(s,s)$
    $GEQRT(A(s,s))$
    for $j = s+1$ to $nbtiles$ do
        $READ\ A(s,j)$
        $UNMQR(A(s,s),\ A(s,j))$
        $WRITE\ A(s,j)$
    end
    for $k = s+1$ to $nbtiles$ do
        $READ\ A(k,s)$
        $TSQRT(A(s,s),\ A(k,s))$
        for $j = s+1$ to $nbtiles$ do
            $READ\ A(s,j)$
            $READ\ A(k,j)$
            $TSMQR(A(s,j),\ A(k,j),\ A(k,s))$
            $WRITE\ A(k,j)$
            $WRITE\ A(s,j)$
        end
        $WRITE\ A(k,s)$
    end
    $WRITE\ A(s,s)$
    if $(s < nbtiles)$ then
        $READ\ A(s,s+1)$
        $GELQT(A(s,s+1))$
        for $j = s+1$ to $nbtiles$ do
            $READ\ A(j,s+1)$
            $UNMLQ(A(s,s+1),\ A(j,s+1))$
            $WRITE\ A(j,s+1)$
        end
        for $k = s+2$ to $nbtiles$ do
            $READ\ A(s,k)$
            $TSLQT(A(s,s+1),\ A(s,k))$
            for $j = s+1$ to $nbtiles$ do
                $READ\ A(j,s+1)$
                $READ\ A(j,k)$
                $TSMLQ(A(j,s+1),\ A(j,k),\ A(s,k))$
                $WRITE\ A(j,k)$
                $WRITE\ A(j,s+1)$
            end
            $WRITE\ A(s,k)$
        end
        $WRITE\ A(s,s+1)$
    end
end
```

Table 3.2: Algorithm 2 — number of reads and writes for each tile

1	$u+1$	$u+1$	$\ldots$	$u+1$	$u+1$	$u+1$
1	$v+1$	$u+2$	$\ldots$	$u+2$	$u+2$	$u+2$
1	$v+1$	$v+2$	$\ldots$	$u+3$	$u+3$	$u+3$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$u+v-2$	$u+v-2$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$u+v-1$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$2v-1$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$2v-1$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$2v-1$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$2v-1$
1	$v+1$	$v+2$	$\ldots$	$2v-3$	$2v-2$	$2v-1$

The total number of reads and writes for Algorithm 2 are as follows:

$$\begin{aligned}
\textit{Number of tile reads} &= u + \sum_{i=1}^{v-1}(u-i)\times(v+i) + \sum_{i=1}^{v-1}(v-i)\times(u+i) \\
&= u + \sum_{i=1}^{v-1} 2\times(uv-i^2) \\
&= 2uv^2 - 2uv - \frac{2v^3}{3} + v^2 - \frac{v}{3} + u \\
\textit{Number of tile writes} &= 2uv^2 - 2uv - \frac{2v^3}{3} + v^2 - \frac{v}{3} + u
\end{aligned}$$

For square matrix, $u = v$:

$$\begin{aligned} Number\ of\ read &= \frac{4}{3}u^3 - u^2 + \frac{2}{3}u \\ Number\ of\ write &= \frac{4}{3}u^3 - u^2 + \frac{2}{3}u \end{aligned}$$

This schema, increases the number of parallel tasks but does not force the locality such that all the tiles with “x” modify the same master tile “M”; and thus it is better to keep it in memory instead of modifying it.

3.4.2 Proposition 2 — imposing locality data flow

Our second algorithmic design follows a data flow fashion that increases locality of the tasks. Unlike Algorithm 2, which prioritizes parallel task flow, Algorithm 3 prioritizes locality of the tasks so that all of them modify the same master tile. Algorithm 3 gives the details of its implementation and Figure 3.6 illustrates the fingerprint of the dataflow pattern during one step. Like Algorithm 2, the reduction process consists of a QR sweep followed by an LQ sweep at each step of the process. Once the QR (the task modifying the green tile of Figure 3.6) is terminated, QR factorizations (the TSQRT) of the red tiles are initiated. Although update by the green tile can start for all the tiles on its right, update of the magenta tile is submitted only. When QR factorizations are finished for red tiles update of the tiles on their right starts for the next column only ((yellow tiles in Figure 3.6) so that all the yellow tasks marked with ”x” modify the same master tile typed with ”M” in Figure 3.6. As every task modifies the master tile, they cannot go in parallel but reading and writing for the master tile is avoided for every yellow task marked with ”x”. Similarly to the QR sweep, the LQ sweep also forces all the tasks to modify the same master tile, thus avoiding reading and writing for the master tile.

Algorithm 3: OOM reduction of general matrix to band form (prioritizing the locality of task flow)

```
for $s = 1$ to $nbtiles$ do
    $READ\ A(s, s)$
    $GEQRT(A(s, s))$
    for $j = s + 1$ to $nbtiles$ do
        $READ\ A(s, j)$
        $UNMQR(A(s, s),\ A(s, j))$
        for $k = s + 1$ to $nbtiles$ do
            $READ\ A(k, s)$
            if $j == s + 1$ then
                $TSQRT(A(s, s),\ A(k, s))$
                $WRITE\ A(k, s)$
            end
            $READ\ A(k, j)$
            $TSMQR(A(s, j),\ A(k, j),\ A(k, s))$
            $WRITE\ A(k, j)$
        end
        $WRITE\ A(s, j)$
    end
    $WRITE\ A(s, s)$
    if $(s < nbtiles)$ then
        $READ\ A(s, s + 1)$
        $GELQT(A(s, s + 1))$
        for $j = s + 1$ to $nbtiles$ do
            $READ\ A(j, s + 1)$
            $UNMLQ(A(s, s + 1),\ A(j, s + 1))$
            for $k = s + 2$ to $nbtiles$ do
                $READ\ A(s, k)$
                if $j == s + 1$ then
                    $TSLQT(A(s, s + 1),\ A(s, k))$
                    $WRITE\ A(s, k)$
                end
                $READ\ A(j, k)$
                $TSMLQ(A(j, s + 1),\ A(j, k),\ A(s, k))$
                $WRITE\ A(j, k)$
            end
            $WRITE\ A(j, s + 1)$
        end
        $WRITE\ A(s, s + 1)$
    end
end
```

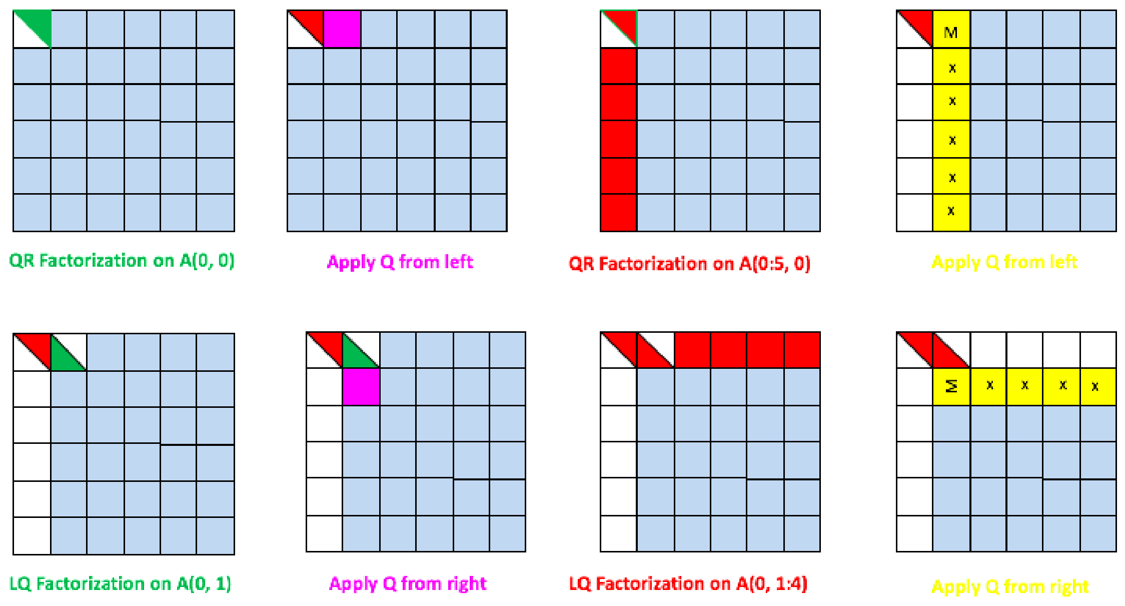

QR Factorization on A(0, 0)
Apply Q from left
QR Factorization on A(0:5, 0)
M
x
x
x
x
x
Apply Q from left
LQ Factorization on A(0, 1)
Apply Q from right
LQ Factorization on A(0, 1:4)
M
x
x
x
x
Apply Q from right

Figure 3.6: An OOM reduction of a general matrix to band form (prioritize the locality of the task flow).

Table 3.3 and Table 3.4 below show the number of reads and writes for each tile in Algorithm 3 for a $u \times v$ tile matrix. Number of tile reads and tile writes for $u \times v$ tile matrix in Algorithm 3 are as follows:

$$
\begin{aligned}
Number\ of\ tile\ reads &= \sum_{i=1}^{2v-1} i + \sum_{i=1}^{v-1} (u-i) \times (v-2+i) + (u-v) \times (2v-1) \\
&+ \sum_{i=1}^{v-1} (v-1-i) \times (u-1+i) \\
&= 2uv - u + (2uv - 3u - v + 1) \times \sum_{i=1}^{v-1} 1 \\
&+ \ 2 \times \sum_{i=1}^{v-1} i - 2 \times \sum_{i=1}^{v-1} i^2 \\
&= 2uv^2 - \frac{2}{3}v^3 - 3uv + v^2 + 2u + \frac{2}{3}v - 1
\end{aligned}
$$

$$\begin{aligned}
Number\ of\ tile\ writes &= \sum_{i=1}^{v}(u+1-i)\times(2i-1)+\sum_{i=1}^{v-1}(v-i)\times 2i \\
&= (2u+2v+3)\times\sum_{i=1}^{v-1} i-(u+1)\times\sum_{i=1}^{v-1}1 \\
&-4\times\sum_{i=1}^{v-1} i^2+(2uv+3v-2v^2-u-1) \\
&= uv^2-\frac{1}{3}v^3+\frac{1}{2}v^2-\frac{1}{6}v
\end{aligned}$$

Table 3.3: Algorithm 3 — number of reads for each tile

1	2	u	$\ldots$	u	u	u
$v-1$	3	4	$\ldots$	$u+1$	$u+1$	$u+1$
$v-1$	v	5	$\ldots$	$u+2$	$u+2$	$u+2$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$v-1$	v	$v+1$	$\ldots$	$2v-5$	$2v-4$	$u+n-1$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2n-3$	$2v-2$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2v-3$	$2v-1$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2v-3$	$2v-1$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2v-3$	$2v-1$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2v-3$	$2v-1$
$v-1$	v	$v+1$	$\ldots$	$2v-4$	$2v-3$	$2v-1$

Table 3.4: Algorithm 3 — number of writes for each tile

1	2	2	$\ldots$	2	2	2
1	3	4	$\ldots$	4	4	4
1	3	5	$\ldots$	6	6	6
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
1	3	5	$\ldots$	$2v-5$	$2v-4$	$2v-4$
1	3	5	$\ldots$	$2v-5$	$2n-3$	$2v-2$
1	3	5	$\ldots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\ldots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\ldots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\ldots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\ldots$	$2v-5$	$2v-3$	$2v-1$

For square matrix, $u = v$:

$$Number\ of\ tile\ reads = \frac{4}{3}u^3 - 2u^2 + \frac{8}{3}u - 1$$
$$Number\ of\ tile\ writes = \frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$$

Now Table 3.5 below shows the comparison of the number of tile reads and writes between Algorithm 2 and Algorithm 3 for a $u \times u$ square tile matrix. Algorithm 3 not only reduces number of tile writes by half but also reduces number of tile reads. That is why we are using Algorithm 3 for our further analysis.

Table 3.5: Total reads and writes of tiles for Algorithm 2 & Algorithm 3

Algorithm	# of total read	# of total write
Algorithm-2	$\frac{4}{3}u^3 - u^2 + \frac{2}{3}u$	$\frac{4}{3}u^3 - u^2 + \frac{2}{3}u$
Algorithm-3	$\frac{4}{3}u^3 - 2u^2 + \frac{8}{3}u - 1$	$\frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$

3.5 Optimize communication overhead

From section 3.3 we know that we can a hide very small portion of the communication cost by the computation. So, our goal is to optimize the number of reads and writes, which in turn will reduce the communication overhead. We want to the hold tiles in memory that are used most when the algorithm runs, thus reducing the total number of tile reads and writes. Table 3.6 below shows the number of reads and writes Algorithm 3 has for each tile of a $u \times u$ square tile matrix. The most-used tiles are from 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.

Table 3.6: Algorithm 3 — total reads and writes for each tile

$(1\ \ 1)$	$(2\ \ 2)$	$(u\ \ 2)$	$\dots$	$(u\ \ 2)$	$(u\ \ 2)$	$(u\ \ 2)$
$(u-1\ 1)$	$(3\ \ 3)$	$(4\ \ 4)$	$\dots$	$(u+1\ \ 4)$	$(u+1\ \ 4)$	$(u+1\ \ 4)$
$(u-1\ \ 1)$	$(u\ \ 3)$	$(5\ \ 5)$	$\dots$	$(u+2\ \ 6)$	$(u+2\ \ 6)$	$(u+2\ \ 6)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$(u-1\ \ 1)$	$(u\ \ 3)$	$(u+1\ \ 5)$	$\dots$	$(2u-5\ \ 2u-5)$	$(2u-4\ \ 2u-4$	$(2u-3\ \ 2u-4)$
$(u-1\ \ 1)$	$(u\ \ 3)$	$(u+1\ \ 5)$	$\dots$	$(2u-4\ \ 2u-5)$	$(2u-3\ \ 2u-3)$	$(2u-2\ \ 2u-2)$
$(u-1\ 1)$	$(u\ \ 3)$	$(u+1\ \ 5)$	$\dots$	$(2u-4\ \ 2u-5)$	$(2u-3\ \ 2u-3)$	$(2u-1\ \ 2u-1)$

If one tile from the lower right corner of the matrix is held in memory, then in each step of Algorithm 3 we can save one read and one write for both the QR and LQ sweeps. In short, we can reduce two reads and two writes in every step as shown in Figure 3.7. If R_{1c} is the number of tile reads and writes reduced by holding one tile in memory, then:

$$\begin{aligned} Hold\ 1\ tile\ from\ lower\ right\ corner\ reduces, R_{1c} &= 2 + 2\times(u-2) \\ &\quad + 2 + 2\times(u-2)\ read, write \\ &= 4(u-1)\ read, write \end{aligned}$$

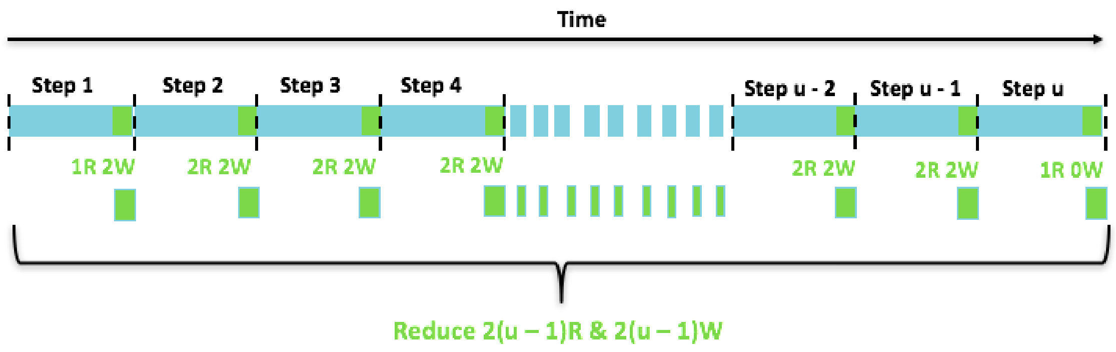


Figure 3.7: Reducing the number of tile reads and writes — holding one tile from the lower corner of the matrix in memory.

From section 3.4.2, we know that each step of Algorithm 3 has two sweeps — QR and LQ. The QR sweep is always followed by the LQ sweep and dependency exists between them. That is why we next analyzed Algorithm 3 step by step. Holding one tile in memory, our goal is to have a reduction of more than two reads and two writes in each step of the algorithm. So we count the number of times tiles are used in each step of the algorithm.

Table 3.7 and Table 3.8 show the number of reads and writes for each tile in $u \times u$ tile matrix after the QR and LQ sweeps in first step.

Table 3.7: Number of reads and writes for each tile after the first step of the QR sweep (Algorithm 3)

$(1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$
$(u-1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$
$(u-1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$(u-1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$
$(u-1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$
$(u-1\ 1)$	$(1\ 1)$	$(1\ 1)$	$\dots$	$(1\ 1)$	$(1\ 1)$	$(1\ 1)$

From Table 3.7 and Table 3.8 one can observe that the tiles from the panel (the first tile column for the QR sweep and first tile row for the LQ sweep) are communicated more in each step of the algorithm. If one tile from the panel is held in memory, then the $(u-2)$ reads can be reduced for both the QR and LQ sweeps, in total $2(u-2)$ reads in first step. Figure-3.8 shows the number of reduced tile reads in each step of the algorithm.

Table 3.8: Number of reads and writes for each tile after the first step of the LQ sweep (Algorithm 3)

$(1\ 1)$	$(2\ 2)$	$(u\ 2)$	$\ldots$	$(u\ 2)$	$(u\ 2)$	$(u\ 2)$
$(u-1\ 1)$	$(2\ 2)$	$(2\ 2)$	$\ldots$	$(2\ 2)$	$(2\ 2)$	$(2\ 2)$
$(u-1\ 1)$	$(2\ 2)$	$(2\ 2)$	$\ldots$	$(2\ 2)$	$(2\ 2)$	$(2\ 2)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$(u-1\ 1)$	$(2\ 2)$	$(2\ 2)$	$\ldots$	$(2\ 2)$	$(2\ 2)$	$(2\ 2)$
$(u-1\ 1)$	$(2\ 2)$	$(2\ 2)$	$\ldots$	$(2\ 2)$	$(2\ 2)$	$(2\ 2)$
$(u-1\ 1)$	$(2\ 2)$	$(2\ 2)$	$\ldots$	$(2\ 2)$	$(2\ 2)$	$(2\ 2)$

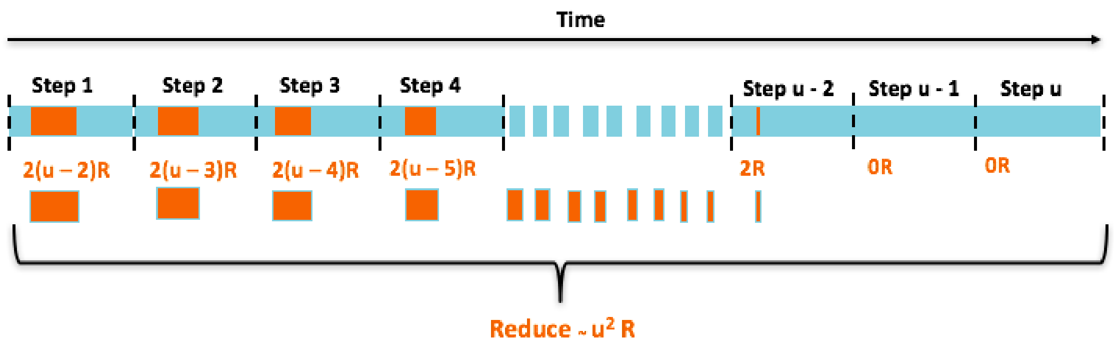


Figure 3.8: Reducing the number of reads and writes — holding one tile from the panel.

If R_{1p} and R_{up} is the number of tile reads reduced by holding one tile and u tile from the panel in memory respectively, then

$$\begin{aligned}
Holding\ 1\ tile\ from\ panel\ reduces,\ R_{1p} &= \sum_{s=1}^{u-1} 2(u-s-1) \\
&= (u^2 - 3u + 2)\ read \\
Holding\ u\ tile\ from\ panel\ reduces,\ R_{up} &= Reduce\ for\ QR\ sweep \\
&+ Reduce\ for\ LQ\ sweep \\
&= \sum_{s=1}^{u-1} (u-s-1)(u-s) \\
&+ \sum_{s=1}^{u-1} (u-s-1)(u-s-1) \\
&= \sum_{s=1}^{u-1} [2(u-s-1)(u-s) - (u-s-1)] \\
&= \frac{2}{3}u^3 - \frac{5}{2}u^2 + \frac{17}{6}u - 1\ read
\end{aligned}$$

Figure 3.9 compares the number of tile reads and writes that can be reduced by holding tiles from different portions (panel or lower right corner of the matrix) of a $u \times u$ tile matrix. Holding tiles from the panel reduces $O(u^3)$ reads and writes, compared with $O(u^2)$ reads and writes by holding tiles from the lower right corner of the matrix. Table 3.9 shows the number of reads and writes for each tile for a $u \times v$ tile matrix when the panel is held in memory. So, the total number of reads and writes when the panel is held in memory is given by the following:

$$\begin{aligned}
Number\ of\ reads\ or\ writes &= 1 \times \sum_{s=0}^{v-1} (u-s) + 2 \times \sum_{s=0}^{v-1} (u-s)(v-s-1) \\
&= 1 \times \sum_{s=0}^{v-1} (u-s) + (2v-2) \times \sum_{s=0}^{v-1} (u-s) \\
&- 2u \times \sum_{s=0}^{v-1} s + 2 \times \sum_{s=0}^{v-1} s^2 \\
&= uv^2 - \frac{v^3}{3} + \frac{v^2}{2} - \frac{v}{6}
\end{aligned} \tag{3.1}$$

For a square matrix with $u \times u$ tile:

$$Number\ of\ readsorwrites = \frac{2}{3}u^3 + \frac{u^2}{2} - \frac{u}{6}$$

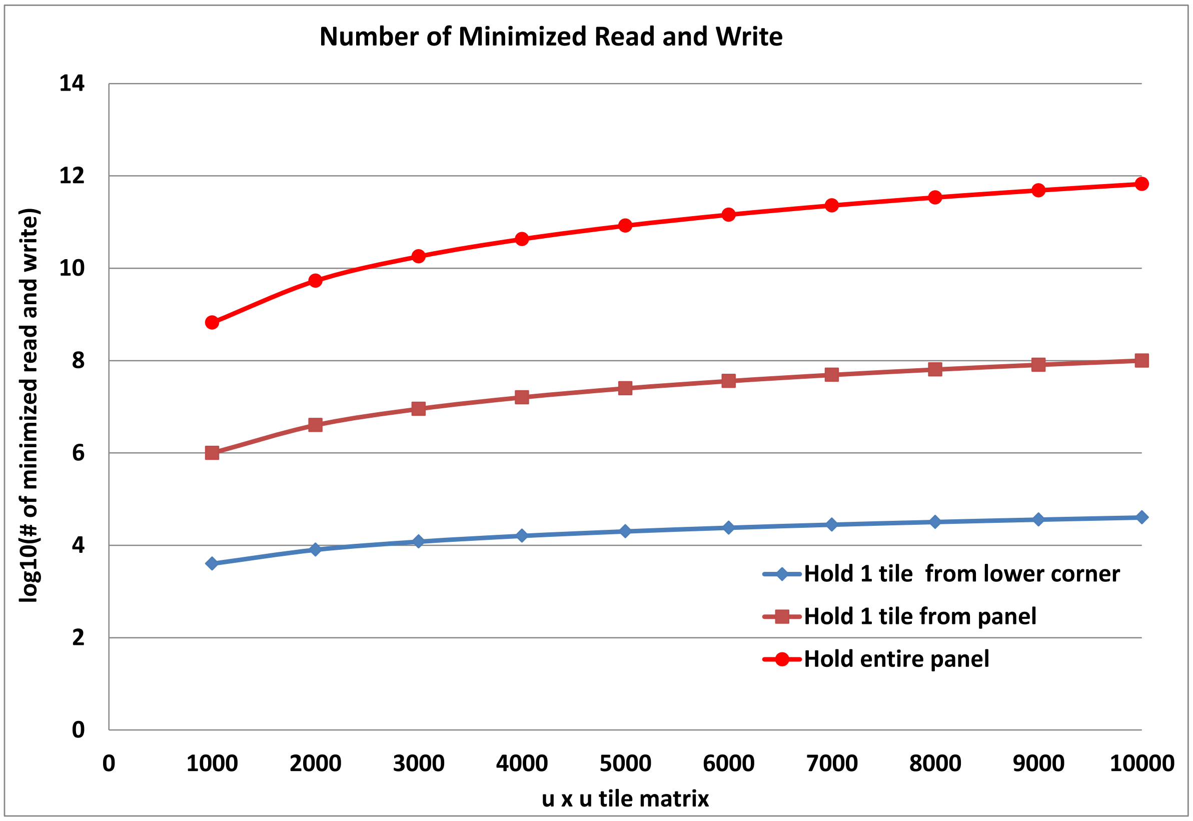


Figure 3.9: Number of reads and writes of tiles that can be minimized.

Table 3.10 below compares the number of reads and writes for Algorithm 3 when the panel is held in memory with both Algorithm 3 and Algorithm 2. From Table 3.10 one can observe that holding the tiles from the panel reduces the number of tiles read by half.

Table 3.9: Number of reads and writes for each tile — panel is held in memory for Algorithm 3.

1	2	2	$\dots$	2	2	2
1	3	4	$\dots$	4	4	4
1	3	5	$\dots$	6	6	6
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
1	3	5	$\dots$	$2v-5$	$2v-4$	$2v-4$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-2$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-1$
1	3	5	$\dots$	$2v-5$	$2v-3$	$2v-1$

Table 3.10: Comparison of total number of reads and writes

Algorithm	# of total read	# of total write
Algorithm-2	$\frac{4}{3}u^3 - u^2 + \frac{2}{3}u$	$\frac{4}{3}u^3 - u^2 + \frac{2}{3}u$
Algorithm-3	$\frac{4}{3}u^3 - 2u^2 + \frac{8}{3}u - 1$	$\frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$
Algorithm-3 (Hold panel)	$\frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$	$\frac{2}{3}u^3 + \frac{1}{2}u^2 - \frac{1}{6}u$

If after holding a panel in memory we can hold more, then tiles from the lower right corner of the matrix are held. Figure 3.10 shows the order of tiles to hold in memory. If ten tiles can be held in memory, then tiles numbered 1 through 10 in Figure 3.10 are held in memory. Another important thing to note is that in Algorithm 3, panel length decreases by one in each step. That means memory that was used to hold tiles from the panel will not be used in the subsequent steps and can be used to hold tiles from lower right corner of the matrix as shown in Figure 3.11.

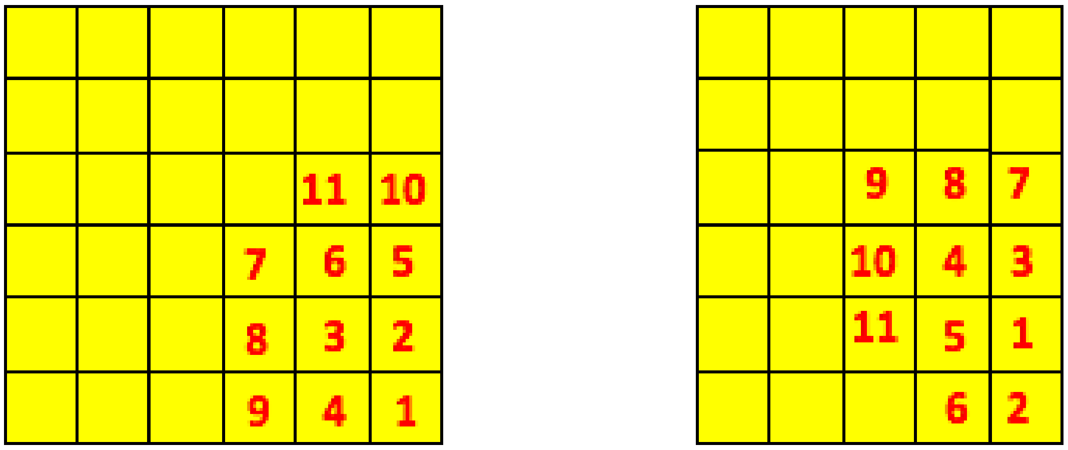


Figure 3.10: Tile order from the lower corner of the matrix held in memory.

In summary, our optimization techniques to reduce data movements are:

1. As first priority, hold tiles from the panel. For a $u \times v$ tile matrix, u tiles are held in memory if $u > v$; otherwise v tiles are held.

2. If after holding the panel we can hold more tiles, then tiles from the lower right corner of the matrix are held in memory.

3. One tile of memory is freed in each step of the algorithm, as the panel length decreases by one in every LQ sweep of the algorithm. This memory tile can be used to hold tiles from the lower right corner of the matrix in subsequent steps.

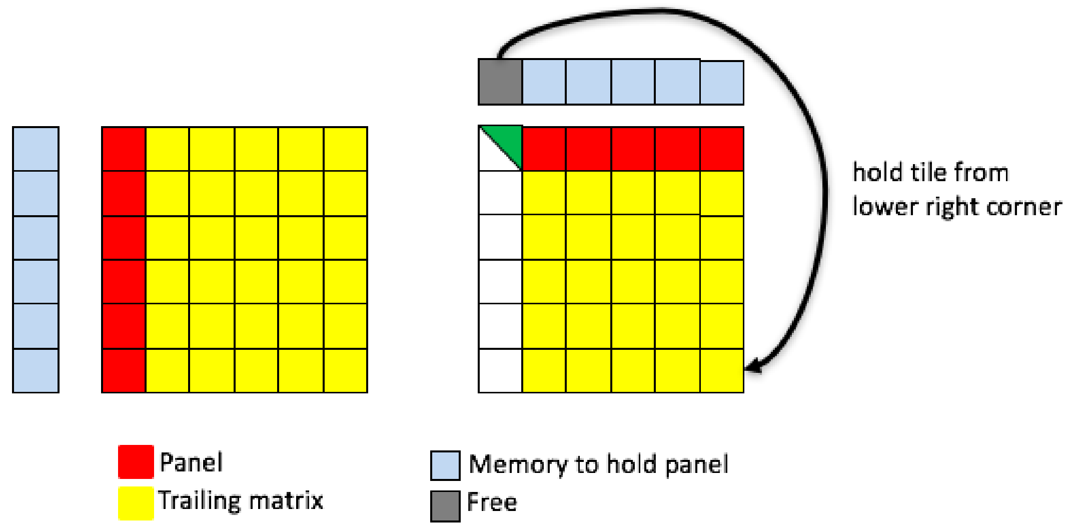


Figure 3.11: Freed memory from the panel in each step — use to hold tiles from the lower right corner of the matrix.

3.5.1 Estimated run time for an OOM two-stage SVD solver

In this section we present a formula to estimate the runtime for first stage of a two-stage OOM SVD solver. For a $u \times v$ tile matrix of tile size b, if only the tile form panel (u tiles if $u > v$; otherwise v) can be held in memory, the total runtime is bounded by the number of tile reads and writes. The estimated time can be easily computed from the number of reads and writes presented in section 3.4.2 and section 3.5. But there might be unused memory to hold tiles from the lower right corner of the matrix after holding the panel as shown in Figure 3.12. Let's say T tile, yellow tiles from the lower right corner, and panel, red tiles from the panel, are hold in memory. The estimated runtime for the first stage of OOM SVD, T_{est}, is given by:

$$T_{est} = T_{read} + T_{write} + T_{compute} - T_{overlap}$$

Where T_{read} and T_{write} are the total read and write times spent in communication, $T_{computation}$ is the computation time for yellow tiles in Figure 3.12 and $T_{overlap}$ is the computation time overlapped by the reads and writes of some of the tiles. One can see that we are not considering computation time for white tiles in Figure 3.12. Because reads and writes of white tiles hide the computation time. If $(u - v + l) \times l$ is the size of submatrix held by T tiles, then l is computed by,

$$l = \frac{-(u-v) + \sqrt{(u-v)^2 + 4 * T}}{2}$$

If NR and S_1 is the number of tile reads required for $u \times v$ and $(u - v + l) \times l$ tile matrix, respectively, then from equation-(3.1), NR and S_1 is given by:

$$\begin{aligned} NR &= uv^2 - \frac{v^3}{3} + \frac{v^2}{2} - \frac{v}{6} \\ S_1 &= (u - v + l)l^2 - \frac{l^3}{3} + \frac{l^2}{2} - \frac{l}{6} \\ &= (u - v)l^2 + \frac{2l^3}{3} + \frac{l^2}{2} - \frac{l}{6} \end{aligned}$$

Each tile of T can save two reads in each step of Algorithm 3. From $(v - l + 1)$th to the $(v - 1)th$ step of Algorithm 3, $(u - v + l) \times l$ tile can save S_1 reads. the number of reads that can be saved in 0 to $v - l$ step using T tiles, S_2, is given by:

$$S_2 = \begin{cases} 2 * (v - l - 1) * T + min(T, (u - v + l + 1) * l), & \text{if } (v - l) > 0 \\ 0, & \text{otherwise} \end{cases}$$

So, $(S_1 + S_2)$ is the total number of tile reads saved by T tiles and also the number of computation for T tiles in the v step. So, now:

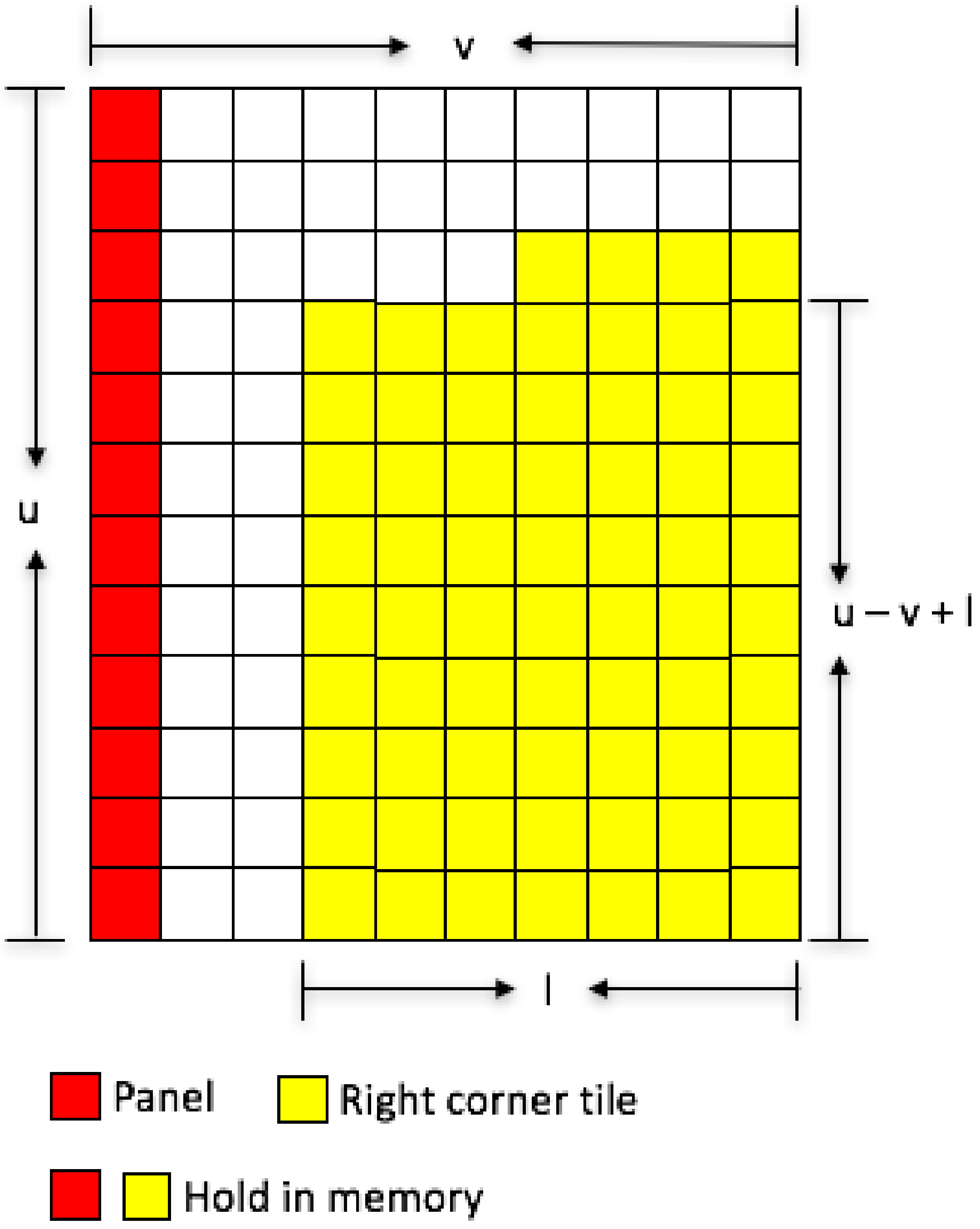


Figure 3.12: Panel and lower right corner tiles are held in memory

$$
\begin{aligned}
T_{read} &= (NR - (S_1 + S_2)) \times \frac{8b^2}{BW} \\
T_{write} &= (NR - (S_1 + S_2)) \times \frac{8b^2}{BW} \\
T_{compute} &= (S_1 + S_2) \times \frac{5b^3}{\alpha}
\end{aligned}
$$

During computation, master tiles and tiles from the panel are sent back to disk for write. So overlap time for v step is as follows:

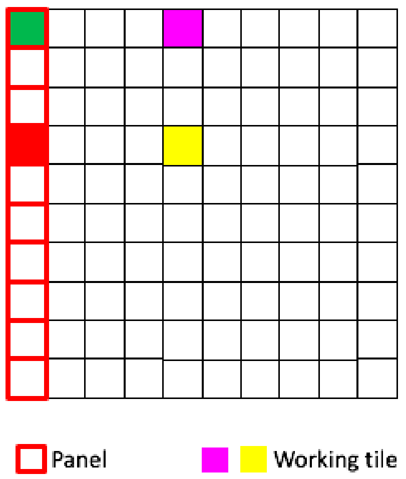
$$
\begin{aligned}
T_{overlap} &= (write\ in\ QR\ step + write\ in\ LQ\ step) \times \frac{8b^2}{BW} \\
&= (\sum_{s=0}^{v}(u-s) + \sum_{s=0}^{v}(v-s) \times \frac{8b^2}{BW} \\
&= 2(uv + u) \times \frac{8b^2}{BW}
\end{aligned}
$$

Now the estimated time for the first stage of OOM SVD solver is given by the following:

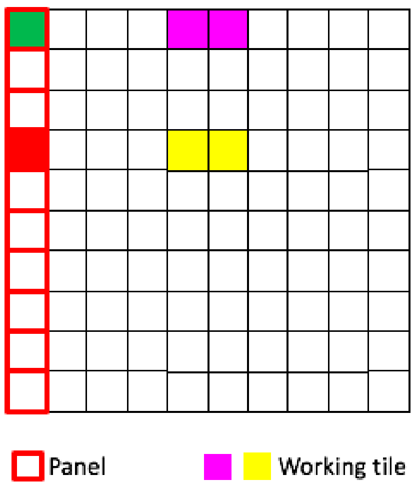
$$
\begin{aligned}
T_{est} &= T_{read} + T_{write} + T_{compute} - T_{overlap} \\
&= 2(NR - (S_1 + S_2)) \times \frac{8b^2}{BW} + (S_1 + S_2) \times \frac{5b^3}{\alpha} - 2(uv + u) \times \frac{8b^2}{BW} \qquad (3.2)
\end{aligned}
$$

3.6 Experiment result

To evaluate the performance of OOM two-stage algorithms we have done a number of experiments and collected an execution trace to show how execution time overlapped with data movement to and from the disk. It is not easy to collect a trace for matrices that doesn't fit in memory, because the size of the trace is too big to visualize in trace viewer software and is also time-consuming. To simulate OOM SVD algorithms for small matrices, read and write times for tiles need to be adjusted as the system puts them in cache. We use the sleep function in the read/write kernel to match read and write times for the disk. If BW is the bandwidth of the disk and b is tile size, the time required to read/write a tile from/to the disk is $\frac{8b^2}{BW}$s. We use normal distribution to generate read and write times with mean $\frac{8b^2}{BW}$s and variance is 40% of mean. We collected the execution trace in a Sandy Bridge(Xeon E5-2670) machine that has HDD of 50 MB/s bandwidth. In all of your trace, *green* represents $GEQRT/GELQT$, *red* represents $TSQRT/TSLQT$, *magenta* represents $GEMQR/GEMLQ$, and *yellow* represents the $TSMQR/TSMLQ$ routine. Read and write tasks are represented by *cyan* and *purple*, respectively.



(a) Two working tile.



(b) Four working tile.

Figure 3.13: Memory use for the first stage of an OOM two-stage algorithm.

Our first experiment holds the panel in memory. That means tiles from the panel will be loaded in memory at each step of the algorithm and sent back to disk for writing when the step ends. All other tiles from the trailing matrix have to be brought in memory once for the QR sweep and again for the LQ sweep for each step of the algorithm. We need two other tiles (called working tile) — one holds the master tile (magenta) and the other holds a tile (yellow) from the same column (row, for the LQ sweep) in memory as shown in Figure 3.13a. Figure 3.14 shows the execution trace of the QR sweep when two tiles are used as working tiles. One can see that execution of the $TSMQR$ routine does not overlap with the read and write task. Because after execution of the $TSMQR$ routine, the yellow tile in Figure 3.13a has to be sent back to the disk for writing before another tile from the same column is loaded into memory to modify the master tile. To overlap execution with communication, another two working tiles (four tiles in total) are needed as shown in Figure 3.13b. Now execution of the $TSMQR$ routine is overlapped with the tile read and write as shown in Figure-3.15. Because there are four working tiles and two of them are used to hold the master tile, one yellow tile is loaded into memory during the execution of the $TSMQR$ routine for another yellow tile.

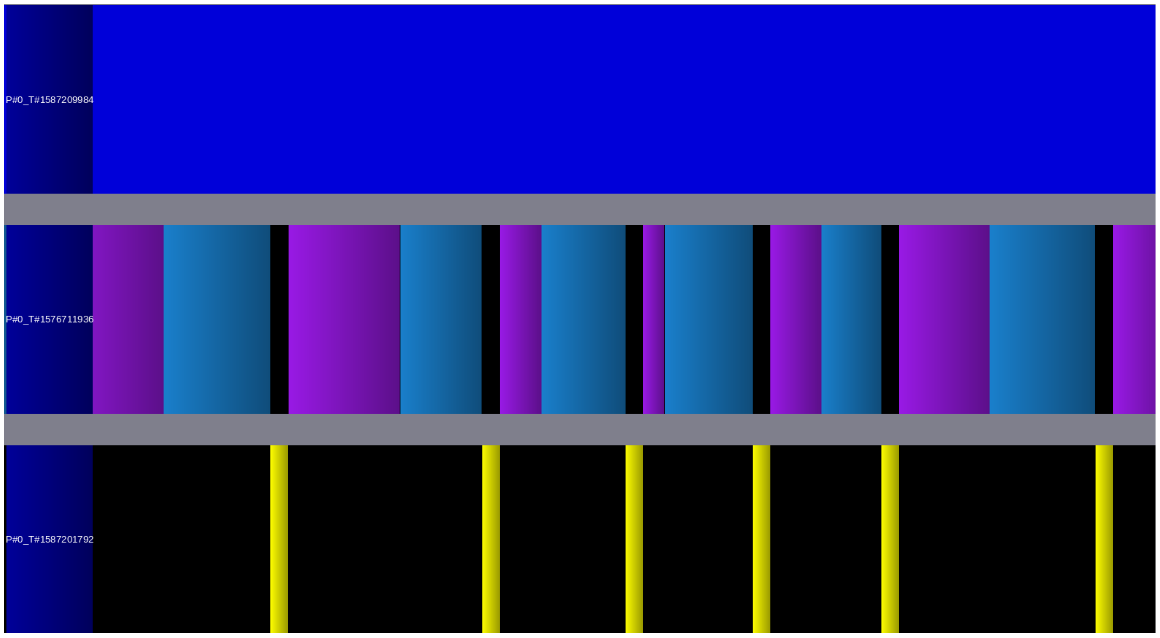


Figure 3.14: Execution trace (two working tiles) — computation does not overlap communication.

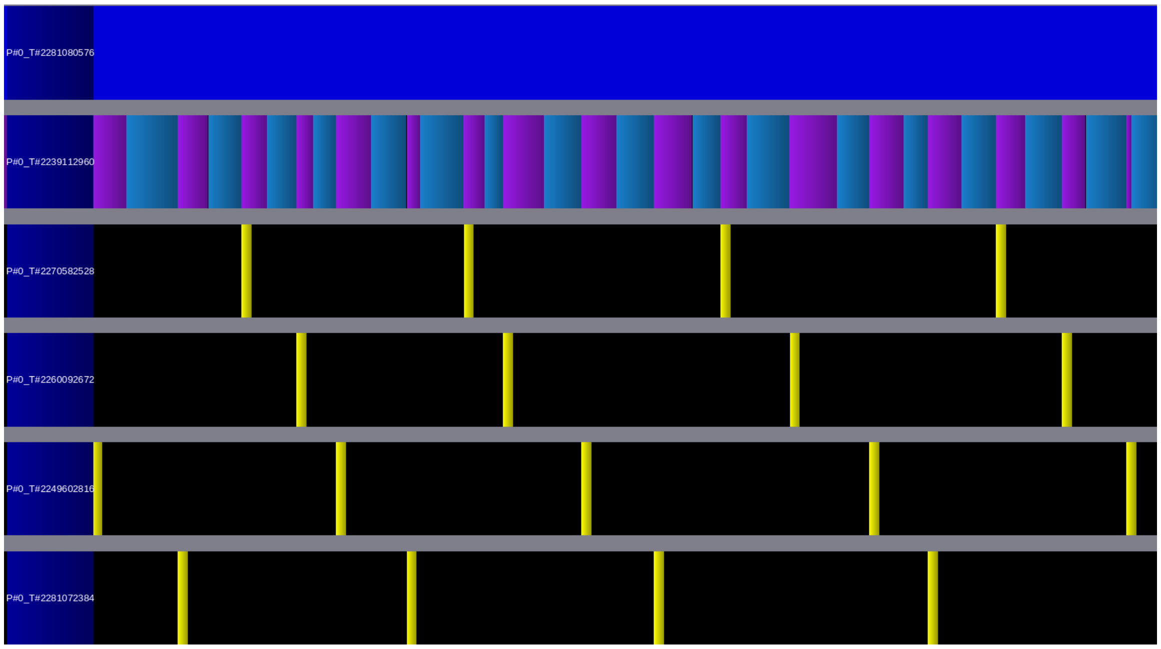
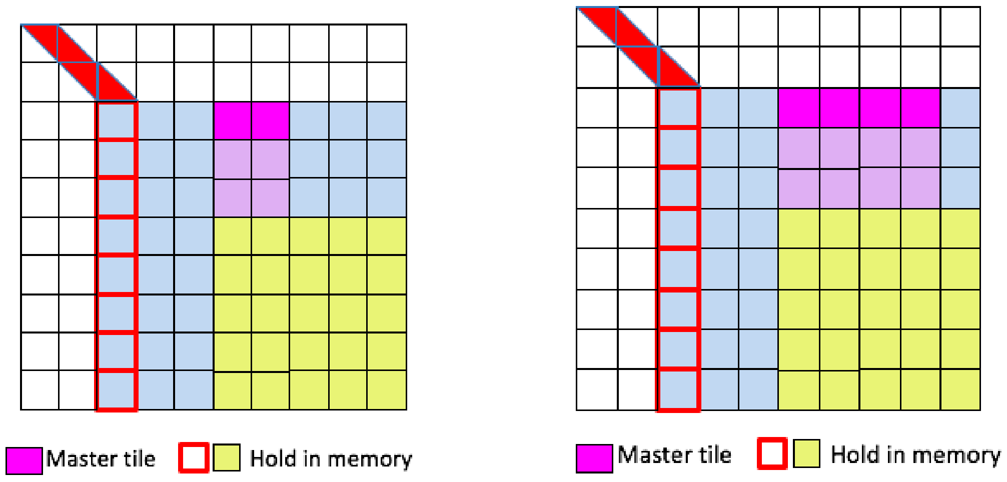


Figure 3.15: Execution trace (four working tiles) — computation overlaps communication.

For both of the above experiments we use one thread to submit tasks in the QUARK queue and another thread to handle read write tasks. The rest of the threads execute the computation tasks. One can notice that in Figure 3.15 no $TSMQR$ tasks are running in parallel. As there are only four working tiles and read/write tasks are slower than the $TSMQR$ tasks, we can bring only one tile into memory during the execution of a computation task.



(a) Four working tiles. **(b)** Two working tiles per thread.

Figure 3.16: Memory use for the first stage of an OOM two-stage algorithm — tiles from the panel and lower right corner are held in memory

In our next experiment, we not only held tiles from the panel but also from the lower right corner as shown in Figure 3.16a. As before, four working tiles are used — two master tiles and two other tiles that modify the master tiles are loaded into the working tile. When threads are working in the non-yellow region in Figure 3.16a, no two tasks can run in parallel as shown in the execution trace in Figure 3.17. But as yellow tiles are in memory now, tasks for this region can be executed in parallel. Although many yellow tiles are in memory, only two tasks are running in parallel as shown in Figure 3.17 because of their dependency (the $TSMQR$ tasks depend on the master tiles) on master tile. All the $TSMQR$ tasks of same the column depend on the same master tile and cannot run in parallel. As there are only two master tiles in memory, only two $TSMQR$ tasks can run in parallel as shown in Figure 3.17. To run the tasks of the yellow region in parallel, two working tiles per thread are needed as shown in Figure 3.16b. When each thread has two working tiles they can run in parallel in the yellow region as shown in the trace of Figure 3.18

For our last experiment, we assume that the entire matrix fits in memory. So our program will read the entire matrix in memory in the QR sweep of the first step of the algorithm and write tiles from the panel to the disk in every step of the algorithm. Figure 3.19 shows the execution trace for the first few steps of the algorithm. In the QR sweep of the first step, the whole matrix is loaded into memory, so no tasks can run in parallel. When all the tiles are in memory, the program runs like an in-memory algorithm and tasks are executed in parallel fashion. In each step of the algorithm, tiles from the panel for both the QR and the LQ sweeps are sent back to disk for writing and can overlap with computation as shown in Figure 3.19.

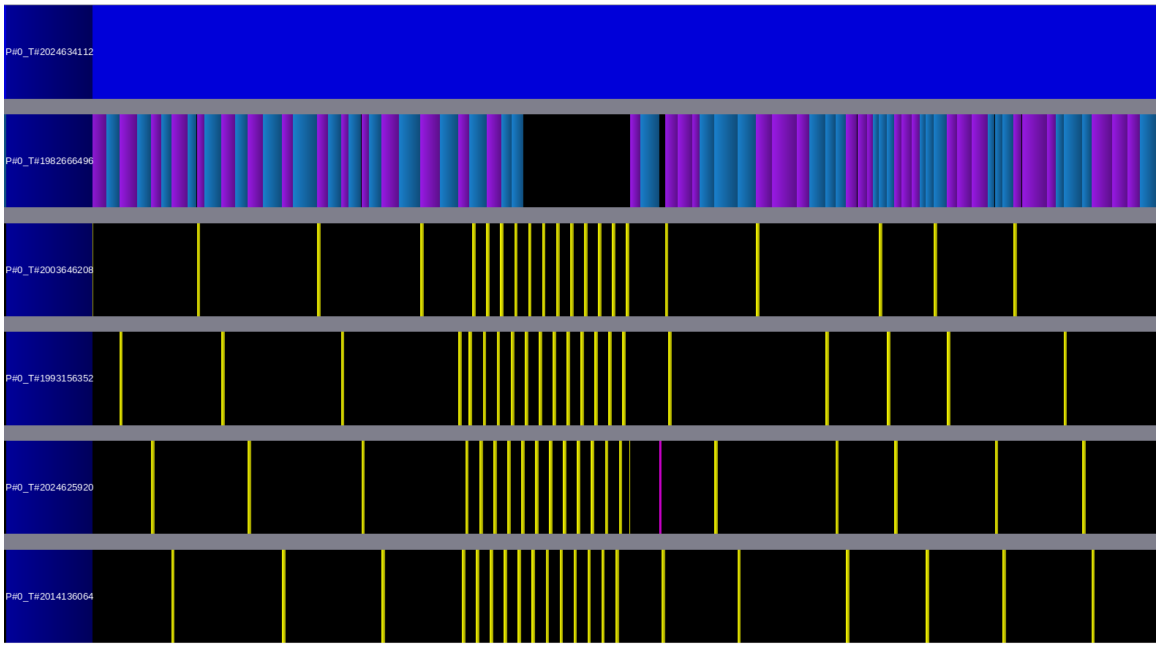


Figure 3.17: Execution trace — four working tiles (only two tasks can run in parallel).

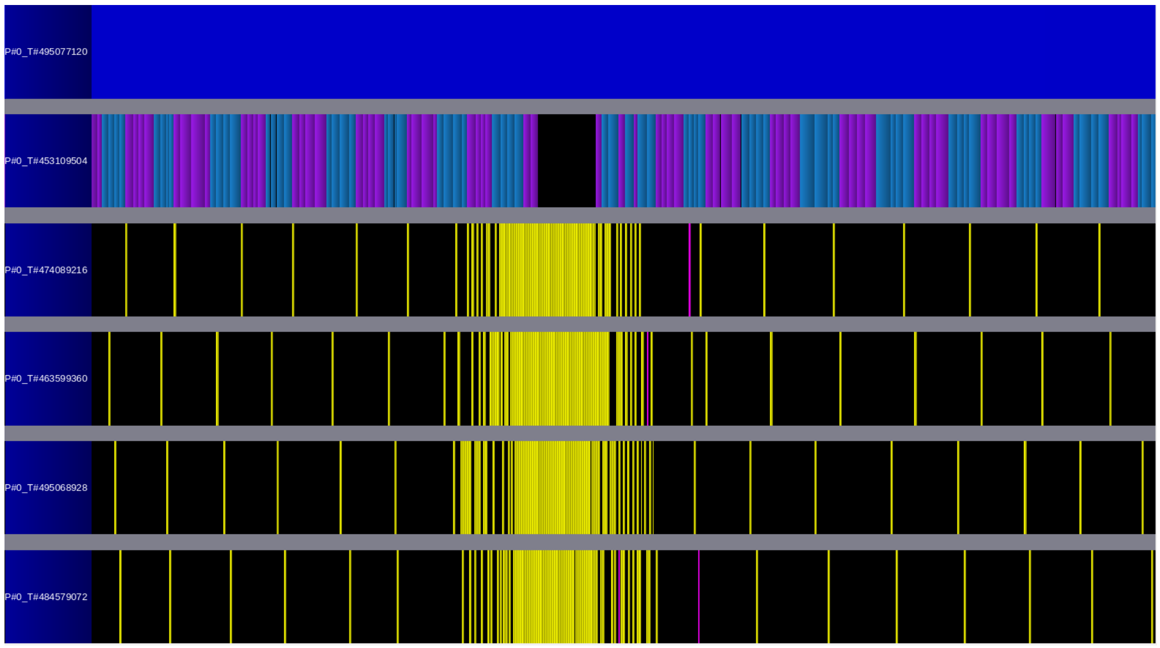


Figure 3.18: Execution trace — two working tiles per thread (All threads can run in parallel.

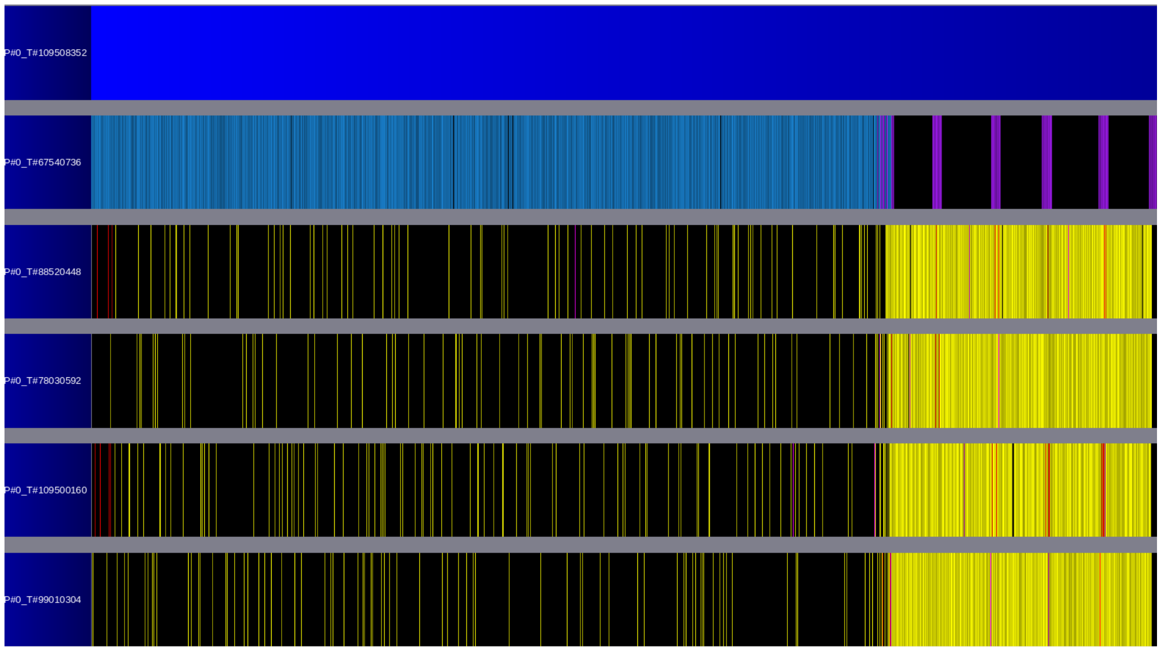


Figure 3.19: Execution trace — the entire matrix fits in memory.

3.7 Performance of an OOM SVD solver

In this section we present the performance of an OOM SVD solver when the matrix does not fit in the main memory. We run our experiment on both Haswell i7-5930K and Haswell E5 2650 V3 machines. The details of the machines and storage devices are provided in Table 2.2 and Table 2.4. Since both machines have 32 GB of memory we ran our OOM SVD solver for matrices that do not fit in 32GB memory. To estimate the runtime of the first stage (reduction of the general matrix to band form), we used Equation 3.2 to show its accuracy through a comparison with actual runtime. Table 3.11 shows the size of the matrix we used in the experiment, and the value of tile size, disk bandwidth, and the performance of the update kernel (the $TSMQR$) we used to estimate the run time for two-stage SVD algorithms.

Table 3.11: Matrix size, tile size, disk bandwidth, and the update kernel performance for runtime estimation.

Matrix	Size	Tile size	Haswell i7-5930K Samsung SSD		Haswell i7-5930K Seagate Constellation	
			Disk bandwidth	Update kernel performance	Disk bandwidth	Update kernel performance
100k x 20k	16GB	128	180	160	130	300
100k x 40k	32GB	128	180	160	130	300
100k x 60k	48GB	512	145	160	110	300
100k x 80k	64GB	512	145	160	110	300
100k x 100k	80GB	512	145	160	110	300

When the matrix is really big (i.e., $100k \times 100k$), small tile size creates many tasks in each step of the algorithm presented in Algorithm 2 and Algorithm 3. The huge number of computation and read/write tasks not only increases the QUARK scheduler's overhead but also decreases the overall performance of update kernel(the $TSMQR$ routine). It also generates so many small read and write tasks that it increases the disk traffic and negatively affects the bandwidth of the disk. Even though Samsung SSD and Seagate Constellation HDD have higher sequential read/write bandwidth, as shown in Table 2.4 we are unable to achieve that because of complex tile access order inside the first-stage of two-stage algorithm. Big tile size helps to have smaller number of tasks and overcome some of these short comings, but also simultaneously increases runtime for second stage (reduction of band matrix to bidiagonal form) of the two-stage algorithm. Table 3.12 shows the effect of tile size for an OOM SVD solver for $100k \times 60k$ matrix when we run it on a Haswell E5 2650V3 machine. Basically, the second stage of the SVD solver is memory bound and performance of it depends on memory bandwidth. In Table 3.12, execution time for both stage is shown for two tile size. Big tile size (i.e., 512) improves the performance of the first stage compared with tile size 128 because of the higher disk bandwidth it achieves. At the same time, big tile size takes more time for the second stage because of the dependency on memory bandwidth for the second stage.

Table 3.12: Effect of tile size - two stage SVD algorithm

Tile size	Disk Bandwidth	Update kenel performance	First stage of two-stage SVD algorithm		Second stage of two-stage SVD algorithm
			Estimated time(s)	Actual Time(s)	
128	80	300	48317	50061	243
512	110	300	14198	13257	1922

In Table 3.13 and Table 3.14, we present the execution times for the first-stage (reduction of general matrix to band form) we estimated using Equation 3.2 described in section 3.5.1 and also the actual time the OOM SVD algorithm is taking when it runs on Haswell i7-5930K and Haswell E5 2650V3 machines. Since the system has 32 GB of memory, our OOM SVD solver uses the maximum amount of memory allowed by the system. The second column of the Table 3.13 and Table 3.14 shows the number of tiles an OOM SVD algorithm uses to hold tiles from the lower right corner of the matrix. We also report total number of tile reads and writes required by the first stage of an OOM SVD algorithm. From Table 3.13 and Table 3.14 we can observe that the estimated first-stage execution time is close to the actual run time for most of the test cases on both Haswell i7-5930K and Haswell E5 2650V3 machines. For example, on Haswell E5 2650V3 machine, for a $100k \times 100k$ matrix, actual runtime for the first stage is 19.04 hours, whereas estimated runtime using Equation 3.2 is 19.70 hours.

Table 3.13: An OOM two-stage algorithm — execution time for the first stage (Haswell i7-5930K, Samsung SSD)

Haswell i7-5930K, Memory 32GB Disk — Samsung SSD					
Matrix	#Tile to hold right lower corner	# of tile reads	# of tile writes	Estimated execution time first stage(h)	Actual execution time first stage(h)
100k x 20k	240624	122774	122774	0.32	0.33
100k x 40k	240624	257526	257526	1.21	1.20
100k x 60k	14247	286256	286256	4.50	4.36
100k x 80k	14247	882337	882337	10.19	9.90
100k x 100k	14247	1728575	1728575	17.73	17.21

Table 3.14: An OOM two stage algorithm — execution time for the first stage (Haswell E5 2650V3, Seagate Constellation)

Haswell E5 2650V3, Memory 32GB Disk — Seagate Constellation					
Matrix	#Tile to hold right lower corner	# of tile reads	# of tile writes	Estimated execution time first stage (h)	Actual execution time first stage (h)
100k x 20k	240604	122774	122774	0.174	0.171
100k x 40k	240604	257666	257666	0.64	0.58
100k x 60k	14668	260753	260753	3.94	3.68
100k x 80k	14668	836539	836539	10.52	10.28
100k x 100k	14668	1664993	1664993	19.70	19.04

Table 3.15 shows the overall runtime for a two-stage OOM SVD solver for the Haswell E5 2650V3 machine. It also compares the performance of a two-stage OOM algorithm with one-stage as both of them reduce the general matrix to bidiagonal form. To compute the extrapolated runtime for one-stage algorithm, we assumed that the one-stage algorithm is getting the same disk bandwidth as the OOM SVD. For all the matrices we tested the two-stage OOM SVD algorithm took less time as compare with one stage on the Haswell E5 2650V3 machine. For example, for a $100k \times 100k$ matrix the two-stage OOM SVD algorithm is taking only 20.57 hours, whereas a one-stage algorithm will take 562 days to solve it.

Table 3.15: Execution time for an OOM two-stage algorithm (Haswell E5 2650V3, Seagate Constellation) and comparison with a one-stage algorithm

Haswell E5 2650V3, Memory 32GB Disk - Seagate Constellation				
Matrix	Execution time first stage (h)	Execution time second stage (h)	Execution time OOM two stage algorithm (h)	Extrapolated OOM one stage time (d)
100k x 40k	0.58	0.02	0.60 hours	116 days
100k x 60k	3.68	0.53	4.22 hours	242 days
100k x 80k	10.28	0.96	11.24 hours	395 days
100k x 100k	19.04	1.54	20.57 hours	562 days

From Table 3.15 one can observe that our two-stage OOM SVD can solve big problems that could not be solved using a traditional SVD algorithm in limited time (i.e. in one or two day). The reason is that the two-stage OOM SVD reduces disk traffic significantly using all the strategies explained in section 3.5 and avoids Level 2 BLAS whenever possible.

Chapter 4

OOM SVD using OOM QR decomposition

4.1 Introduction and motivation

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.

- Generate QR factorization of the a general matrix A, as $A = QR$.
- Generate singular value decomposition of 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 SVD of A and QR + SVD of A are as follows:

$$\begin{aligned}
Computational\ complexity\ for\ SVD\ of\ A(m,n) &= QR\ for\ A(m,n) \\
&+\ LQ\ for\ A(m,n) \\
&= 2n^2(m - \frac{n}{3}) \\
&+ 2n^2(m - \frac{n}{3}) \\
&= (4mn^2 - \frac{4}{3}n^3) \\
&= 2n^2(2m - \frac{2}{3}n) \\
Computational\ complexity\ for\ QR + SVD\ of\ A(m,n) &= QR\ for\ A(m,n) \\
&+ SVD\ for\ R(n,n) \\
&= QR\ for\ A(m,n) \\
&+ QR\ for\ R(n,n) \\
&+ LQ\ for\ R(n,n) \\
&= 2n^2(m - \frac{n}{3}) \\
&+ \frac{4}{3}n^3 + \frac{4}{3}n^3 \\
&= 2n^2(m + n)
\end{aligned}$$

So, the computation complexity to reduce $m \times n$ matrix A to bidiagonal form is $(4mn^2 - \frac{4}{3}n^3)$ flop, where as the two-step approach requires $2n^2(m - \frac{n}{3})$ flop for QR decomposition of A, and $\frac{8}{3}n^3$ flop to reduce matrix R to bidiagonal form, in total $2n^2(m + n)$ 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 only.

- 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 algorithm might be used. Instead of reducing

A to bidiagonal form directly using a two-stage OOM SVD algorithm, A might be decomposed as $A = QR$ using faster OOM QR. Later, SVD is computed by an in-memory SVD solver if R fits in memory; otherwise the OOM SVD of R must be faster than OOM SVD of A, as R must be much smaller than the original matrix A.

4.2 QR factorization

4.2.1 Introduction

A QR factorization decomposes an $m \times n$ real matrix as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $n \times n$ real upper triangular matrix. To factorize a general matrix A as QR, a series of elementary Householder matrices of the general form $H = I - \tau vv^T$ are applied, where τ is a scaling factor and v is a column reflector. In LAPACK, QR factorization is performed as a blocked algorithm by the $DGEQRF$[4] routine for double precision. The factorization algorithm is a two-step process.

- Panel factorization — for a panel of size nb, nb columns are factored using Householder transformations, and nb elementary Householder matrices are accumulated. The product of the Householder matrices is represented as $H_1H_2 \ldots H_{nb} = I - VTV^T$, where V is an $m \times nb$ matrix in which columns are the vectors v and T is an $nb \times nb$ upper triangular matrix. Panel factorization requires $(\theta(n^2))$ FLOPS, which is a small fraction of the total number of FLOPS$((\theta(n^3)))$ performed for a whole factorization algorithm.

- Update — in the update phase, nb transformations that are accumulated during the panel factorization are applied all at once to the rest of the trailing sub-matrix by Level-3 BLAS operations ($DGEMM$).

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 currently on available shared-memory machines. The execution flow of a block factorization algorithm represents a fork-join model where panel factorization is a sequence of operations interleaved with parallel updates of the trailing sub-matrix. The problem of fork-join bottleneck in block algorithms has been overcome in [11], [13], [53], [43], [44] where panel factorization and trailing submatrix updates are broken into smaller tasks of block size b that can be represented as a DAG. 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 slow, sequential tasks behind fast, parallel ones.

Algorithm 4: Tile QR algorithm

```
for $(k = 0;\ k < min(A.mt, A.nt);\ k = k + 1)$ do
    $A_{kk,kk} \leftarrow GEQRT(A_{k,k})$
    for $(n = k + 1;\ n < A.nt;\ n + +)$ do
        $A_{k,n} \leftarrow UNMQR(A_{k,k}, A_{k,n})$
    end
    for $(m = k + 1;\ m < A.mt;\ m + +)$ do
        $A_{m,k} \leftarrow TSQRT(A_{k,k}, A_{m,k})$
        for $(n = k + 1;\ n < A.nt;\ n + +)$ do
            $A_{m,n} \leftarrow TSMQR(A_{k,n}, A_{m,n}, A_{m,k})$
        end
    end
end
```

4.2.2 Tile QR factorization

High-performance implementation of tile QR factorization is presented in [11], [13] for multicore architecture. The algorithm processes square tile instead of rectangular panel as in an LAPACK blocked algorithm [4]. The tile QR algorithm presented in Algorithm 4 has the following four basic computational kernels:

- 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 of the tile matrix. The DTSQRT routine couples the R factor, produced by DGEQRT or a previous call to DTSQRT, with a tile below the diagonal tile and generates a square matrix V for Householder reflectors, and updates the R factor.
- DORMQR applies the orthogonal transformations computed from DGEQRT to the right of the diagonal tile.
- DTSMQR applies the orthogonal transformations computed from DTSQRT to the right of the tiles factorized by DTSQRT.

The kernels are executed as a task and scheduled by $QUARK$ [56]. Figure 4.1 shows the fingerprint of the first step of the Algorithm 4 for a 3×3 tile matrix.

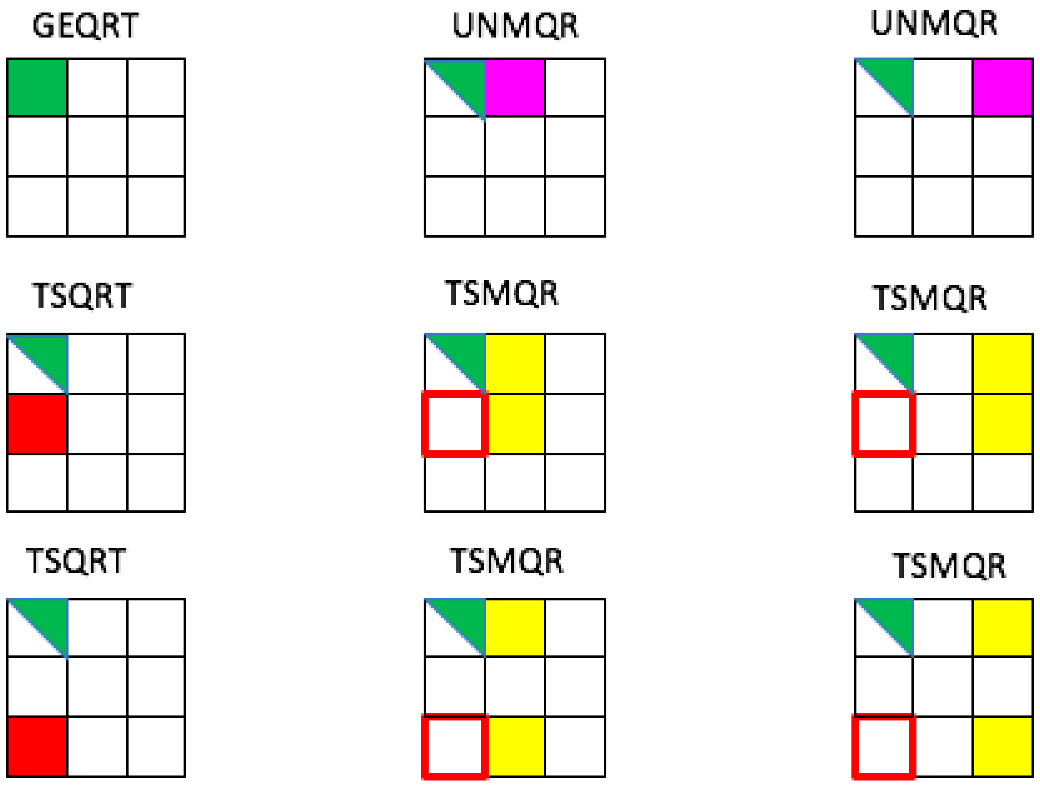


Figure 4.1: Execution of the first step of Algorithm 4.

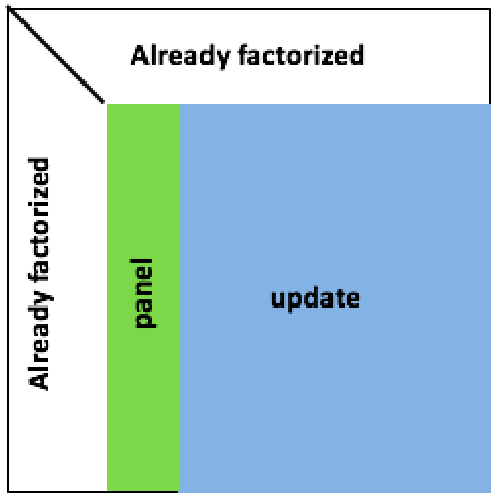
For an $m \times m$ matrix with tile size b and, if $u = \frac{m}{b}$, the number row/column tile in the matrix, Table 4.1 shows the computation cost for the kernels used in tile QR. The most expensive kernel is the update kernel — $TSMQR$ and consumes $O(m^3)$ flop. For all our theoretical analysis, we consider the computation cost of the $TSMQR$ routine.

Table 4.1: Computation cost for tile QR kernel

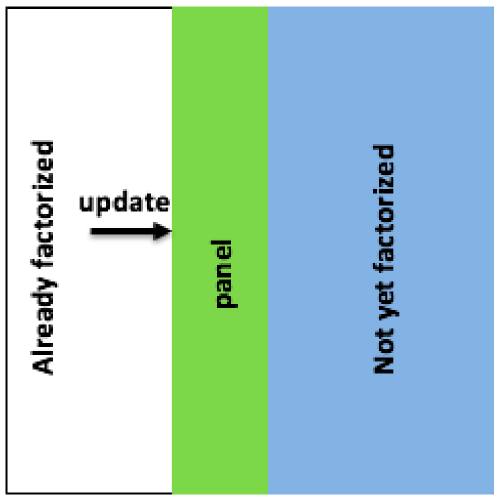
Kernel	Computation cost (flop)	Total cost for $u \times u$ tile matrix
GEQRT	$2b^3$	$O(u) \times 2b^3 = O(m) \times b^2$
ORMQR	$3b^3$	$O(u^2) \times 3b^3 = O(m^2) \times b$
TSQRT	$\frac{10}{3}b^3$	$O(u^2) \times \frac{10}{3}b^3 = O(m^2) \times b$
TSMQR	$5b^3$	$O(u^3) \times 5b^3 = O(m^3)$

4.2.3 Block and tile looking variants

The two main algorithmic variants exist for both the block and tile algorithms explained above — (1) left looking and (2) right looking. They differ only in the location of the update with respect to the panel. The right looking variant operates on the current panel and applies the corresponding updates to the right as shown in Figure 4.2a. 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 applies all updates coming from the left up to the current panel as shown in Figure 4.2b and therefore delays subsequent updates of the remaining parts of the matrix. That is why the left-looking invariant is called the "lazy" variant.



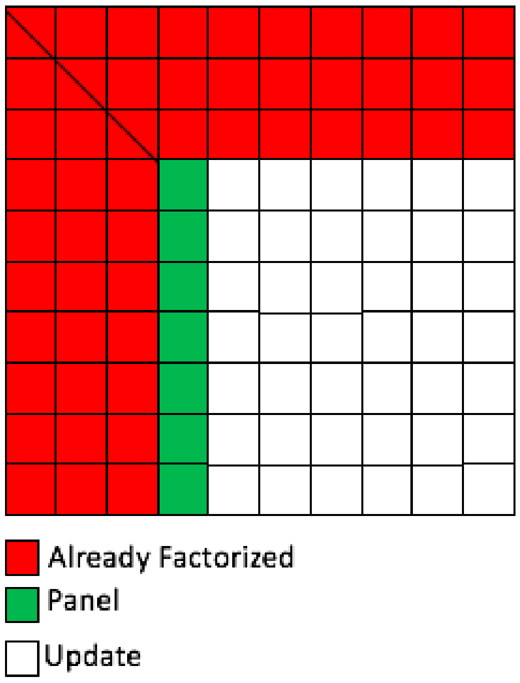
(a) Right-looking block QR decomposition



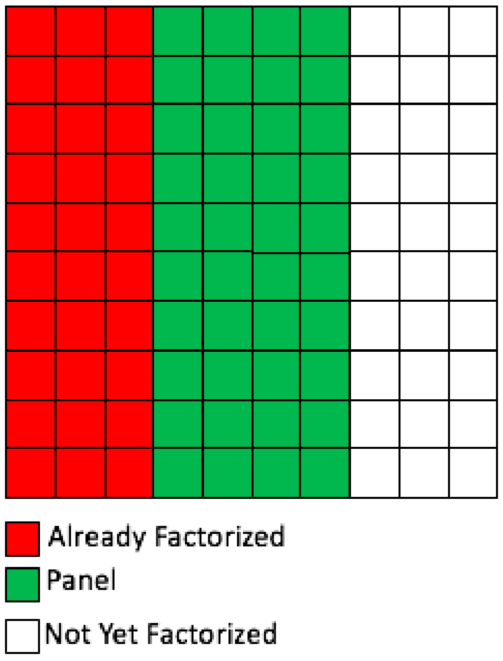
(b) Left-looking block QR decomposition

Figure 4.2: Block QR decomposition — right looking vs. left looking

For tile algorithms the algorithmic principles of the right-looking and the left-looking variants are similar to the block algorithms as shown in Figure 4.3a and Figure 4.3b. All the computations for the panel and the updating of the matrix are now split into tiles. Like blocked algorithms right-looking tile algorithms also need to access the trailing matrix for each panel. Both for tile algorithms and blocked algorithms, the update operations (left-looking or right-looking variants) may run concurrently with the panel operations. noticeably, the right-looking and left-looking variants actually highlight a trade-off between the degree of parallelism (right looking) and data reuse (left looking) and can considerably affect the overall performance.



(a) Right-looking tile QR decomposition



(b) Left-looking tile QR decomposition

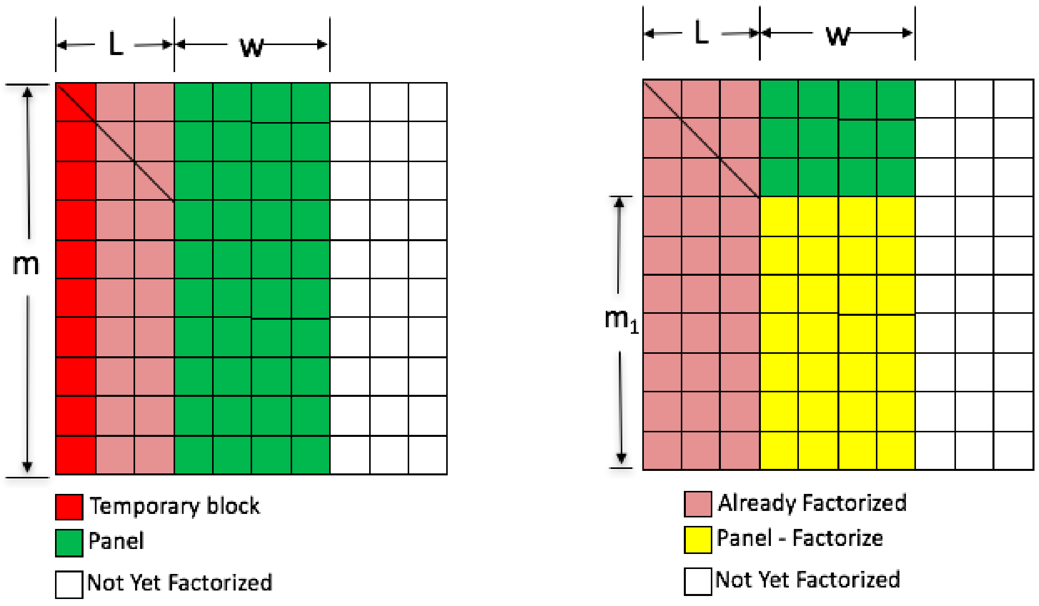
Figure 4.3: Tile QR decomposition — right-looking vs. left-looking.

4.3 OOM QR algorithms

OOM QR factorizes a matrix A as $A = QR$, where the matrix A may not fit entirely in main memory. A left-looking variant is preferable for OOM QR algorithms because it delays the update and avoids the writing of the trailing matrix, contrary to right-looking algorithm, which updates the trailing matrix for every panel it processes and causes huge data traffic on the disk. In OOM left-looking tile algorithms, only two block tile columns may be in memory at any time. One of these is the block updated and factored, which we refer to as the panel. The other is one of the block columns lying to the left of the panel, which we refer to as a temporary block. So, two steps are followed for each panel — (1) In the update step, tiles in the panel are updated by the tiles from the temporary block; (2) In the factorization step, tiles in the panel are factorized after the update. All the tiles of the matrix are brought into the panel once and then sent back to disk to be written after the factorization.

4.3.1 A theoretical study of the design of an OOM QR

The efficiency of OOM QR algorithm depends on its ability to hide the tile read time in the temporary block. As left-looking variant delayed update and applies all at once, we want to investigate the circumstances in which the updating of the panel may be overlapped with the reading of the tiles in the temporary block. 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 in the temporary block as as shown in Figure 4.4a.



(a) Updating the panel.

(b) Factorizing the panel.

Figure 4.4: Left-looking tile QR — updating and factorizing

Theorem 4.1. *For the OOM tile QR algorithm, the updating of a panel of width w, where $w = k \times b$, can be overlapped with the reading of the tile in the temporary block if $w \geq \dfrac{1.6\alpha}{BW}$, where b is the tile size, BW is the communication bandwidth, and α is the computational performance efficiency of the system.*

Proof. Assume the panel is already loaded into memory for update. If u is the number of row tiles, then, $m = u \times b$ and $w = k \times b$. As the update kernel — $TSMQR$ is the most expensive kernels for tile QR algorithm and consumes most of the overall flops as shown in Table 4.1 we are considering the cost of the update kernel only. The computation cost for the update kernel is $5b^3$ flop.

Time to update the panel by tiles from the temporary block, t_{update}, is given by:

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

Time to read the tiles in the temporary block, t_{read} is:

$$\begin{aligned} t_{read} &= \frac{u \times 8b^2}{BW \times 10^6} \\ &= \frac{m \times 8b}{BW} \end{aligned}$$

To overlap update with communication:

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

□

So, if the panel width, w, is at least $\frac{1.6\alpha}{BW}$ the updating of the panel overlaps with the reading of a tile in the temporary block. For the Haswell E5 2650V3 machine, if panel width is 3200, panel updating overlaps with the reading of tiles in the temporary block. One can see that overlapping the updating of a panel with reading depends only on panel width, not panel height. So, we do not need to hold the entire tile column in the temporary block. Two tiles in the temporary block are sufficient to overlap updating with reading.

After updating the panel is factorized. Yellow tiles from the panel as shown in Figure 4.4b are used in factorization step. While factorizing the $m_1 \times w$ submatrix of the panel as shown in Figure 4.4b, we want to investigate whether reading and writing another submatrix of the same size is possible.

Theorem 4.2. *For OOM tile QR, while factorizing a $m_1 \times w$ submatrix, where $w = k \times b$, a sub-matrix of the same size can be communicated (reading and writing) between disk and memory if $w \geq \frac{6.4\alpha}{BW}$, where b is the tile size, BW is the communication bandwidth, and α is the computational performance efficiency of the system.*

Proof. According to [12], computation cost to factorize $m_1 \times w$ sub-matrix is $\frac{5w^2(3m_1-w)}{6}$. So time to factorize the panel, $t_{factorize}$, is given by:

$$t_{factorize} = \frac{5w^2(3m_1 - w)}{6\alpha}$$

If $m_1 \gg w$ then:

$$t_{factorize} \approx \frac{5w^2 m_1}{2\alpha}$$

Time for communication (read and write) of a submatrix of size $m_1 \times w$:

$$t_{read+write} = \frac{16 m_1 w}{BW}$$

To overlap update with read write,

$$\begin{aligned} & t_{factorize} \geq t_{read+write} \\ & => \frac{5w^2 m_1}{2\alpha} \geq \frac{16 m_1 w}{BW} \\ & => w \geq \frac{6.4\alpha}{BW} \end{aligned}$$

□

For the Haswell E5 2650V3 machine, if the panel width is 12800, factorization of a submatrix can overlap with the reading and writing of another submatrix of the same size. But because two of these 12800-wide submatrices are needed, overlapping may not always be possible.

4.3.2 Algorithm design

From theorem 4.1, we already know that panel width should be $\frac{1.6\alpha}{BW}$ to overlap panel updating with the reading of a tile in the temporary block that will be used for updating in next step. In designing the algorithm, our goal is to make panel width large enough so that the updating of the panel overlaps the reading of the left side tiles of the panel. In Table 4.2 we have shown how many times each tile is used for a 10×8 tiled matrix with a panel width of three tiles in both the updating and factorization steps. Tiles from the left side of the panel are read once in the temporary block for each panel update. Tiles in and above the diagonal in the panel are used a different number of times than tiles below the diagonal, because during the updating step all the tiles below the diagonal also update tiles in and above the diagonal. From Table 4.2 we can determine whether we need to hold the entire panel in memory. If tiles in and above the diagonal are held in memory, tiles in the panel below the diagonal can be read row-wise, updated and factorized, and sent back to the disk before next the row is loaded into memory. For this case we need 15 tiles to hold tiles in and above the diagonal, and another 3 tiles for each row below the diagonal, for a total of 18 tiles compare with 30 tiles if the entire panel is loaded into memory. Panel width can be increased by 1 with the 30 tiles that were used to hold the entire panel.

Table 4.2: Number of reads & writes for the OOM QR

1	0	0	10	10	10		
1	1	0	9	9	9		
1	1	1	8	8	8		
1	1	1	7	7	7		
1	1	1	1	6	6		
1	1	1	1	1	5		
1	1	1	1	1	1		
1	1	1	1	1	1		
1	1	1	1	1	1		
1	1	1	1	1	1		

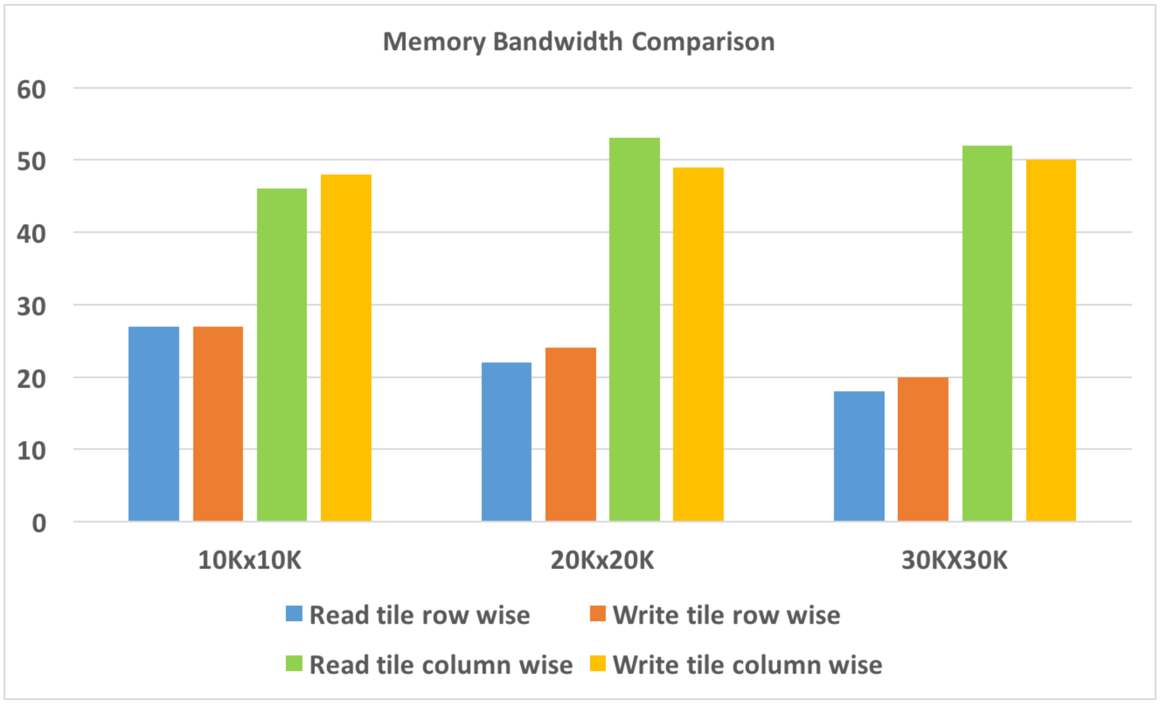


Figure 4.5: Memory bandwidth (Western Digital WDC1002FAEX) — tiles are accessed row-wise vs. column-wise.

The potential danger is that if both tiles below the diagonal, and tiles from the left side of the panel, have to be accessed row-wise, disk bandwidth may be impacted. Before moving forward, we benchmarked read and write bandwidth in Figure 4.5 when tiles are accessed row-and column-wise for Western Digital — the WDC1002FAEX disk. Accessing tiles row-wise gives almost half of the bandwidth that we can achieve accessing them column-wise. To hide overhead for such poor bandwidth, panel width has to be increased, but this might not always be feasible. For peak disk bandwidth, tiles must be accessed column-wise, as they are in consecutive location inside the disk, and so the best way to achieve it is to hold the entire column of tiles and as many of them as possible. If the entire column tiles does not fit in main memory, we will hold as many tiles from the top of the column as possible and keep reading the rest when necessary. This approach will ensure sequential access for tiles not only in the panel but for the tiles on the left side of the panel.

Algorithm 5: OOM QR algorithm

```
for $(k = 0;\ k < A.nt;\ k = k + kb)$ do
    for $(n = k;\ n < min(k + kb, A.nt);\ n++)$ do
        for $(m = 0;\ m < A.mt;\ m++)$ do
            $READ\ \ A_{m,n}$
        end
    end
    $//Update\ A(0:m,\ k:k+kb)\ by\ A(0:m,\ 0:k)$
    for $(n = 0;\ n \leq min(m, k-1);\ n++)$ do
        for $(m = 0;\ m < A.mt;\ m++)$ do
            $READ\ \ A_{m,n}$
            for $(j = k;\ j < (k + kib);\ j++)$ do
                if $(m == n)$ then
                    $A_{m,j} \leftarrow UNMQR(A_{m,n}, A_{m,j})$
                else
                    $A_{m,j} \leftarrow TSMQR(A_{n,j}, A_{m,j}, A_{m,n})$
                end
            end
        end
    end
    $//Factorize\ A(m,\ k:k+kb)$
    for $(kk = k;\ kk < A.mt;\ kk++)$ do
        $A_{kk,kk} \leftarrow GEQRT(A_{kk,kk})$
        for $(n = kk + 1;\ n < min(k + kb, A.nt);\ n++)$ do
            $A_{kk,n} \leftarrow UNMQR(A_{kk,kk}, A_{kk,n})$
        end
        for $(m = kk + 1;\ m < A.mt;\ m++)$ do
            $A_{m,kk} \leftarrow TSQRT(A_{kk,kk}, A_{m,kk})$
            for $(n = kk + 1;\ n < min(k + kb, A.nt);\ n++)$ do
                $A_{m,n} \leftarrow TSMQR(A_{kk,n}, A_{m,n}, A_{m,kk})$
            end
        end
    end
    for $(n = k;\ n < min(k + kb, A.nt);\ n++)$ do
        for $(m = 0;\ m < A.mt;\ m++)$ do
            $WRITE\ \ A_{m,n}$
        end
    end
end
```

Algorithm 5 gives the details of OOM QR implementation and Figure 4.6 and Figure 4.7 illustrate the fingerprint of the dataflow pattern for both the updating and factorization steps. Each panel of the OOM QR algorithm has the following two phases:

- Updating — The panel is updated by all the previous delayed transformations. Figure 4.6 shows the updating of the panel by three column tiles to the left to the panel. Tiles in and below the diagonal are used to update the panel. Updating by a diagonal tile (red) enables all the magenta tiles (the $UNMQR$ task) to be updated in parallel. When updating for the diagonal block ends, nondiagonal tiles (red) from the same column start updating panel tiles in the same row (yellow tiles). All the yellow tasks (the $TSMQR$) for the different columns are now executed in parallel.

- Factorization — A panel is factorized after the it is updated. As all the tiles are loaded into memory in the updating phase, the factorization of the panel is the same as the QR factorization of the red tiles (shown in Figure 4.7) explained in Algorithm 4 in section 4.2.2.

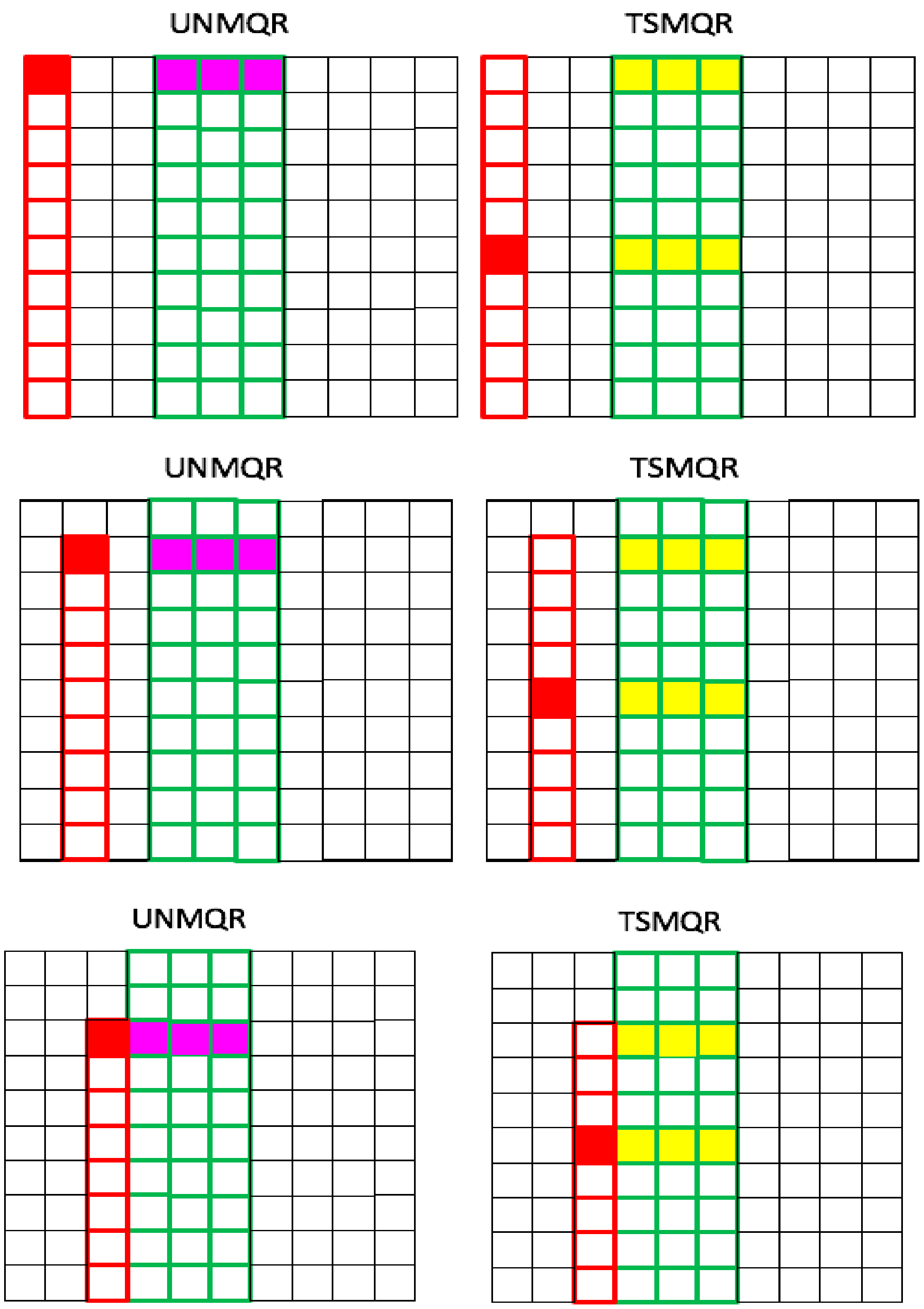


Figure 4.6: Left-looking OOM QR — the updating of the panel.

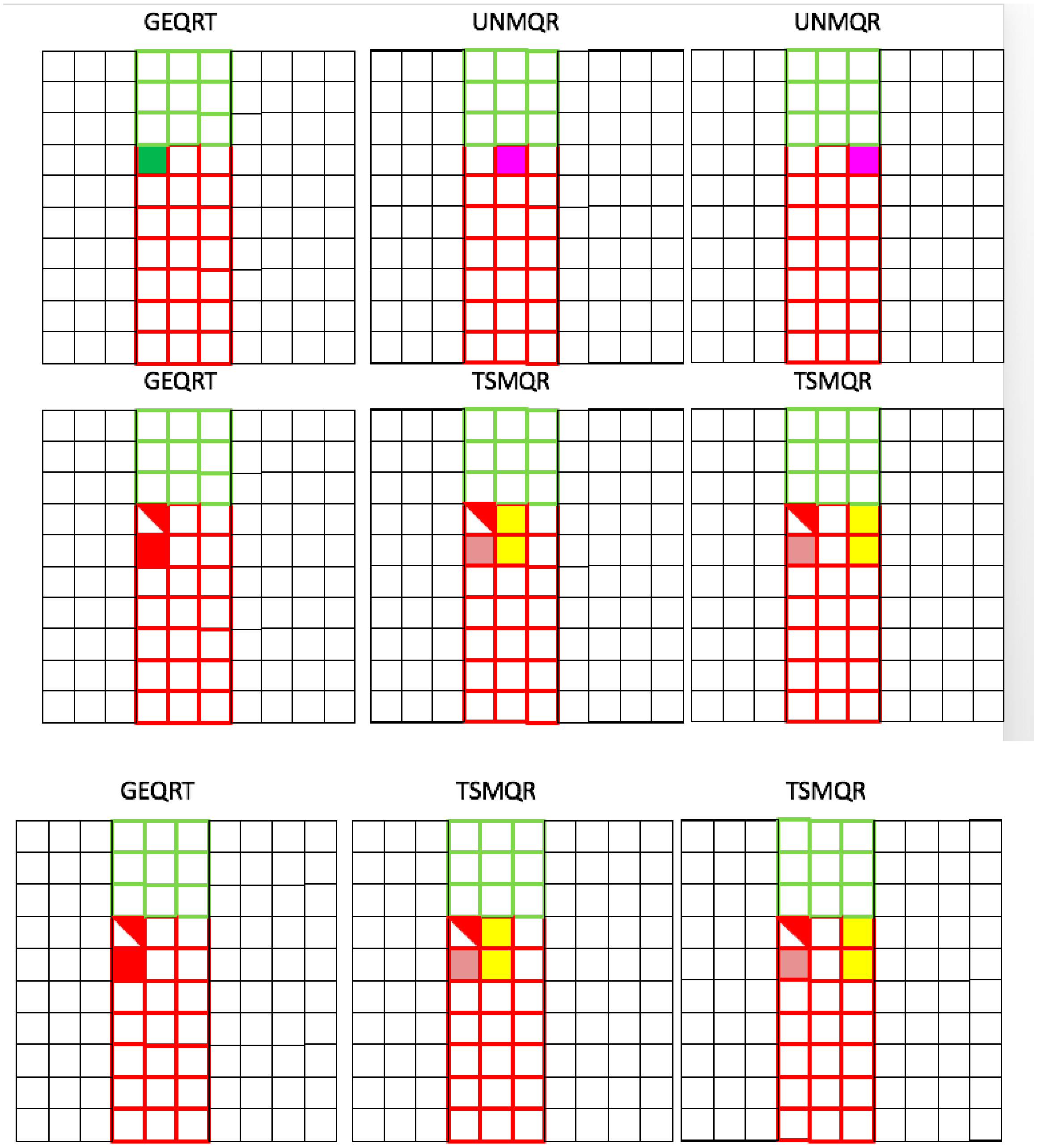


Figure 4.7: Left-looking OOM QR — the factorization of a panel.

4.3.3 Experiment and Results

To evaluate the performance of the OOM QR tile algorithm, we have conducted a number of experiments and collected execution traces to show how execution overlaps with tile reading/writing from/to the disk. Like OOM SVD we have collected traces for small matrices and adjusted tile reading/writing time using the sleep function inside the reading/writing kernel. We collected the execution traces in Sandy Bridge (Xeon E5-2670) machine that has an HDD of 50 MB/s bandwidth. In our traces *green* represents $GEQRT$, *red* represents $TSQRT$, *magenta* represents $GEMQR$, and *yellow* represents the $TSMQR$ routine. Reading and writing tasks are represented by *cyan* and *purple*, respectively, as before.

For a $u \times v$, tile matrix with $u > v$, and a panel width WT tiles, there will be $\lceil v/WT \rceil$ panels. The width of the first panel is $v\%WT$ if v is not completely divisible by WT; otherwise, it will be WT. Making the first panel $v\%WT$ when v is not divisible by WT has a big advantage. Because the first panel does not require updating the reading of left side tiles in the temporary block can be avoided. All other panels with the width $v\%WT$ pay the cost of tile reading, as updating doesn't overlap with reading tiles in the temporary block for panels of that size.

In our first experiment, we use fewer tiles in the panels so that updating does not overlap with reading time. Figure 4.8 shows the execution traces with panels 20 tiles wide, and Figure 4.9 enlarges some sections of the traces of Figure 4.8. As shown in Figure 4.9a, the first panel does not require updating, and factorization can start when the tiles are in memory. For all other panels, the updating step reads the left-side tiles of the panel in the temporary block and uses them to modify tiles within the panel. Figure 4.9b and Figure 4.9c show the execution traces of the updating. Because not enough tiles are in the panel to hide reading time for the tiles on the left side of the panel, threads are waiting for the tasks to execute, and the traces contain so many gaps inside, as shown in Figure 4.9b and Figure 4.9c. After updating, factorization of the panel begins. Since all the tiles used in the factorization step are in memory, enough tasks now exist for all the threads to execute in parallel as shown in Figure 4.9d.

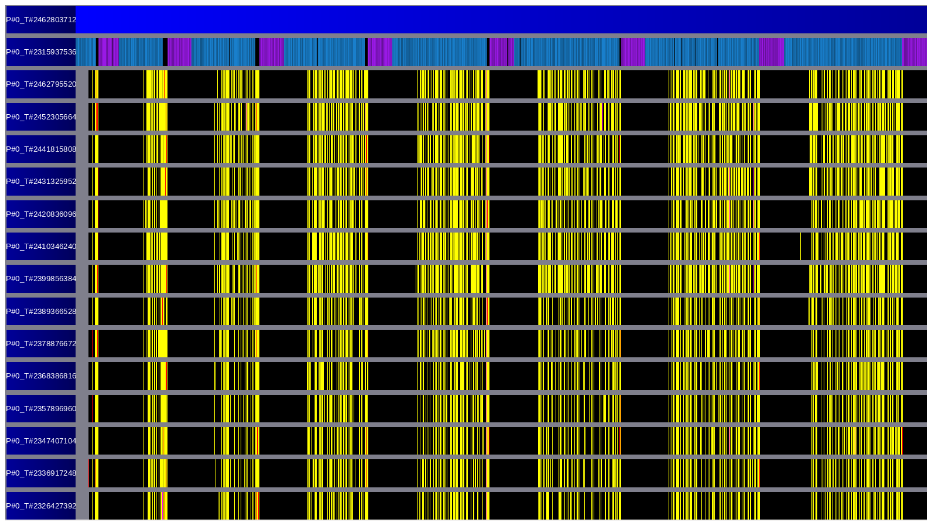
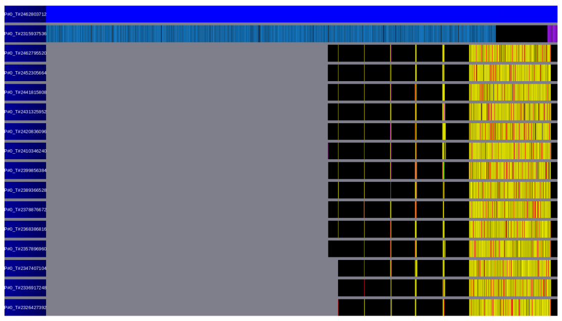
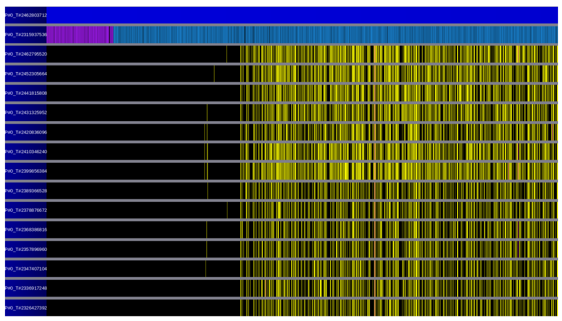


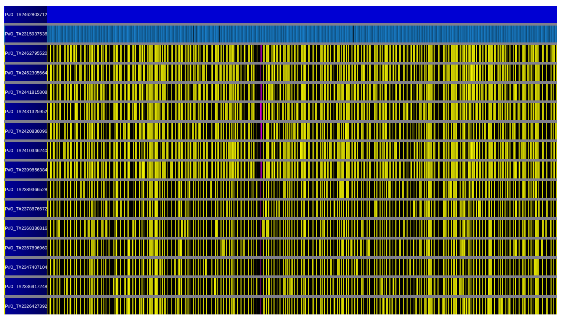
Figure 4.8: Execution traces of OOM QR (panel width: 20 tiles).



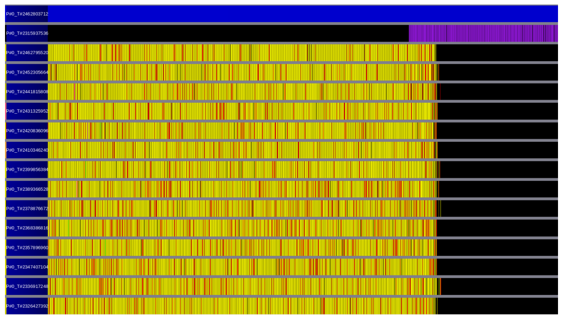
(a) Factorization — first panel.



(b) Updating of a panel.



(c) Updating — overlapping computation with reading.



(d) Factorizing and writing back to the disk.

Figure 4.9: Execution traces of OOM QR (panel width: 20 tile) on a multicore CPU.

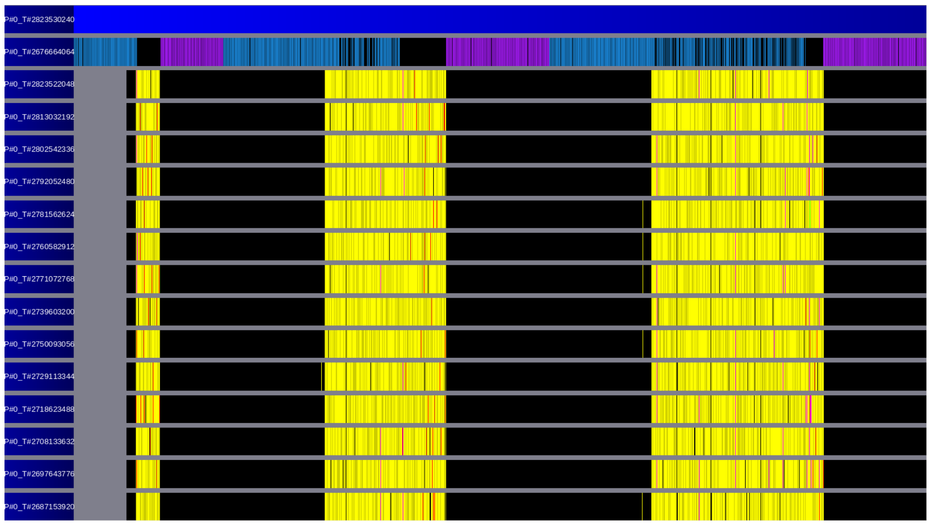
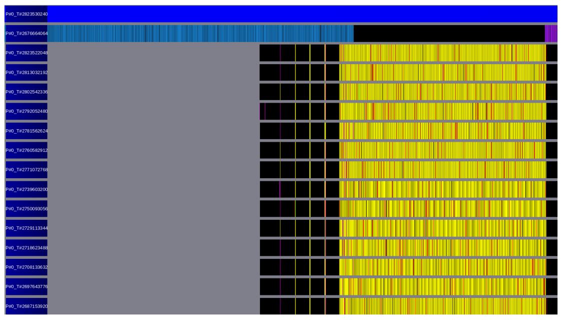
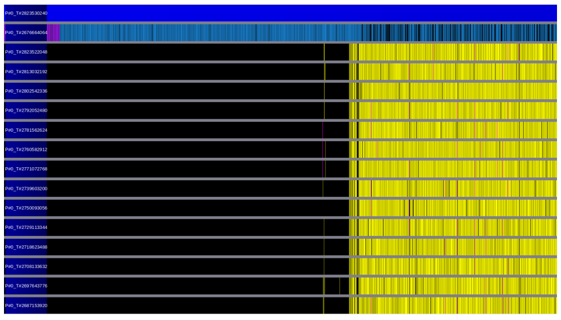


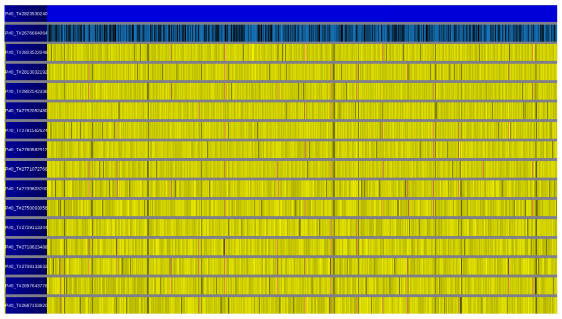
Figure 4.10: Execution traces of OOM QR (panel width: 60 tiles).



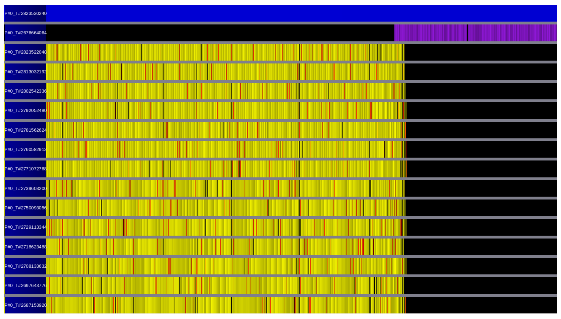
(a) First panel — factorization.



(b) Updating a panel.



(c) Updating — overlapping computation with reading.



(d) Factorizing and writing back to the disk.

Figure 4.11: Execution trace of OOM QR (panel width: 65 tile) on a multicore CPU.

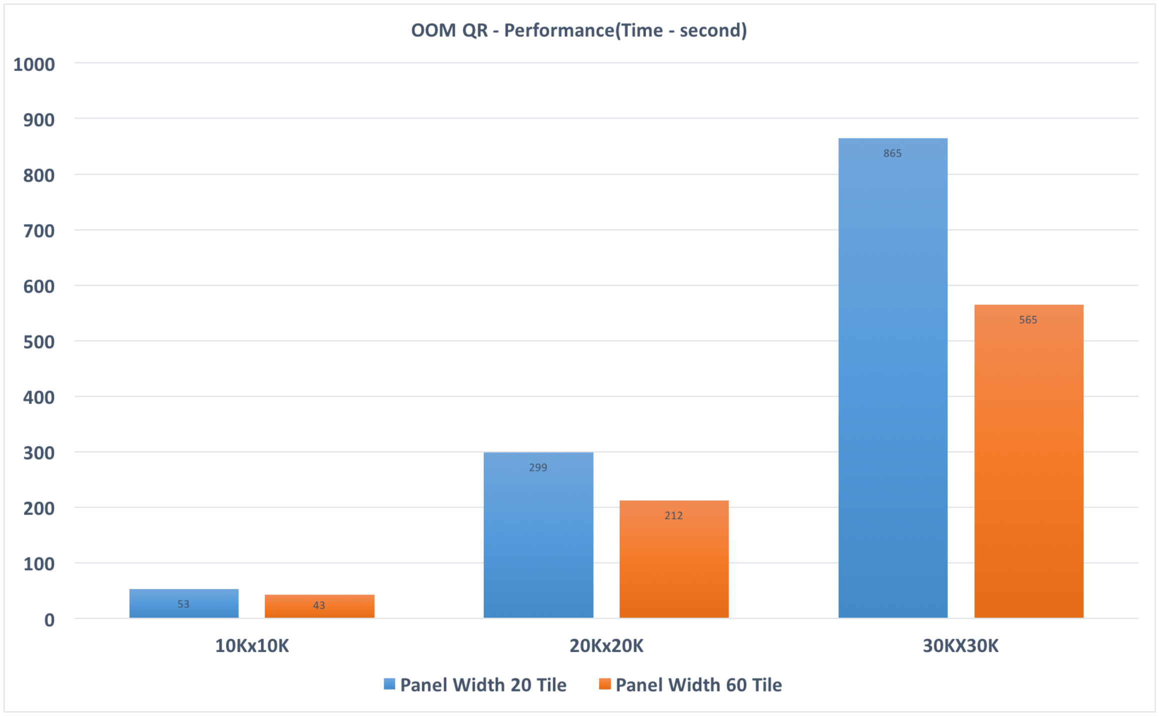


Figure 4.12: Performance of OOM QR with a different panel width.

4.3.4 Challenges and optimization

One can see that in Figure 4.9a, and 4.9b and Figure 4.11a, and 4.11b the execution of tasks is delayed as shown in the traces, even though tiles are in memory. We discovered by studying the QUARK scheduler and analyzing how computation and reading/writing tasks are submitted to the QUARK queue in Algorithm 5. Figure 4.13 shows how reading/writing tasks and computation tasks are submitted to QUARK. Accessing consecutive data from the disk helps to reach achievable disk bandwidth. As tiles are stored column-wise in the disk to achieve maximum bandwidth, tiles have to be accessed column-wise. So ensure proper access, reading/writing tasks are submitted to the QUARK scheduler column-wise fashion. On the other hand, tasks can be run in parallel if they are placed in the scheduler according to their dependency.

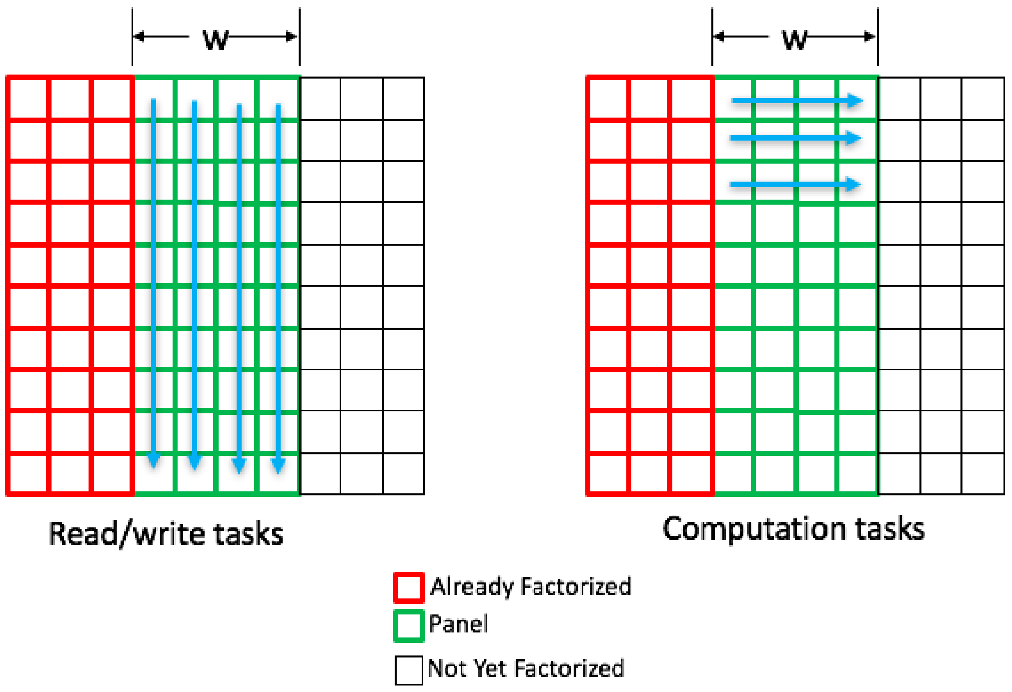


Figure 4.13: Reading/Writing and computation tasks are submitted to the QUARK scheduler.

When reading/writing tasks are submitted in the QUARK queue column-wise to achieve peak disk bandwidth, many reading/writing tasks are in front of computation tasks in the QUARK queue. So, when QUARK starts scheduling tasks from it's queue, it processes reading tasks from the queue and computation tasks wait in the queue for their turn. When the QUARK scheduler processes computation tasks, it resolves the dependency issue and assigns the computation tasks to a thread for execution. As many reading tasks are in front of computation tasks in the QUARK queue, execution of the computation tasks is delayed.

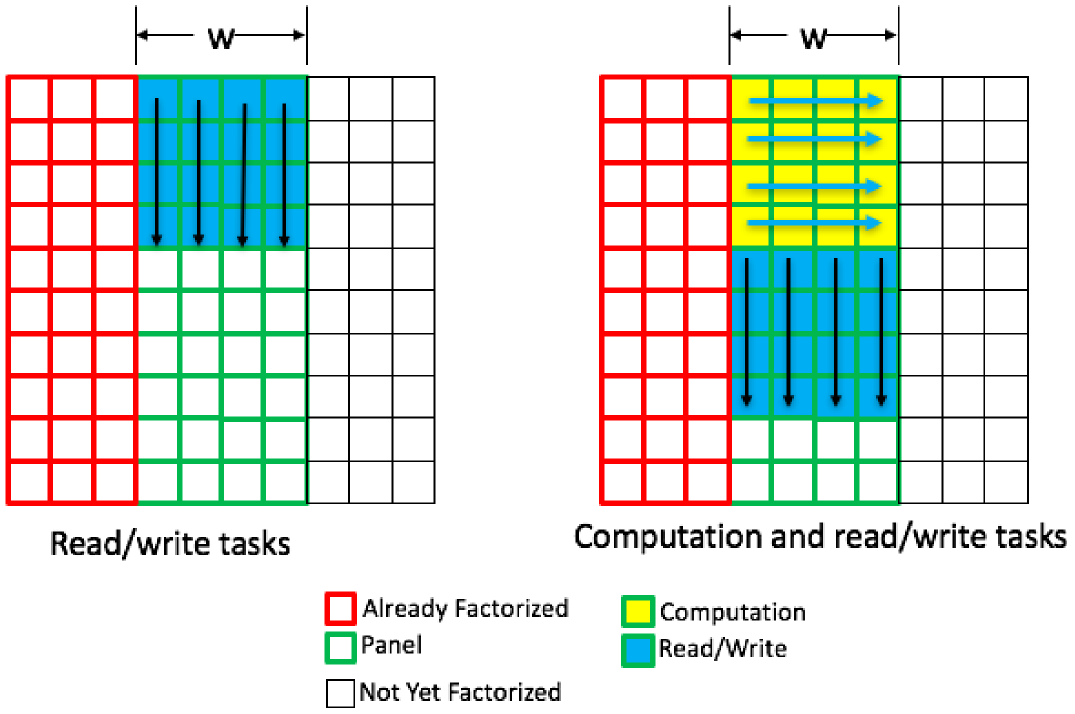


Figure 4.14: Left-looking OOM QR — updating.

To overcome this problem reading/writing tasks and computation tasks are submitted to the QUARK scheduler in such a way so that QUARK can resolve dependency issues for computation tasks and disperse them as early as possible. And simultaneously reading/writing tasks are processed efficiently so that reading/writing bandwidth is not affected by them. So, we start with R row of the panel. After submitting R row of reading/writing tasks column-wise, computation tasks for R are are submitted row-wise to the QUARK queue. Simultaneously, tiles from the left side of the panel are read as they are required to update the panel. When computation tasks are processed read/write tasks for the next block are inserted in the QUARK queue as shown in Figure 4.13 and Figure 4.14. This will reduce number of reading tasks in front of computation tasks in the queue and allows the QUARK scheduler to process computation tasks when they are ready. We have also set the

QUARK_UNROLL_TASKS environmental variable to 5000 so that QUARK can consider a large number of tasks when resolving dependency and dispersing the tasks.

Important to note here is that the entire panel is updated before factorization. But all the tiles in the panel are not used in the factorization step. In Figure 4.15 we have shown which tiles are used for updating and factorization for different panels. Only the orange-colored tiles are used for factorization. So, tiles above the orange-colored tiles are sent back to the disk in column-wise fashion for writing during the factorization step so that writing is overlapped with computation.

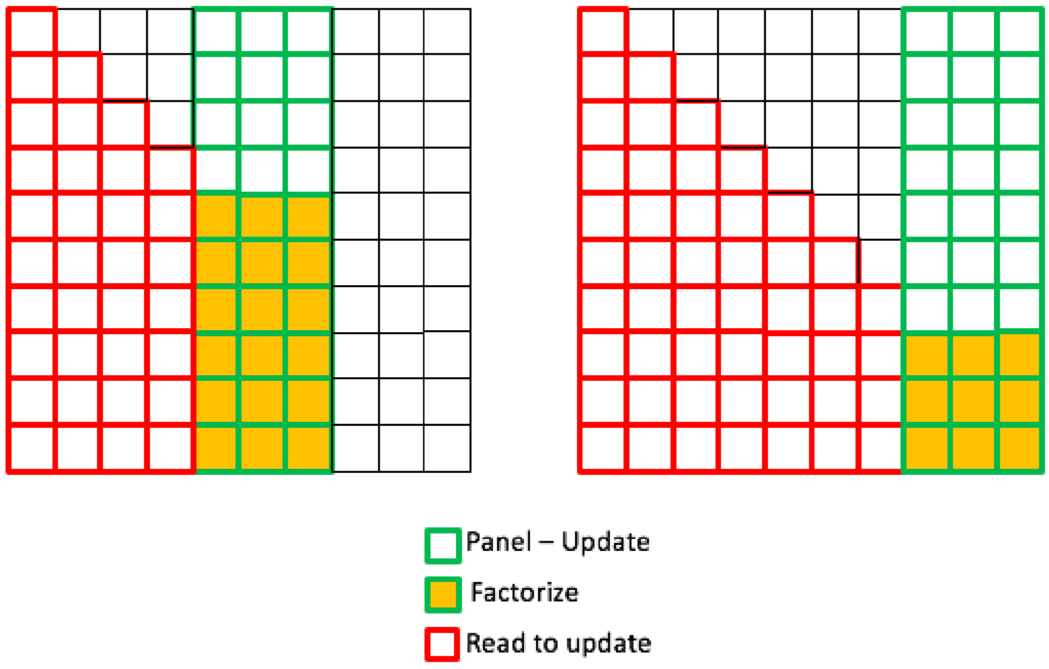


Figure 4.15: Updating and factorization for different panels in OOM QR.

In Figure 4.16 and Figure 4.17, we have shown the traces after adding the above optimization. For the first panel, factorization starts as soon as the tiles are in memory as shown in Figure 4.17a. After factorizing, the fist panel it is written back to disk, and the next panel is loaded into memory, and computation starts immediately as shown in Figure 4.17b. The updating of the the second panel is followed by factorization. As shown in Figure 4.15 some of the tiles from the panel will not be used during the factorization step and can be sent back to disk for writing while factorizing the rest of the tiles as shown in Figure 4.17c. As enough tiles are in the panel, the updating of the panel completely overlapped with reading as shown in Figure 4.17c. For the second panel, not enough tiles exist to write during the factorization step. In Figure 4.17e, we see that some tiles are written back, and when the factorization step completes, tiles that are used during the factorization step are written back. The number of tiles used in the factorization step decreases as the panels of the matrix are processed from left to right. For the last panel, enough tiles might exist to be sent back to the disk and writing time might be overlapped with factorization completely as in Figure 4.17f.

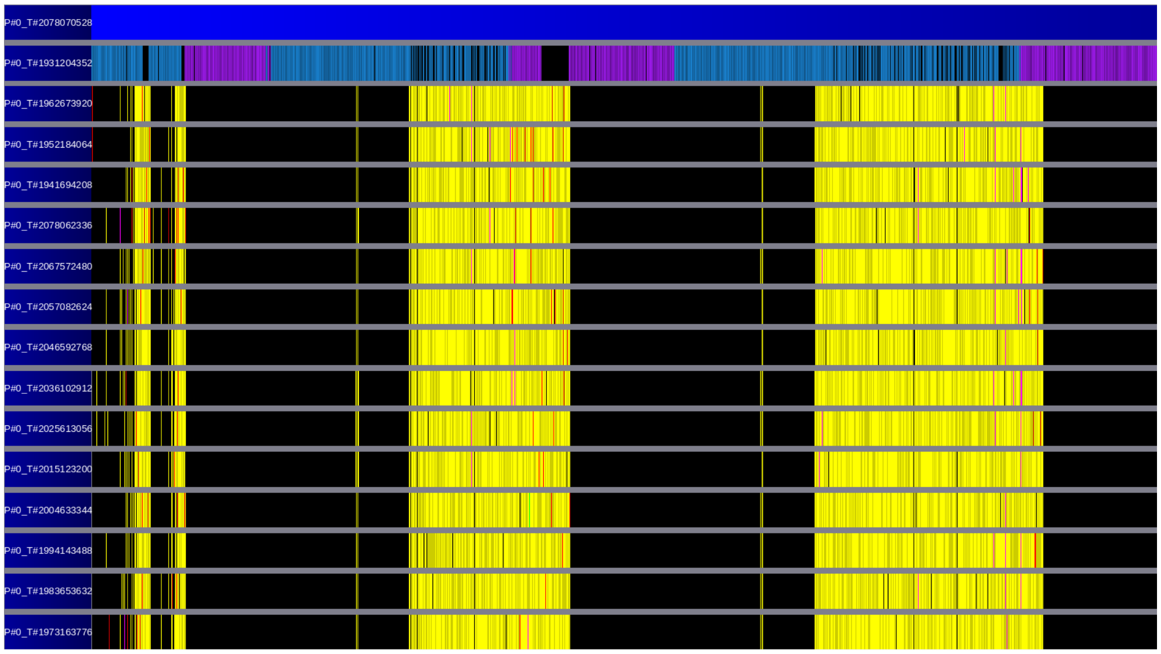
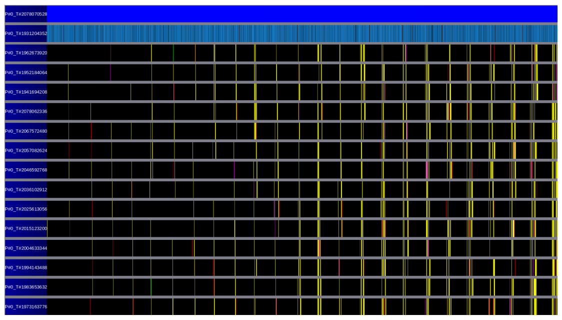
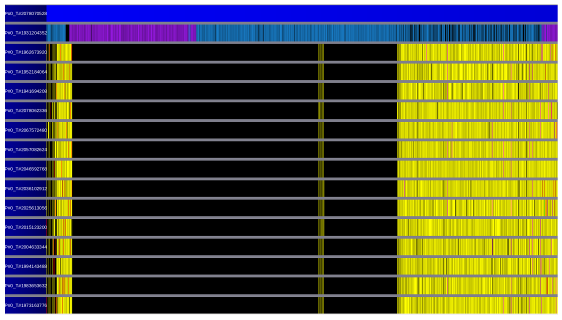


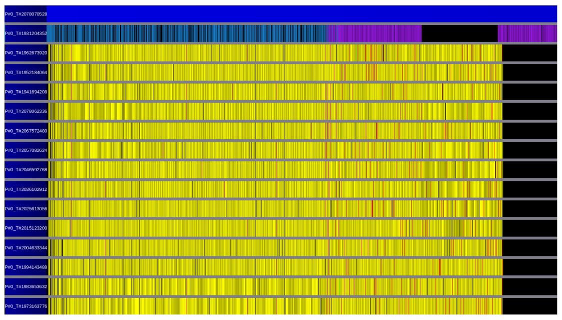
Figure 4.16: Execution traces of optimized OOM QR.



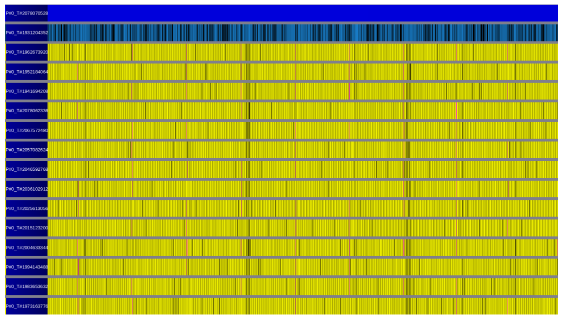
(a) Factorizing the first panel.



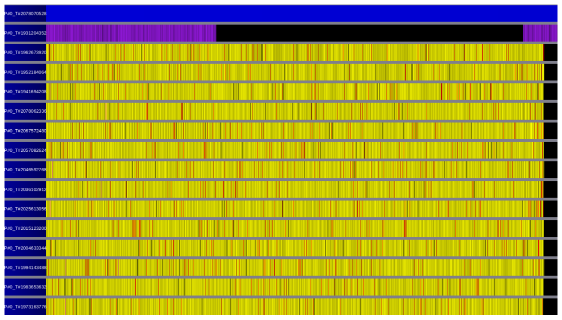
(b) Updating the second panel.



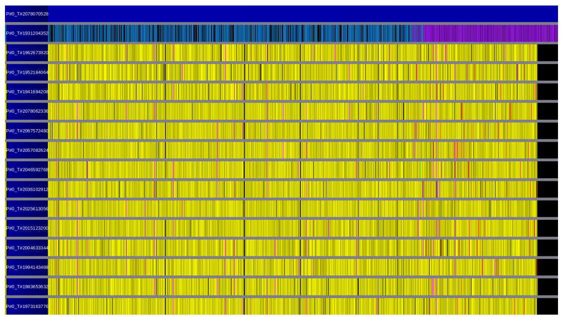
(c) Updating and factorizing the second panel — overlapping computation with reading and writing.



(d) Updating the second panel — the reading of the left side of the panel overlaps with updating.



(e) Factorizating — the updated panel is sent back to the disk.



(f) Updating and factorizing the last panel.

Figure 4.17: Execution traces of optimized OOM QR on a multicore CPU.

We have used 16 threads for OOM QR so far, and 14 threads are used for computation. Next, we overloaded a Sandy Bridge machine with 18 threads so that 16 threads are used for computation. Figure 4.18 shows the performance of OOM QR when all the optimizations are applied.

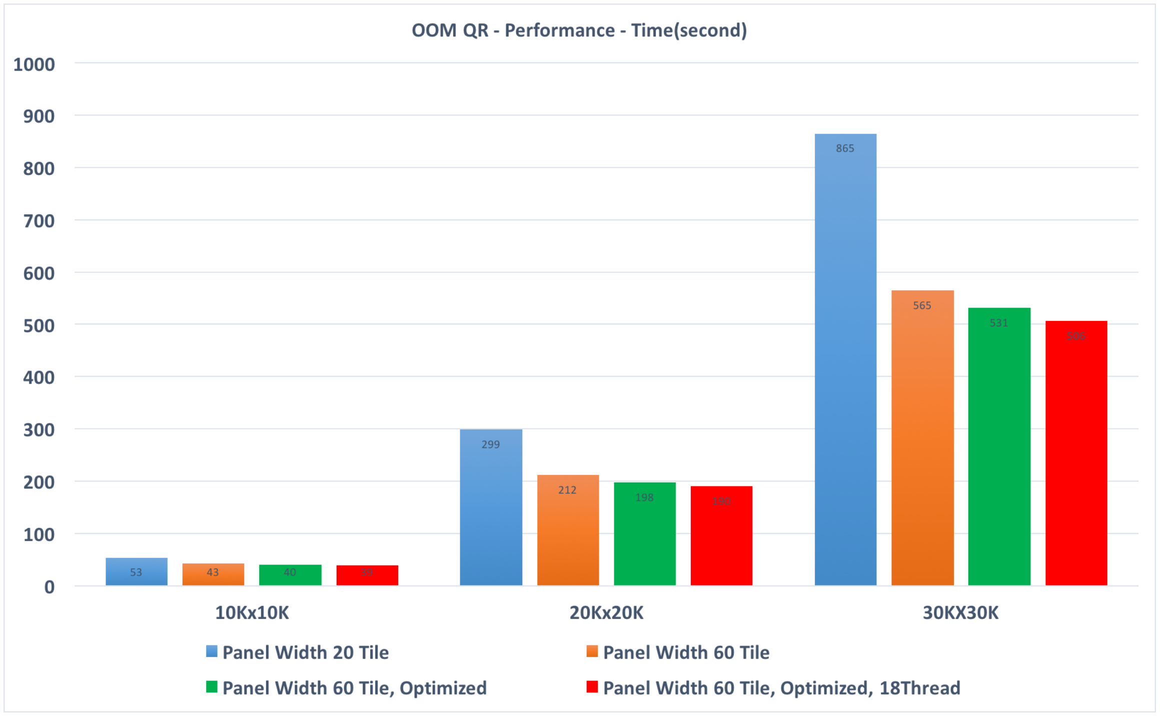


Figure 4.18: Performance of OOM QR after optimization.

The significant time saving comes from the case when the panel width is large enough to overlap computation with the reading of the tiles from the left side of the panel. Other optimizations such as starting the computation and writing back the tiles as soon as possible also helps a lot. And because enough work is ready to be computed when panel width is big enough, using 18 threads decreases the overall runtime.

Table 4.3: Matrix size, disk bandwidth, and QR performance - for extrapolating QR performance

Matrix	Size	Haswell i7-5930K Samsung SSD		Haswell E5 2650V3 Seagate Constellation	
		Disk bandwidth (MB/s)	in-memory QR performance (Gflop/s)	Disk bandwidth (MB/s)	in-memory QR performance (Gflop/s)
100k x 20k	16G	430	170	150	300
100k x 40k	32G	430	170	150	300
100k x 60k	48G	430	170	150	300
100k x 80k	64G	430	170	150	300
100k x 100k	80G	430	170	150	300

4.4 OOM QR Performance

In this section we present performance of OOM QR when the matrix does not fit in main memory. We have run our experiment on both Haswell i7-5930K and Haswell E5 2650 V3 machines. The details of the machines and storage devices are given in Table 2.2 and Table 2.4. Since both the machines had 32 GB of memory, we ran our OOM QR with a matrix size of more than 32 GB. We compare the performance of OOM QR with the extrapolated performance of in-memory QR when the matrix is in main memory. Obviously, we consider the reading/writing time of the matrix, as we have assumed if the matrix is in disk all algorithms read the matrix at the beginning and write back at the end of computation. Table 4.3 shows the size of the matrix for which we did the experiment and the values of disk bandwidth and in-memory QR performance we used to calculate extrapolated performance for matrices that do not fit in memory.

Table 4.4: OOM QR performance(Haswell i7-5930K, Samsung SSD)

Haswell i7-5930K, Memory 32GB Disk - Samsung SSD					
Matrix	Read time(s)	Extrapolated in-memory QR time(s)	Write time(s)	Extrapolated in-memory QR time(s) (with read write)	OOM QR(s)
100k x 40k	74	1631	74	1779	1682
100k x 60k	111	3388	111	3610	3524
100k x 80k	148	5521	148	5817	5682
100k x 100k	186	7843	186	8215	7999

Table 4.4 and Table 4.5 show the performance of our OOM QR for the Haswell i7-5930K and Haswell E5 2650V3 machines. We have also showed in-memory QR performance that we extrapolated using Table 4.3. Performance of OOM QR is close to extrapolated performance of in-memory QR for most the test cases.

Table 4.5: An OOM QR performance (Haswell E5 2650V3, Seagate Constellation)

Haswell E5 2650V3, Memory 32GB Disk - Seagate Constellation					
Matrix	Read time(s)	Extrapolated in-memory QR time(s)	Write time(s)	Extrapolated in-memory QR time(s) (with read write)	OOM QR(s)
100k x 40k	213	924	213	1350	1271
100k x 60k	320	1920	320	2560	2437
100k x 80k	426	3128	426	3980	3859
100k x 100k	533	4444	533	5510	5285

4.5 OOM SVD using OOM QR

In this section we present performance of an OOM SVD solver for tall matrices using OOM QR. In the first step, we factorized the original matrix by OOM QR and then used our two-stage OOM SVD solver for SVD computation. As we mentioned earlier, OOM QR is faster than the OOM two-stage algorithm, as QR factorization delayed the update and does not read/write the whole trailing matrix. Table 4.6 presents the performance of OOM SVD for tall matrices and compares it when OOM SVD solves the problem directly without using OOM QR. For example, to solve a $200k \times 100k$ matrix, two-stage OOM SVD takes 86.12 hours, whereas (OOM QR + OOM SVD) can solve it in 24.35 hours. Table 4.6 also shows the time to solve all the problems using one-stage algorithm. For example, to solve a $200k \times 100k$ matrix, a one-stage approach takes 1405 days compared with 24.35 hours if an OOM QR and a two-stage OOM SVD solver are used to solve the same problem.

Table 4.6: An OOM two-stage algorithm using OOM QR, performance (Haswell E5 2650V3, Seagate Constellation)

Haswell E5 2650V3, Memory 32GB Disk - Seagate Constellation						
Matrix	OOM QR time(h)	Two stage of R(h)		QR of A + Two stage of R(h)	Estimated OOM two stage of A(h)	Extrapolated OOM one stage of A(d)
		First stage(h)	Second stage(h)			
200k x 40k	0.74	0.20	0.023	0.96 hours	7.38 hours	255.88 days
200k x 60k	1.46	0.53	0.533	2.52 hours	25.11 hours	546.51 days
200k x 80k	2.59	4.96	0.962	8.51 hours	51.64 hours	935.59 days
200k x 100k	3.77	19.04	1.535	24.35 hours	86.12 hours	1405.64 days

One can see that our two-step approach for tall matrices can solve big problems that could not be solved before using a one-stage algorithm. As many applications require SVD for tall matrices, our OOM SVD for tall matrices can solve such big problems in a brief time; for example, a $200k \times 100k$ matrix in 24.35 hours.

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

In this research project, we have studied the design of CA algorithms for OOM SVD. When the matrix is too large and does not fit into system memory, traditional SVD algorithms cannot solve those problems. New techniques and algorithms are required to perform the computation while the matrix is out of memory. The growing gap between floating point arithmetic and bandwidth for different memory systems and disks makes designing algorithms that will avoid communication overhead more difficult. Accessing trailing matrix at each step of the SVD algorithm makes it even more challenging.

In Chapter 3 we analyzed communication overhead for both one-stage and two-stage algorithms and the reduction of data movement cost that might be achieved for OOM SVD. We have presented a theoretical analysis to hide communication overhead, provided different strategies to avoid communication cost, and designed and optimized a CA OOM SVD solver. We have done a number of experiments to validate our theoretical analysis and provided the performance of an OOM SVD solver for big matrices to prove the effectiveness of our different strategies for CA OOM SVD. Our two-stage OOM SVD algorithm is capable of solving any big problems that do not fit in system memory and could not be solved before using traditional existing algorithm.

Many applications including web search, gene expression analysis, and so forth require SVD for tall matrices that do not fit in system memory. In Chapter 4 we showed improved performance of OOM SVD when matrices are tall and do not fit in main memory. To design OOM SVD for tall matrices, we have analyzed communication cost for OOM QR and designed an efficient CA OOM tile QR algorithm. Finally, we have improved the performance of OOM SVD using OOM QR for tall matrices. The performance results for big matrices presented in Chapter 4 show the effectiveness of a two-step approach (OOM QR + OOM SVD) and the significant improvement that we achieved over OOM SVD for the original problem.

5.2 Future Directions

In this dissertation we have focused on the design and implementation of an OOM SVD solver for multicore systems. Our future work will involve the following:

- **Compute singular vectors:** We have computed singular values only. We want to extend our work to compute the first few singular vectors or all the left and and right singular vectors if possible. In this case, all the routines that are used to compute left and right singular vectors must be rewritten, as all the transformation matrices that are used to reduce the general matrix to bidiagonal form will be stored in the disk.

- **Improve the performance of the OOM SVD solver:** For the SVD solver, reduction of the general matrix to bidiagonal form is the most expensive part. In two-stage algorithms, the performance of first stage (reduction of the general matrix to band form) can be improved by accessing the tiles that are in consecutive locations on the disk. Instead of holding tiles from the right lower corner of the matrix, tiles from upper part of the matrix can be held in memory to facilitate sequential disk access. A decision program must be integrated to decide which tile to hold and replace the old one when they are not needed in the program for the subsequent step.

- **Implement OOM SVD using GPUs and Coprocessors:** We have covered theoretical analysis for an OOM SVD solver if GPUs and coprocessors are used for computation. In this case, we assume the matrix will be in CPU memory and will be communicated between the CPU and the GPU/coprocessor through PCIe. Our future work will include investigation of kernel performance and the implementation of an SVD solver that will use GPUs and coprocessors.

- **An OOM eigenvalue solver:** Finally, we want to extend our work for an OOM eigenvalue solver for a symmetric matrix when the matrix is too big to fit in main memory.

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009. DOI: 10.1088/1742-6596/180/1/012037. 13

[2] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM. 13

[3] 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/`. 6

[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, 3rd edition, 1999. 73, 75

[5] H. C. Andrews and C. L. Patterson. Singular value decompositions and digital image processing. *IEEE Trans Acoust., Speech, Signal Processing ASSP-24*, 1, February 1976. 1

[6] R. H. Bartels, G. H. Golub, and M. Saunders. Numerical techniques in mathematical programming. In *Nonhnear Programming*, pages 123–176, New York, 1971. Academm Press. 1

[7] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM TOMS*, 26(4):602–616, 2000. 14

[8] N. Bregman, R. Bailey, and C. Chapman. Ghosts in tomography: the effects of poor angular coverage in 2-D seismic traveltime inversion. *Can. J. Explor. Geophys*, 25(1):7–27, 1989. 1

[9] 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. 17

[10] A. Buttari, J. J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. Multithreading for synchronization tolerance in matrix factorization. In *Scientific Discovery through Advanced Computing, SciDAC 2007*, Boston, MA, June 24-28 2007. Journal of Physics: Conference Series 78:012028, IOP Publishing. DOI: 10.1088/1742-6596/78/1/012028. 12

[11] 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. 12, 13, 74, 75

[12] 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. 82

[13] 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. 12, 13, 74, 75

[14] 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. 17

[15] P. Deift, J. W. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular value decomposition and Hamiltonian mechanics. *SIAM J. Numer. Anal.*, 28(5):1463–1516, October 1991. (LAPACK Working Note #11). 6

[16] J. W. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990. (Also LAPACK LAWN #3). 6

[17] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. In *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011. 13

[18] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 33–42, New York, NY, USA, 2011. ACM. 13

[19] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive LU factorization. *Advances in Parallel Computing, Special Issue*, 22:429–436, 2012. ISBN 978-1-61499-040-6 (print); ISBN 978-1-61499-041-3 (online). 13

[20] 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, 14

[21] V. Farra and R. Madariaga. Seismic waveform modeling in heterogeneous media by ray perturbation theory. *Journal of Geophysical Research: Solid Earth*, 92(B3):2697–2712, 1987. 1

[22] V. Fernando and B. Parlett. Accurate singular values and differential qd algorithms. *Numerisch Math.*, 67:191–229, 1994. 6

[23] W. Gansterer, D. Kvasnicka, and C. 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. 14

[24] G. H. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *J. Soc. Indust. Appl. Math. Ser. B Numer. Anal.*, 2:205–224, 1965. 6

[25] G. H. Golub and W. Kahan. Calculating the singular values and pseudoinverse of a matrix. *SIAM J. Numer. Anal.*, 2(3):205–224, 1965. 1

[26] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944. 2

[27] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. In J. Wilkinson and C. Reinsch, editors, *Handbook for Automattc Computation, II, Linear Algebra*. Springer-Verlag, New York, 1971. 1

[28] G. H. Golub and J. H. Wilkinson. Ill-conditioned eigensystems and the computation of the Jordan canonical form. *SIAM Rev.*, 18(4), October 1976. 1

[29] M. Gu and S. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Mat. Anal. Appl.*, 16:79–92, 1995. 6

[30] F. G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In *Proceedings of the IFIP WG 2.5 Working Conference on Software Architectures for Scientific Computing Applications*, pages 211–234, Ottawa, Canada, October 2-4 2000. Kluwer Academic Publishers. ISBN: 0792373391. 13

[31] F. G. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Trans. Math. Soft.*, 38(3):article 17, 2012. DOI: 10.1145/2168773.2168775. 13

[32] A. Haidar, J. Kurzak, and P. 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. 18, 29

[33] A. Haidar, H. Ltaief, and J. 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. 7, 13, 18, 20

[34] A. Haidar, H. Ltaief, P. Luszczek, and J. 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. 13

[35] A. Haidar, H. Ltaief, A. YarKhan, and J. J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency Computat.: Pract. Exper.*, 2011. DOI: 10.1002/cpe.1829. 12, 13

[36] 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). 14

[37] R. J. Hanson. A numerical method for solving Fredholm integral equations of the first kind using singular values. *SIAM J. Numer. Anal.*, 8(3):616–626, 1971. 1

[38] H. Hotelling. Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.*, 24:417–441, 498–520, 1933. 1

[39] H. Hotelling. Simplified calculation of principal components. *Psychometrica*, 1:27–35, 1935. 1

[40] E. R. Jessup and D. Sorensen. A Parallel Algorithm for Computing the Singular Value Decomposition of a Matrix. *SIAM J. Matrix Anal. Appl.*, 15:530–548, 1994. 6

[41] E. P. Jiang and M. W. Berry. Information filtering using the Riemannian SVD (R-SVD). In A. Ferreira, J. D. P. Rolim, H. D. Simon, and S.-H. Teng, editors, *Solving Irregularly Structured Problems in Parallel, 5th International Symposium, IRREGULAR 98, Berkeley, California, USA, August 9-11, 1998, Proceedings*, volume 1457 of *Lecture Notes in Computer Science*, pages 386–395. Springer, 1998. 1

[42] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. DOI: TPDS.2007.70813. 12

[43] 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. 74

[44] J. Kurzak and J. Dongarra. QR Factorization for the CELL Processor. LAPACK Working Note 201, May 2008. 74

[45] J. Kurzak and J. J. Dongarra. QR factorization for the CELL processor. *Scientific Programming*, 00:1–12, 2008. DOI: 10.3233/SPR-2008-0268. 12

[46] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency Computat.: Pract. Exper.*, 21(1):15–44, 2009. DOI: 10.1002/cpe.1467. 12, 13

[47] C. Lawson and R. Hanson. *Solving Least Squares Problems.* Prentice-Hall, Englewood Cliffs, N.J., 1974. 1

[48] 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. 13

[49] H. Ltaief, P. Luszczek, and J. Dongarra. High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. *ACM TOMS*, 39(3), 2013. In publication. 2, 13

[50] H. Ltaief, P. Luszczek, A. Haidar, and J. 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. 13

[51] P. Luszczek, H. Ltaief, and J. Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In *IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, USA, May 16-20 2011. 13, 20

[52] B. C. Moore. Principal component analysis in linear systems: Controllability, observability, and model reduction. *IEEE Transactions on Automatic Control*, AC-26(1), February 1981. 1

[53] 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. 74

[54] G. W. Stewart. The decompositional approach to matrix computation. *Computing in Science & Engineering*, 2(1):50–59, Jan/Feb 2000. ISSN: 1521-9615; DOI 10.1109/5992.814658. 2

[55] University of Tennessee Knoxville. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*, November 2010. 13

[56] 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. 13, 75

Vita

Khairul Kabir was born in Tangail, Bangladesh. He obtained his bachelor's degree in computer science and engineering(CSE) from Bangladesh University of Engineering and Technology (BUET) and received his master's degree in computer science from University of Tennessee, Knoxville. In July of 2011, Khairul started work as a graduate research assistant at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville. Khairul Kabir is expected to graduate with a doctor of philosophy degree in computer science in May 2017.