Tightening I/O Lower Bounds through the Hourglass Dependency Pattern

Lionel Eyraud-Dubois, Guillaume Iooss, Julien Langou, Fabrice Rastello

Tightening I/O Lower Bounds through the Hourglass Dependency Pattern

Lionel Eyraud-Dubois* Guillaume Iooss† Julien Langou‡
Fabrice Rastello§

April 2024

Abstract

When designing an algorithm, one cares about arithmetic/computational complexity, but data movement (I/O) complexity plays an increasingly important role that highly impacts performance and energy consumption. For a given algorithm and a given I/O model, scheduling strategies such as loop tiling can reduce the required I/O down to a limit, called the I/O complexity, inherent to the algorithm itself.

The objective of I/O complexity analysis is to compute, for a given program, its minimal I/O requirement among all valid schedules. We consider a sequential execution model with two memories, an infinite one, and a small one of size S on which the computations retrieve and produce data. The I/O is the number of reads and writes between the two memories.

We identify a common "*hourglass pattern*" in the dependency graphs of several common linear algebra kernels. Using the properties of this pattern, we mathematically prove tighter lower bounds on their I/O complexity, which improves the previous state-of-the-art bound by a parametric ratio. This proof was integrated inside the IOLB automatic lower bound derivation tool.

Keywords: I/O complexity, Data Movement Lower Bound

1 Introduction

When designing an algorithm, we usually reason about its computational complexity, to estimate the increase in its execution time, when its problem

*Inria, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, lionel.eyraud-dubois@inria.fr

†Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, guillaume.iooss@inria.fr

‡University of Colorado Denver, USA, Centre Inria de Lyon, France, julien.langou@ucdenver.edu

§Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, fabrice.rastello@inria.fr

sizes increase. However, the amount of computation is not the only factor when estimating the performance of an algorithm. The data movement (I/O) also plays an increasingly important role in both performance and the energy consumed by an algorithm.

In order to minimize the amount of data movement of an algorithm, we modify its schedule by using program transformations, such as the tiling transformation [12]. However, finding an optimal schedule is not trivial, due to the size and complexity of the optimization space. The notion of *I/O complexity* of an algorithm indicates the minimal amount of data movement required for any valid schedule of an algorithm. It provides an indication on how far it is possible to optimize the I/O of an algorithm.

However, due to the number of possible valid schedules, it is not possible to directly compute the I/O complexity. Instead, we search for a lower and upper bound on the minimal amount of data movement. To find an upper bound, one only needs to exhibit a valid schedule and compute its I/O cost. However, finding a lower bound requires us to reason over all possible schedules, which requires a mathematical proof based on the data dependency graph of the program.

Several proof techniques exist for deriving lower bounds, such as the wavefront [10] or the K-partitioning technique [11]. Depending on the shape of the dependencies, some of these techniques might work better than the others: for example, the wavefront technique is usually the most effective for stencil-like computation, while the K-partitioning technique is usually better for linear-algebraic computation. Other algorithms might require a specialized proof to obtain an asymptotically tight bound (e.g., for the SYRK kernel [4]). However, there are still some kernels for which the ratio between their proven lower bound and their best upper bound is parametric. For example, such a ratio is described in Table 1 of [17] for the kernels of the Polybench benchmark suite [18].

Some of the proof techniques mentioned above have been automatized and implemented in automatic data movement lower bound derivation tools, such as IOLB [17], so that they can be applied to any kernel given as an input.

Contributions In this paper, we consider several important linear algebra kernels whose dependence graph exhibits a dependency pattern called the *hourglass pattern*. We propose a new proof technique that uses the properties of this pattern to improve the lower bound on the minimal data movement required by these kernels.

In more detail:

- We define the *hourglass pattern*, a pattern of the dependencies of a program, and present its properties.

- We provide a lower bound derivation proof, based on an adaptation of the K-partitioning technique, that tighten the lower bound of a program exhibiting an hourglass pattern. This proof has been fully automatized, inside the tool IOLB [17, 16].
- We present new data movement lower bounds for several linear algebra kernels: Modified GramSchmidt (MGS), QR Householder (GEQR2 and ORG2R in the LAPACK [15] library; also called A2V and V2Q), bidiagonal matrix reduction (GEBD2) and Hessenberg matrix reduction (GEHD2). For all of these kernels, their asymptotic bound was improved by a parametric factor, compared to the bound obtainable by the classical K-partitioning technique. Figure 4 and Figure 5 summarize the new lower bounds found for several linear algebra kernels.
- We also provide tiled orderings for MGS and Householder, resulting in upper bounds that asymptotically match these new lower bounds. This proves the optimality of the new I/O lower bounds.

Outline In Section 2, we provide some background on the I/O complexity and introduce the K-partitioning method for deriving a lower bound on the minimal amount of data movement of a computation. In Section 3, we present the *hourglass pattern*, a pattern of dependencies whose properties can be used to improve the derived lower bound. In Section 4, we show how to exploit the hourglass pattern to adapt the K-partitioning method, in order to obtain a tighter bound. In Section 5, we list different linear algebra kernels exhibiting an hourglass pattern, and their associated improved lower bound. Additionally, Annex A contains the tiled algorithm for two of these kernels. The data movement of these algorithms provides an upper bound to the minimal amount of data movement, which matches asymptotically their new lower bound.

2 Background - I/O complexity and the K-partitioning method

In this section, we present a state-of-the-art method of proof – the K-partitioning method – which infers a lower bound on the amount of data movement needed by a computation.

Memory model and I/O complexity We consider a simple two-level memory model, composed of (1) a slow memory of unbounded size, and (2) a fast memory of size S. Both memories can transmit data from one to another, as long as the constraint on the size of the small memory is satisfied. When we perform an operation, the data used must be present in

the small memory, and the data produced must be committed in the small memory.

We consider a program, which performs a collection of operations organized in *statements*. Each statement SX has multiple *instances* $SX[\vec{i}]$, where $\vec{i}$ is a vector of the surrounding loop indexes.

In this paper, we consider a subclass of programs called *polyhedron* (or affine) *programs*. These programs are combinations of nested loops and statements such as: (i) the loop bounds are affine constraints using the surrounding loop indexes and the program parameters (e.g., the sizes of an input array); (ii) the array accesses are affine expression of the surrounding loop indexes and program parameters. All the programs presented in this paper, such as Figure 3, are polyhedral programs. Also, we will call a quantity "parametric" when it is a function of the parameters of the program, which are considered as symbolic constants.

A statement instance might depend on the data produced by another instance, for example when the first instance uses a value produced by the second instance. We call this a *dependency* between these two statement instances. These dependencies impose constraints on the order of execution of the program. This order of execution is called the *schedule* or the *ordering*. Frequently, the data consumed/produced by the statements of a program are too big to fit all at once in the small memory, it it thus necessary to *spill*, i.e., to transfer some data back into the slow memory and retrieve it later when needed. However, doing so increases the amount of data movement between both memories.

Given a valid schedule for a program, the *I/O cost* for this program and for this ordering is the amount of data movement required, i.e. the number of data transfers between both memories. The *I/O complexity* of a program is the minimal I/O cost that can be reached by any valid schedule. This quantity is interesting, in particular in the context of an architecture where the transfer of data is the limiting factor for performance. Knowing the minimal amount of data transfers is a good algorithmic indicator to know if it could be theoretically optimized further. However, because we need to find the minimal I/O cost *for all possible orderings*, the exact I/O complexity is hard to evaluate. Instead, we rely on bounds on the I/O complexity of a program: we can provide a mathematical proof for the lower bound, and exhibit an ordering (i.e., an implementation of a program) that reaches an I/O cost and provides an upper bound.

CDAG and red-white pebble game When trying to prove a lower bound, one should decide whether redundant computation is allowed or not. The *red-white pebble game*, a variation of the red-blue pebble game of Hong and Kung [11], was introduced by Olivry et al. [17] in order to model the state of the memories during the execution of a program *without*

recomputation, which matched the assumptions we make in this paper.

This game is played on the *Computational Directed Acyclic Graph* (CDAG) of the program. This is a directed graph G, where

- the nodes V represent the computation (statement instance) of a program, and
- the edges represent the flow dependencies between the computations of the program.

Notice that the inputs of a program are nodes that do not have incoming edges. The outputs of a program are a subset of nodes $O \subset V$; they might have outgoing edges.

During a *red-white pebble game*, red and white pebbles are placed on the nodes of a CDAG. A *white pebble* represents a computation that was performed, and a *red pebble* represents a computation whose output is currently stored in the small memory. A game follows this set of rules:

- At the start, the only pebbles in the CDAG are white pebbles, placed on the inputs of the program.
- At most S red pebbles can be simultaneously present on the nodes of the CDAG.
- **Spill:** a red pebble can be removed from a node.
- **Compute:** When a node does not have a white pebble, but all its predecessors have red pebbles, then we can place both a white and a red pebble on it.
- **Load:** A red pebble can be added to nodes with a white pebble.
- The game ends when each node has a white pebble on it.

Notice that once a white pebble is placed on a node, it cannot be removed. This prevents recomputation. In order to compute the amount of data movements, we focus on the number of red pebbles added with the **Load** rule during a game. This means that we only focus on the "Load" portion of the data movements and ignore its "Store" part. The resulting bounds are still valid, and because the number of "Load" often dominates the number of "Store", their tightness should not be strongly impacted. This assumption is identical to the one made in [17].

K-partitioning method The *K-partitioning* method introduced in the seminal paper of Hong and Kunk [11] is a proof technique that allows to derive a lower bound.

The first idea is to consider a partition of the CDAG and games that play on each set of the partition one by one.

Then, we consider the notion of *K-bounded set*. An *inset* of a set E of nodes of the CDAG, noted $InSet(E)$, is the set of data used by E but not produced by a computation of E. A K-bounded set is a set of nodes E of the CDAG whose inset has a size at most K: $|InSet(E)| \leq K$. This notion is interesting because an $(S+T)$-bounded set requires at least T additional data to fit in the small memory (of size S). Thus, even if the small memory is filled with interesting data, we will need at least T load operations to perform the computations of this set. In addition, we assume that our K-bounded sets are *convex*: if there is a dependency chain between two points of a K-bounded set E, then all the intermediate points must belong to E.

Finally, a *K-partition* is a partition into convex K-bounded sets. The idea is to consider all K-partitions of a CDAG and to count how many sets are in this partition. By choosing $K = S + T$, we know that there will be at least T loads per set of the partition. Therefore, a lower bound on the number of loads is T times the minimal number of sets in such a partition.

Theorem 1 ($(S+T)$-partitioning I/O lower bound [9])**.** *Let S be the size of the small memory, and for any $T > 0$ let U be the maximal size of a $(S+T)$-partition. Let V be the set of nodes of the CDAG of the program. Then, a lower bound on the number Q of data movement of the program is:*

$$T.\left\lfloor \frac{|V|}{U} \right\rfloor \leq Q$$

Then, we pick a value of T (which means a value of K) that leads to the tightest lower bound.

To estimate the minimal number of sets in a K-partition, we can estimate the maximum size of a set inside this partition. In other words, an upper bound on the size of a K-bounded set can be transformed into a lower bound on the amount of data movement required.

Upper bound on the size of a K-bounded set An upper bound on the size of a K-bounded set E can be obtained by analyzing the dependencies of the program. Indeed, for a polyhedral program, dependencies between its statement instances are associated with affine relations, matching the loop indices of the data producing instance with the data consuming instance. When examining the path of affine dependencies starting from any node of E to a node of the inset of E, we can either obtain a projection or a translation. In both cases, the image of E through these affine functions ϕ can be mapped to geometrical borders or projections of E, and can be associated with parts of $InSet(E)$. This is the key geometrical intuition that leads us to use the Brascamp-Lieb theorem.

The Brascamp-Lieb theorem is a geometrical way to bound the volume of a set by the volume of its projections, which can be bounded by K.

Theorem 2 (Brascamp-Lieb theorem [6]). *Let d and d_j be non-negative integers and $\phi_j : \mathbb{Z}^d \mapsto \mathbb{Z}^{d_j}$ be a collection of group homomorphisms for all $1 \leq j \leq m$.*

If we have a collection of coefficients $s_j \in [0, 1]$ such that, for any subgroup $\mathcal{H} \subset \mathbb{Z}^d$:

$$rank(\mathcal{H}) \leq \sum_{j=1}^{m} s_j \times rank(\phi_j(\mathcal{H})).$$

Then, for any non-empty finite set $E \subset \mathbb{Z}^d$:

$$|E| \leq \prod_{j=1}^{m} |\phi_j(E)|^{s_j}.$$

For example, if we consider a 3D set and the 3 canonical projections $\phi_{j,k}(i, j, k) = (j, k)$, $\phi_{i,k}(i, j, k) = (i, k)$ and $\phi_{i,j}(i, j, k) = (i, j)$, this theorem gives us the following inequality between the volume of E and the area of its faces $\phi_x(E)$:

$$|E| \leq |\phi_{j,k}(E)|^{1/2} \times |\phi_{i,k}(E)|^{1/2} \times |\phi_{i,j}(E)|^{1/2}.$$

As another example, we could consider instead the projections $\phi_i(i, j, k) = (i)$, $\phi_j(i, j, k) = (j)$ and $\phi_k(i, j, k) = (k)$, to obtain the following inequality:

$$|E| \leq |\phi_i(E)| \times |\phi_j(E)| \times |\phi_k(E)|.$$

In the case of a K-bounded set, we consider the path of dependencies to automatically derive these projections $\phi \in \Phi$. So, $\phi(E)$ can be mapped to one part of the inset of E, and its size is bounded by K.

3 The hourglass pattern

In this section, we describe the intuition of our core contribution. We consider a specific pattern of dependencies, called the *hourglass pattern*, that forces a convex K-bounded set to have a specific shape. We can exploit this property to significantly improve the derived I/O complexity lower bound of programs that exhibit such a pattern.

In the whole section, we use the Modified Gram-Schmidt algorithm as an illustrative example, whose right-looking variant is provided in Figure 1. Using the automatic tool IOLB [17] to apply the K-partitioning method (described in Section 2) to the MGS computation results in a lower bound in $\Omega\left(\frac{MN^2}{\sqrt{S}}\right)$. By using the hourglass pattern, we obtain a more precise lower bound:

$$\frac{M^2N(N-1)}{8(S+M)} \leq Q(MGS)$$

```
1  for (k = 0; k < N; k += 1) {
2    nrm = 0.0;
3    for (i = 0; i < M; i += 1)
4      nrm += A[i][k] * A[i][k];
5    R[k][k] = sqrt(nrm);
6
7    for (i = 0; i < M; i += 1)
8      Q[i][k] = A[i][k] / R[k][k];
9
10   for (j = k + 1; j < N; j += 1) {
11     R[k][j] = 0.0;
12     for (i = 0; i < M; i += 1)
13 SR:   R[k][j] += Q[i][k] * A[i][j];
14     for (i = 0; i < M; i += 1)
15 SU:   A[i][j] = A[i][j] - Q[i][k] * R[k][j];
16   }
17 }
```

Figure 1: Modified Gram-Schmidt - Right-Looking (from Polybench [18]). The input matrix A is of size $M \times N$, and the output of the algorithm are matrices Q (the orthonormalized column vector basis) and R such that $A = QR$. The usual right-looking Gram-Schmidt reuses the matrix A, instead of defining a new matrix Q. SR and SU are labels of two statements, updating R and A.

3.1 Intuition of the hourglass pattern

Intuition Figure 2 presents the main idea of the hourglass pattern. It is a repeating succession of reduction and broadcast statements, such that the number of elements reduced/broadcasted is parametric, thus greater than the cache size S. There are 3 categories of dimensions in this pattern: (a) the dimensions over which the reduction and the broadcast are performed (horizontal axis of Figure 2), (b) the "temporal" dimensions over which the hourglass pattern is repeated (vertical axis of Figure 2), and (c) the neutral dimensions that do not interact with the hourglass pattern.

Running example The hourglass pattern appears on several linear algebra kernels, including the Modified Gram-Schmidt kernel (Figure 1). The pattern appears between the last two statements: statement SR which updates $R[k][j]$, and statement SU which updates $A[i][j]$. The statement SR is a reduction along the i dimension and uses, in particular, all the values of $A[\cdot][j]$ produced during the previous iteration of k. The statement SU broadcasts $R[k][j]$ across the i dimension to update all the $A[\cdot][j]$ of the current iteration of k. Therefore, dimension k is a temporal dimension, dimension i is the reduction/broadcast dimension and dimension j is a neutral dimension. There is exactly one dimension in each category in this example,

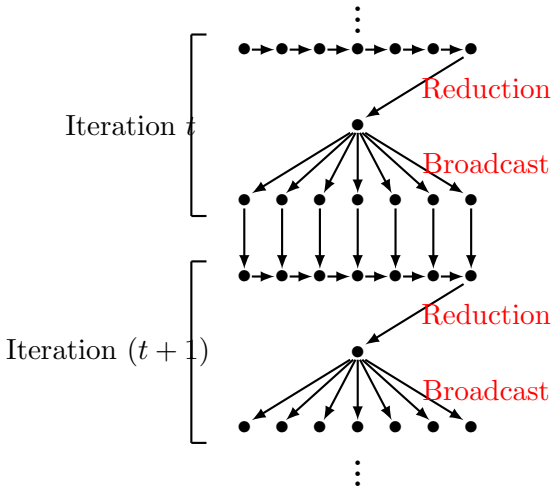

Figure 2: Shape of an hourglass pattern, inside the dependence graph. A node is an instance of a statement of the program, and an edge is a data dependency between two nodes. The t dimension is an external loop surrounding the hourglass.

but in general, there might be several.

Consequences of the hourglass pattern When considering a K-bounded set over this pattern, we notice that if it spans over several iterations of the temporal dimension t, then the set **must** include all the nodes of the broadcast/reduction in between, due to the convexity property of the set. Therefore, we have two situations:

- Either the K-bounded set spans over several iterations of t and includes all the nodes over the reduction/broadcast dimension. Notice that this is not always possible, depending on the size of the broadcast/reduction dimension and the value of K.
- Or, the K-bounded set is "flat" along the t dimension.

For both situations, we can deduce much stronger constraints on the sizes of the projection $|\phi(E)|$ used in the Brascamp-Lieb theorem (Theorem 2). This provides the derivation of an improved lower bound compared to the classical methodology.

3.2 Hourglass pattern - formal definition

In this section, we provide a formal definition of the hourglass pattern.

```
1  for(k = 0; k < N; k += 1){
2      norma2 = 0.0;
3      for(i = k + 1; i < M; i += 1){
4          norma2 += A[i][k] * A[i][k];
5      }
6      norma = sqrt(A[k][k] * A[k][k] + norma2);
7      A[k][k]=(A[k][k] > 0)?
8              (A[k][k]+norma):(A[k][k]-norma);
9
10     tau[k] = 2.0 /
11         (1.0 + norma2 / (A[k][k] * A[k][k]));
12
13     for(i = k + 1; i < M; i += 1){
14         A[i][k] /= A[k][k];
15     }
16     A[k][k] = (A[k][k] > 0)?(-norma):(norma);
17
18     for(j = k + 1; j < N; j += 1){
19         tau[j] = A[k][j];
20         for(i = k + 1; i < M; i += 1){
21 SR:           tau[j] += A[i][k] * A[i][j];
22         }
23         tau[j] = tau[k] * tau[j];
24         A[k][j] = A[k][j] - tau[j];
25         for(i = k + 1; i < M; i += 1){
26 SU:           A[i][j] = A[i][j] - A[i][k] * tau[j];
27         }
28     }
29 }
```

Figure 3: QR Householder computation - Part A2V (LAPACK routine GEQR2).

Preliminary notations Given an instance for a statement SX, we call an *iteration vector* the tuple of the (integral) values of its surrounding loop indices. The *iteration domain $\mathcal{D}_{SX}$ of the statement SX* is the set of iteration vectors that respect the conditions on the indices of the surrounding loops.

As mentioned above, in general, there might be several reduction or temporal dimensions. Thus, we will consider sets of dimensions, and represent their iteration as a vector $\vec{i} = (i_1, i_2, \dots)$. Given an iteration $\vec{k}$, we write $\overrightarrow{k+1}$ to represent the next valid lexicographic value of $\vec{k}$. We extend this notation to $\overrightarrow{k+n}$ where n is an integer.

The hourglass pattern Considering a statement S of the CDAG of a program, the *hourglass pattern* is a pattern of dependencies with the following properties:

- *Partitioning of the dimensions.* The dimensions of the statement SX can be partitioned into 3 groups: (i) the temporal dimensions $\vec{k}$, (ii) the reduction/broadcast dimensions $\vec{i}$, and (iii) the neutral dimensions $\vec{j}$. For simplicity of the presentation, we assume that $\vec{k}$ are the first/outer dimensions and that $\vec{i}$ are the last/inner dimensions.

- *Path in the dependence graph.* For any valid value of $\vec{i}$ and $\vec{i'}$, there is a dependency chain between the instances $SX[\vec{k}, \vec{j}, \vec{i}]$ and $SX[\overrightarrow{k+1}, \vec{j}, \vec{i'}]$.

- *Large width of the hourglass.* Let us consider W, the number of statement instances on all the dependency chains between $SX[\vec{k}, \vec{j}, \vec{i}]$ and $SX[\overrightarrow{k+2}, \vec{j}, \vec{i}]$. This expression depends on the parameters of the program, and cannot be bounded by a constant value.

Notice that, to have such a chain of dependencies, it must include a reduction and a broadcast. The dimensions $\vec{i}$ are the dimensions which are reduced on and broadcasted over. The dimensions $\vec{k}$ are the dimensions that are incremented by a constant factor when looping along this loop. So, once a path is found, partitioning the dimensions should be unambiguous, if this condition is also satisfied.

The automatic detection of such an hourglass pattern has been implemented inside the IOLB tool, using a polyhedral library [23, 24].

Examples For the MGS computation (Figure 1), we consider the statement SU and the cycle of dependencies going through the SR. We confirm that there are $2M$ statement instances inside a dependency chain between two instances $SU[k, j, i]$ and $SU[k+2, j, i]$: $SR[k+1, j, \cdot]$ and $SU[k+1, j, \cdot]$. The same reasoning would hold if we considered the statement SR instead of SU.

For the A2V QR Householder computation (Figure 3), we consider the statement SU, and the cycle of dependencies going through the SR statement. There are $(M-k)$ statement instances $SR[k, j, \cdot]$, inside a dependency chain between two instances $SU[k, j, i]$ and $SU[k+1, j, i]$.

4 Lower bound proof using the hourglass pattern

In this section, we show how to exploit the hourglass pattern of a program to derive a tighter lower bound on the data movement. We will use the Modified Gram-Schmidt program (Figure 1) as a running example to illustrate our proof.

Once again, this proof has been integrated inside the automatic data movement lower bound derivation tool, IOLB [17], and is applied when an hourglass pattern is detected.

Preliminary notations Let us denote with E a set of integral iteration vectors. The cardinality of such a set $|E|$ is the number of integer points inside this set.

Given a dimension k, $\phi_k : (i, j, k) \mapsto (k)$ is the projection to the dimension k. This notation can be extended to several dimensions. For example, $\phi_{j,k} : (i, j, k) \mapsto (j, k)$.

Given a set E, a dimension k and a value k_0, $E_{k=k_0}$ is a slice of E, i.e., the set of points of E whose value along the dimension k is k_0: $E_{k=k_0} = \{(i, j, k) \in E \mid k = k_0\}$. We extend this notation to several dimensions, as in $E_{k=k_0, j=j_0}$. In the rest of the paper, we use a compact notation for slices, by omitting the dimension when it is not ambiguous, like E_{k_0,j_0}.

Starting point and intuition The goal of the proof is to find an upper bound on the size of a K-bounded set, to transform it into a lower bound on the data volume required by a portion of the program. So, we start the proof with an arbitrary convex K-bounded set E containing instances of the broadcast statement $SX[\vec{k}, \vec{j}, \vec{i}]$, which we assume is part of an hourglass pattern. Just like in the classical proof, an analysis of the dependencies of the program yields a set Φ of projections $\phi_{\vec{x}}$ for some dimensions $\vec{x}$, which project on the inset of E.

The classical proof involves applying the Brascamp-Lieb theorem on the K-bounded set E, by bounding the size $|\phi_{\vec{x}}(E)|$ of each of these projections by K.

In our proof, we split E into two parts (Section 4.1) and we adapt the set of projections when we apply the Brascamp-Lieb theorem on each of these parts (Sections 4.2 and 4.3) to obtain more precise bounds.

Running example In the case of MGS, by following the chain of dependencies associated with the accesses $A[i][j]$, $Q[i][k]$ and $R[k][j]$ of statement SU, we infer that the projections are $\phi_{i,j}$, $\phi_{i,k}$ and $\phi_{k,j}$. When following the classical proof, the application of the Brascamp-Lieb theorem results in the following inequality:

$$|E| \leq |\phi_{i,j}(E)|^{\frac{1}{2}}.|\phi_{i,k}(E)|^{\frac{1}{2}}.|\phi_{k,j}(E)|^{\frac{1}{2}} \leq K^{3/2}$$

We prove in Section 4.4 a tighter upper bound: $|E| \leq \frac{K^2}{M} + 2K$.

4.1 Part 1 - Decomposition of the K-bounded set

In the first part of the proof, we decompose E into the union of two fragments: (i) I' which has volume along the $\vec{k}$ dimensions, and (ii) F which is flat along the $\vec{k}$ dimensions. Both parts have their own upper bound, obtained with different reasoning, that is described in Section 4.2 and Section 4.3.

Decomposition of E We consider the number of different values of $\vec{k}$ for a given value of $\vec{j}$:

$$Tick_{\vec{j}} = \{\vec{k} \mid \exists \vec{i},\ SX[\vec{k},\vec{j},\vec{i}] \in E_{\vec{j}}\}.$$

We split E into two sets:

- $E' = \cup_{\vec{j} \in J_{3+}} E_j$ where $J_{3+} = \{\vec{j} \mid 3 \leq Tick_{\vec{j}}\}$, and
- $E'' = \cup_{\vec{j} \in J_{12}} E_j$ where $J_{12} = \{\vec{j} \mid 1 \leq Tick_{\vec{j}} \leq 2\}$.

The intuition between this separation is that (i) E' contains the connected components which need to include an entire line of statement instances along the $\vec{i}$ dimensions, and (ii) E'' contains the connected components which are "flat" along the $\vec{k}$ dimensions.

We start by focusing on E', to show that these components need to include a parametric number of iterations along the $\vec{i}$ dimension, using the hourglass pattern.

Lemma 3 (Structure of E')**.** *Given some $\vec{j} \in \phi_{\vec{j}}(E')$, let us consider the slice $E'_{\vec{j}}$ along the dimensions $\vec{j}$. Let us consider:*

- $\overrightarrow{k_{\min}}(\vec{j}) = min_{\vec{k}}\{\vec{k} \mid SX[\vec{k},\vec{j},\vec{i}] \in E'_{\vec{j}}\}$,
- $\overrightarrow{k_{\max}}(\vec{j}) = max_{\vec{k}}\{\vec{k} \mid SX[\vec{k},\vec{j},\vec{i}] \in E'_{\vec{j}}\}$,
- *$\vec{a}$, any subset of the indices of $\vec{i}$,*

- $|\phi_{\vec{a}}(\mathcal{D}_S)| \geq W_{\vec{a}}$, *a lower bound on the size of the projection of* $\mathcal{D}_S$, *the iteration domain of the statement* SX.

Then:

1. $E'_{\vec{j}}$ *is a connected component.*
2. *For all* $\overrightarrow{k_{\min}}(\vec{j}) <_{lex} \vec{k} <_{lex} \overrightarrow{k_{\max}}(\vec{j})$, $|\phi_{\vec{a}}(E'_{\vec{j},\vec{k}})| \geq W_{\vec{a}}$.

Proof. (1) Because SX satisfies the hourglass pattern properties, for any two instances $SX[\vec{k},\vec{j},\vec{i}]$ and $SX[\vec{k'},\vec{j},\vec{i'}]$ in $E'_{\vec{j}}$ where $\vec{k} <_{lex} \vec{k'}$, we can prove that there is a chain of dependencies from one statement instance to another. Therefore, using the convexity property of E, we conclude that E'_j is a connected component.

(2) By considering one instance of index $\vec{k} = \overrightarrow{k_{\min}}$ and another of index $\vec{k} = \overrightarrow{k_{\max}}$, we can show that all the $SX[\vec{i},\vec{j},\vec{k}]$ are in the middle of a dependency chain between the two of them. Therefore, by projecting the $\vec{i}$ on the subset of dimension $\vec{a}$, we conclude:

$$\forall \overrightarrow{k_{\min}}(\vec{j}) <_{lex} \vec{k} <_{lex} \overrightarrow{k_{\max}}(\vec{j}),\ |\phi_{\vec{a}}(E'_{\vec{j},\vec{k}})| \geq |\phi_{\vec{a}}(\mathcal{D}_S)| \geq W_{\vec{a}}.$$

□

We consider $E' = I' \uplus B'$ defined by:

- $I' = \cup_{\vec{j} \in J_{3+}} (E_{\vec{j}} - E_{\vec{j},\overrightarrow{k_{\min}}(\vec{j})} - E_{\vec{j},\overrightarrow{k_{\max}}(\vec{j})})$: the "inside" of the connected components of E', according to dimensions $\vec{k}$.
- $B' = \cup_{\vec{j} \in J_{3+}} (E_{\vec{j},\overrightarrow{k_{\min}}(\vec{j})} \cup E_{\vec{j},\overrightarrow{k_{\max}}(\vec{j})})$: the "boundaries" of the connected components of E', according to dimensions $\vec{k}$.

We define $F = B' \uplus E''$ the flat parts of E, and we adapt our previous decomposition of E to obtain the desired decomposition:

$$E = E' \uplus E'' = I' \uplus F.$$

4.2 Part 2 - Bound on the size of I'

Let us focus on the upper bound of the size of I', using Lemma 3. The goal is to have tighter bounds on the projections involving some of the reduce/broadcast dimensions $\vec{i}$, to use them with the Brascamp-Lieb theorem, instead of the classical "$\leq K$" bound.

Lemma 4 (Bounds on the size of some of the projections of I')**.** *Let us consider a projection* $\phi_{\vec{x},\vec{a}} \in \Phi$, *where* $\vec{a}$ *is a subset of* $\vec{i}$, *and* $\vec{x}$ *is a subset*

of $(\vec{j}, \vec{k})$. *Assume that we have a lower bound* $|\phi_{\vec{a}}(\mathcal{D}_S)| \geq W_{\vec{a}}$ *on the size of the projection of the iteration domain* $\mathcal{D}_S$ *of statement* S. *Then:*

$$|\phi_{\vec{x}}(I')| \leq \frac{K}{W_{\vec{a}}}.$$

Proof. Because I' is a subset of a K-partition, then we have $|\phi_{\vec{a},\vec{x}}(I')| \leq K$.

By slicing I' along the dimensions of $\vec{x}$, we also have:

$$|\phi_{\vec{a},\vec{x}}(I')| = |\phi_{\vec{a},\vec{x}}(\cup_{\overrightarrow{x''}} I'_{\overrightarrow{x''}})| = \sum_{\overrightarrow{x''}} |\phi_{\vec{a},\vec{x}}(I'_{\overrightarrow{x''}})|.$$

Furthermore:

$$\begin{aligned}
|\phi_{\vec{a},\vec{x}}(I'_{\overrightarrow{x''}})| &\geq |\phi_{\vec{a}} I'_{\overrightarrow{x''}})| && \text{(a slice along } \vec{x} \text{ only has a single point to be projected along } \vec{x}) \\
&\geq |\phi_{\vec{a}}(I'_{\overrightarrow{j''},\overrightarrow{k''}})| && (I'_{\overrightarrow{j''},\overrightarrow{k''}} \subset I'_{\overrightarrow{x''}}, \text{ for some } (\overrightarrow{j''}, \overrightarrow{k''}) \text{ matching } \overrightarrow{x''} \text{ on its dimensions}) \\
&\geq W_{\vec{a}}. && \text{(by Lemma 3)}
\end{aligned}$$

Therefore: $|\phi_{\vec{a},\vec{x}}(I')| \geq W_{\vec{a}} \times |\phi_{\vec{x}}(I')|$. In other words:

$$|\phi_{\vec{x}}(I')| \leq \frac{|\phi_{\vec{a},\vec{x}}(I')|}{W_{\vec{a}}} \leq \frac{K}{W_{\vec{a}}}.$$

□

To obtain a bound on $|I'|$, we use the set of projections Φ, modified in the following way:

- We add a projection on $\vec{i}$ whose bound is: $|\phi_{\vec{i}}(I')| \leq W$, where W is the width of the hourglass.
- When a projection on $(\vec{x}, \vec{a})$ shares some of its dimensions $\vec{a}$ with $\vec{i}$, we use the projection on $\vec{x}$ instead, and the bound given by Lemma 4.
- The rest of the projections, which do not involve dimensions of $\vec{i}$, are unchanged and associated with a classical upper bound $|\phi_{\vec{x}}(I')| \leq K$.

Then, we apply the Brascamp-Lieb theorem, while optimizing the values of the power s of the size of the projections $|\phi(E)|$. Notice that some projections can be filtered out through this optimization process ($s = 0$), to obtain a tighter bound on I'.

Running example We notice that Φ contains two projections $\phi_{i,j}$ and $\phi_{i,k}$, both of them using the dimension i and another dimension. Therefore, from Lemma 4, we have the following bounds:

$$|\phi_j(I')| \leq \frac{K}{M} \quad \text{and} \quad |\phi_k(I')| \leq \frac{K}{M}.$$

We apply the Brascamp-Lieb theorem on I', using the projections on i, on j instead of a projection on (i,j), and on k instead of a projection on (i,k), each with coefficient 1:

$$|I'| \leq |\phi_i(I')| \times |\phi_j(I')| \times |\phi_k(I')|.$$

By using our specialized bounds, we have:

$$|I'| \leq M \times \frac{K}{M} \times \frac{K}{M} = \frac{K^2}{M}.$$

4.3 Part 3 - Bound on the size of F

We now focus on the bound of the remaining part of E, i.e., $F = (B' \uplus E'')$. Our goal is to exploit the fact that this set is "flat" along the $\vec{k}$ dimensions, to improve the upper bounds on the projections used in the Brascamp-Lieb application to F. In particular, our proof will consider each $F_{\vec{j}}$ separately, the slice of F for a given value $\vec{j}$. Then, by picking a list of well-chosen projections for the Brascamp-Lieb theorem, we obtain interesting bounds on the size of $F_{\vec{j}}$, that are finally summed to obtain our bound on F.

We recall that E' and E'' contains the $\vec{j}$-slices of E for the values of $\vec{j}$ such that $Tick_{\vec{j}}$ are respectively at least 3 and at most 2. Since $B' \subset E'$, B' and E'' do not share the same set of $\vec{j}$. So we get a "flatness bound": $\forall \vec{j} \in \phi_{\vec{j}}(F),\ |\phi_{\vec{k}}(F_{\vec{j}})| \leq 2$.

In addition, we notice that because $F_{\vec{j}} \subset E$, for any projection $\phi \in \Phi$, $|\phi(F_{\vec{j}})| \leq |\phi(E)| \leq K$.

In this part of the proof, instead of focusing on F, we apply the Brascamp-Lieb theorem to the slice $F_{\vec{j}}$.

As in Section 4.2, we start with the same list of projections Φ to the in-set of E, obtained by inspecting the chain of dependencies of the considered statement. Then, we customize this list of projections to exploit the properties of $F_{\vec{j}}$:

- We add a projection on $\vec{k}$ and the flatness bound $|\phi_{\vec{k}}(F_{\vec{j}})| \leq 2$.
- We identify a projection that involves a non-empty subset of the dimensions of $\vec{j}$: after applying the Brascamp-Lieb theorem, we will not try to immediately use an upper bound for this projection. Let us call $\phi_{\vec{w}}$ this projection.

- The remaining projections are left alone, and associated with the classical upper bound $|\phi_{\vec{x}}(F_{\vec{j}})| \leq |\phi_{\vec{x}}(F)| \leq K$.

At that point, we have obtained an upper bound on $|F_{\vec{j}}|$ of the following form:

$$|F_{\vec{j}}| \leq e \times |\phi_{\vec{w}}(F_{\vec{j}})|.$$

where e is a parametric expression using the parameter K and independent of the value of $\vec{j}$. Notice that the Brascamp-Lieb power above $|\phi_{\vec{w}}|$ must be equal to 1, due to the fact that this is the only projection involving the $\vec{j}$ dimensions.

Let us consider the collection of projected sets $\phi_{\vec{w}}(F_{\vec{j}})$. Two of these projected sets are either (i) identical, along the dimensions of $\vec{j}$ which are not present in $\vec{w}$, or (ii) disjoint. Because the distinct values of these projections are always a subset of the inset of E, of size K, the sum of the size of the disjoint union of these distinct values is also bounded by K. Let us call R the number of values that can be taken by the dimensions of $\vec{j}$ which are not in $\vec{w}$. This is also the maximum number of times a $\phi_{\vec{w}}(E)$ projects on the same value. Notice that $R = 1$ when $\vec{w}$ covers all the dimensions of $\vec{j}$, and that R can be a parametric expression in general. So, we have:

$$\sum_{\vec{j} \in \phi_{\vec{j}}(F)} |\phi_{\vec{w}}(F_{\vec{j}})| \leq R \times K.$$

Combining all these observations, we obtain the following upper bound on the size of F:

$$\begin{aligned} |F| = \sum_{\vec{j} \in \phi_{\vec{j}}(F)} |F_{\vec{j}}| &\leq \sum_{\vec{j} \in \phi_{\vec{j}}(F)} e \times |\phi_{\vec{w}}(F_{\vec{j}})| \\ &\leq e \times \sum_{\vec{j} \in \phi_{\vec{j}}(F)} |\phi_{\vec{w}}(F_{\vec{j}})| \leq e \times R \times K. \end{aligned}$$

Running example In the MGS case, the flatness bound yields: $\forall j, |\phi_k(F_j)| \leq 2$. We can apply the Brascamp-Lieb theorem on F_j, with the projection $\phi_{i,j}$, together with the projection ϕ_k:

$$|F_j| \leq |\phi_k(F_j)| \times |\phi_{i,j}(F_j)| \leq 2 \times |\phi_{i,j}(F_j)|.$$

And since dimension j is included in the dimensions (i, j) of the projection, we have $R = 1$ and therefore:

$$|F| = \sum_{j \in \phi_j(F)} |F_j| \leq 2 \times \sum_{j \in \phi_j(F)} |\phi_{i,j}(F_j)| \leq 2K.$$

4.4 Part 4 - Wrapping things up

We have found in Section 4.2 and Section 4.3 an upper bound of the size of the two parts of E. We simply sum them together to obtain an upper bound of the size of E, then apply Theorem 1 to deduce a lower bound on the data movement.

Running example Thanks to the above results, we can obtain lower bounds on the data movement of MGS.

Theorem 5 (Lower bounds for MGS)**.** *The communication volume Q for the MGS algorithm on a $M \times N$ matrix can be bounded as follows:*

$$\frac{M^2N(N-1)}{8(S+M)} \leq Q$$

Furthermore, if $S \leq M$, we also have:

$$\frac{(M-S)N(N-1)}{4} \leq Q$$

Proof. From the previous results, we have:

$$|E| = |I'| + |F| \leq \frac{K^2}{M} + 2K.$$

Then, by using Theorem 1 with $K = 2S$:

$$(K-S) \times \frac{MN(N-1)}{2 \cdot \left(\frac{K^2}{M} + 2K\right)} = \frac{M^2N(N-1)}{8(S+M)} \leq Q$$

Due to Lemma 3 and because we have at least an input dependency involving the dimension i, $|InSet(E')| > M$. So, if we have $S \leq M$, then E' must be empty. Therefore, $E = F$, and we can use only the second part of the bound: $|E| \leq 2K$.

Using Theorem 1 again, but this time with $K = M$, we obtain:

$$(K-S) \cdot \frac{MN(N-1)}{2 \times 2K} = (M-S) \cdot \frac{N(N-1)}{4} \leq Q$$

□

5 Experimental results - New lower bounds

In this section, we report the data movement lower bounds generated by IOLB for four kernels exhibiting an hourglass pattern. We compare the results using our technique (new bound) with those obtained without it (old bound). These kernels are:

Kernel	Old bound [17]	New bound (hourglass)
MGS	$\Omega\left(\frac{MN^2}{\sqrt{S}}\right)$	$\Omega\left(\frac{M^2N(N-1)}{S+M}\right)$
QR HH A2V	$\Omega\left(\frac{MN^2}{\sqrt{S}}\right)$	$\Omega\left(\frac{MN^2(N-M)}{N-M-S}\right)$
QR HH V2Q	$\Omega\left(\frac{MN^2}{\sqrt{S}}\right)$	$\Omega\left(\frac{MN^2(N-M)}{N-M-S}\right)$
GEBD2	$\Omega\left(\frac{MN^2}{\sqrt{S}}\right)$	$\Omega\left(\frac{MN^2(M-N+1)}{8(S+M-N+1)}\right)$
GEHD2	$\Omega\left(\frac{N^3}{\sqrt{S}}\right)$	$\Omega\left(\frac{N^4}{N+2S}\right)$

Figure 4: Summary of the new asymptotic data movement lower bounds.

- Modified Gram-Schmidt (Figure 1), already used as a running example in Section 4.
- QR Householder algorithm: both its A2V (Figure 3) and V2Q parts (left-looking variants of respectively the GEQR2 and ORG2R subroutines in LAPACK [15]).
- Reduction to a bidiagonal matrix (GEBD2 subroutine)
- Reduction to a Hessenberg matrix (GEHD2 subroutine)

Figure 5 summarizes all the newly found full lower bounds, and Figure 4 focuses on their leading term, to emphasize their improvements. More precisely:

- Section 5.1 contains an asymptotic analysis of the MGS lower bound.
- Section 5.2 presents the lower bound for both QR Householder parts, and the GEBD2 computation.
- Section 5.3 presents the lower bounds for the GEHD2 computation.

Annex A contains a description of a tiled algorithm for MGS and for HH A2V and the computation of their amount of data movement. This provides an upper bound to the minimal amount of data movement required by these algorithms, which matches asymptotically the provided lower bound.

5.1 MGS - Asymptotic analysis

In this section, we analyze the bound obtained in Theorem 5 for MGS, by specializing it for different ordering of M and S:

- If $S \leq M/2$, we have $M/2 \leq M - S$, so that the second bound yields:

$$\frac{MN^2}{8} = \Omega(MN^2) \leq Q$$

Kernel	Old bound [17]
MGS	$\frac{2M+3MN+MN^2}{\sqrt{S}} + 5M - MN + \frac{7N-N^2}{2} - S - 6$
QR HH A2V	$\frac{3MN^2+6M+7N-N^3-9MN-6}{3\sqrt{S}} + 5M - MN + 5N - S - 13$
QR HH V2Q	$\frac{3MN^2-N^3+6M+7N-9MN-6}{3\sqrt{S}} + 2M + 2N + \frac{N-N^2}{2} - S - 4$
GEBD2	$\frac{3MN^2-N^3-9MN+6M+7N-6}{3\sqrt{S}} + 5N + 5M - MN - S - 13$
GEHD2	$\frac{5N^3-30N^2+55N-30}{3\sqrt{S}} + \frac{69N-9N^2}{2} - 3 * S - 56$
Kernel	**New bound (hourglass)**
MGS	$\frac{N^2M^2+2M^2-3NM^2}{8(M+S)} + 5M - MN + \frac{7N-N^2}{2} - S - 6$
QR HH A2V	$\frac{3MN^2-9MN+7N+6M-6-N^3}{24(1-\frac{S}{N-M})} + 5M - MN + 5N - S - 13$
QR HH V2Q	$\frac{3MN^2-N^3+6M+7N-9MN-6}{24(1+\frac{S}{M-N})} + 2M + 2N + \frac{N-N^2}{2} - S - 4$
GEBD2	$\frac{3MN^2-N^3+3N^2-15MN+4N+18M-12}{24(1+\frac{S}{1+M-N})} + 5N + 7M - MN - S - 18$
GEHD2	$\frac{N^3-6N^2+11N-6}{12(1+\frac{S}{N-M-1})} - N^2 + 12N - S - 19$

Figure 5: Data movement lower-bounds (with constants) automatically derived by IOLB [17] without/with hourglass detection. In GEHD2's new bound, a new parameter M is introduced, corresponding to the place where we split the outer loop. Depending on S and N, it can be instantiated with a different parametric expression (cf Section 5.3).

- If $M/2 \leq S$, we have $S + M \leq 3S$, so that the first bound becomes:

$$\frac{M^2N^2}{24S} = \Omega\left(\frac{M^2N^2}{S}\right) \leq Q$$

The first result matches the amount of data movements obtained by the classical ordering of the MGS algorithm, as presented at the end of Section 3.1. Both the algorithm and the bound are thus asymptotically optimal when S is small.

Demmel et al. [7] propose a tiled ordering for the case $2M \leq S$ (in Section F.2 of their paper), that achieves an amount of data movement $O(M^2N^2/S)$, thus matching asymptotically the second upper bound. Both of these bounds are thus optimal up to a constant factor. For reference, the tiled ordering is detailed in Appendix A.1, together with the proof on its amount of data movement.

We can compare these bounds with the lower bound returned by the classical hourglass-less proof, whose asymptotic bound is $\Omega(\frac{MN^2}{\sqrt{S}})$. Our first bound for small values of S is stronger by a factor of $Theta(\sqrt{S})$. By writing our second bound as $\Omega(\frac{M}{\sqrt{S}} \cdot \frac{MN^2}{\sqrt{S}})$, we see that our bound is stronger by a factor of $\Theta(\frac{M}{\sqrt{S}})$. Since the input matrix has size $M \times N$, we can assume that $S < MN$, otherwise, the whole matrix fits in the cache and there is no need

```
1  for (k = N − 1; k > −1; k − = 1) {
2      for (j = k + 1; j < N; j += 1){
3          tau[j] = 0.0;
4          for(i = k + 1; i < M; i += 1){
5  SR:         tau[j] += A[i][k] * A[i][j];
6          }
7      }
8      for(j = k + 1; j < N; j += 1){
9  ST:     tau[j] *= tau[k];
10     }
11     A[k][k] = 1.0 − tau[k];
12     for(j = k + 1; j < N; j += 1){
13         A[k][j] = −tau[j];
14     }
15     for(j = k + 1; j < N; j += 1){
16         for(i = k + 1; i < M; i += 1){
17 SU:       A[i][j] −= A[i][k] * tau[j];
18         }
19     }
20     for(i = k + 1; i < M; i += 1){
21         A[i][k] = −A[i][k] * tau[k];
22     }
23 }
```

Figure 6: QR Householder computation - Part V2Q (LAPACK subroutine ORG2R). We assume that $M \geq N$.

to minimize data transfers. Besides, we know that $N \leq M$, which leads to $S < M^2$. We can conclude that $\frac{M}{\sqrt{S}} > 1$: our bound is asymptotically at least as strong as the previous bound.

The results of Theorem 5 can also be presented differently, to improve the constant in front of the dominant term. Indeed:

If $S \ll M$, the first bound yields $\dfrac{MN^2}{4} \leq Q$,

Similarly, if $M \ll S$, the second result becomes $\dfrac{M^2N^2}{8S} \leq Q$.

5.2 Householder QR factorization and GEBD2

In this section, we present the bounds to both parts of the Householder QR factorization (LAPACK routines GEQR2 et ORG2R). The computation of the first part (A2V) is given in Figure 3 and the computation of the second part (V2Q) in Figure 6.

By using the hourglass reasoning, we obtain the following new lower bounds for both kernels.

Theorem 6 (Lower bounds for HH - part A2V)**.** *The communication volume Q for the A2V part of the HH algorithm on a $M \times N$ matrix, with $M > N$,*

can be bounded as follows:

$$\frac{(3M-N)N^2(M-N)^2}{24(MS+(M-N)^2)} \leq Q$$

If $M \gg N$, then the bound becomes:

$$\frac{M^2N(N-1)}{8(S+M)} \leq Q$$

Theorem 7 (Lower bound for HH - Part V2Q)**.** *The communication volume Q for the HH V2Q algorithm on a $M \times N$ algorithm, with $M > N$, can be bounded as follows:*

$$\frac{N(N-1)(3M-N-1)(M-N)^2}{24((M-N)^2+SM)} \leq Q$$

When $M \gg N$, this bound becomes:

$$\frac{N(N-1)M^2}{8(S+M)} \leq Q$$

One interesting detail of the proof concerns the detection of the hourglass, and the criteria on its size (third criterion introduced in Section 3.2). For the MGS computation, the size of its hourglass was constant and equal to M. In the case of both Householder computations, the size of their hourglass is parametrized by the outer loop iteration value k, and is equal to $(M-1-k)$. Its smallest value happens for $k = N-1$, and is $(M-N)$. By using this value in Lemma 3, we can derive the announced lower bound.

The lower bound proof of the GEBD2 subroutine is similar to both Householder proofs.

Theorem 8 (Lower bounds for GEBD2)**.** *The communication volume Q for the GEBD2 subroutine on a $M \times N$ matrix, with $M \geq N$, can be bounded as follows:*

$$\frac{MN^2(M-N+1)}{8(S+M-N+1)} \leq Q$$

If $M \gg N$, then the bound becomes:

$$\frac{M^2N^2}{8(S+M)} \leq Q$$

```
1  for (j = 0; j < n − 2; j+= 1) {
2    norma2 = 0.0;
3    for (i = j + 2; i < n; i+= 1) {
4      norma2 += A[i][j] * A[i][j];
5    }
6    norma = sqrt( A[j + 1][j] * A[j + 1][j] + norma2 );
7    A[j + 1][j] = (A[j + 1][j] > 0)?
8            (A[j + 1][j]+norma ):( A[j + 1][j]−norma );
9    tau = 2.0/(1.0 + norma2/(A[j + 1][j] * A[j + 1][j] ));
10   for (i = j + 2; i < n; i+= 1 ) {
11     A[i][j] /= A[j + 1][j];
12   }
13   A[j + 1][j] = (A[j + 1][j] > 0)?(−norma ):( norma );
14   for (i = j + 1; i < n; i+= 1) {
15     tmp[i] = A[j + 1][i];
16     for (k = j + 2; k < n; k+= 1) {
17       tmp[i] += A[k][j] * A[k][i];
18     }
19   }
20   for (i = j + 1; i < n; i+= 1) {
21     tmp[i] *= tau;
22   }
23   for (i = j + 1 ; i < n ; i+= 1) {
24     A[j + 1][i] −= tmp[i];
25   }
26   for (i = j + 2; i < n; i+= 1) {
27     for (k = j + 1; k < n; k+= 1) {
28       A[i][k] −= A[i][j] * tmp[k];
29     }
30   }
31   for (i = 0; i < n; i+= 1) {
32     tmp[i] = A[i][j + 1];
33     for (k = j + 2; k < n; k+= 1) {
34       tmp[i] += A[i][k] * A[k][j];
35     }
36   }
37   for (i = 0; i < n; i+= 1) {
38     tmp[i] *= tau;
39   }
40   for (i = 0; i < n; i+= 1) {
41     A[i][j + 1] −= tmp[i];
42   }
43   for (i = 0; i < n; i+= 1) {
44     for (k = j + 2; k < n; k+= 1) {
45       A[i][k] −= tmp[i] * A[k][j];
46     }
47   }
48 }
```

Figure 7: Hessenberg matrix factorization of a $N \times N$ matrix (LAPACK subroutine GEHD2)

5.3 Hessenberg matrix reduction

In this section, we present the bound for the Hessenberg matrix factorization kernel (GEHD2 kernel in LAPACK), whose code is in Figure 7. By using the hourglass reasoning, we obtain the following new lower bound for the GEHD2 kernel.

Theorem 9 (Lower bound for GEHD2)**.** *The communication volume Q for the Hessenberg matrix factorization algorithm on a $N \times N$ algorithm can be bounded as follows:*

$$\frac{1}{12}.\frac{N^4}{N+2S} \leq Q$$

When $N \gg S$,

$$\frac{N^3}{24} \leq Q$$

As in Section 5.2, the size of the hourglass depends on the value of iteration j of the outermost loop (temporal dimension), and is equal to $(N-2-j)$, where $0 \leq j < N-2$. This means that its minimal value is 1, which causes an issue with our proof.

A way to solve this problem is to introduce a new parameter $M < N-2$ and to consider a loop splitting of the outermost temporal dimension j at iteration M. First, notice that a loop splitting does not change the dependencies of a program, thus the lower bound of the split version will apply to the original one.

The first half of the split program satisfies the hourglass pattern and has a minimum size of its hourglass of $(N-M-1)$. Thus, as long as this quantity is not bounded by a constant, the hourglass reasoning should apply. The second half does not satisfy the last condition of the hourglass pattern, so the classical derivation is used there. Notice that its bound is asymptotically worse than the bound on the first part, so the bound of the first part will dominate. We consider $M = \frac{N}{2} - 1$ to obtain the first bound, and $M = N - S - 2$ to obtain the second bound when $N \gg S$.

6 Related work

I/O complexity and automation of the proof The seminal work of Hong and Kung [11] introduced the red-blue pebble game and used it as a formalism to manually prove the I/O lower bound of programs. Following this contribution, many papers [1, 2, 3, 13, 14, 8, 22, 19, 20, 21, 5] focused on the manual proof of lower bounds for various (classes of) programs. A large portion of these papers consider a formalism that forbid recomputation. This assumption is necessary to decompose complex CDAGs into simpler subregions, and to be able to recombine the bounds of each region.

Irony et al. [13] used the Loomis-Whitney bound, a specialization of the Brascamp-Lieb theorem with canonical projections, on the `gemm` algorithm. This was later extended by the work of Christ et al. [6] for the class of affine programs, by using the Brascamp-Lieb theorem. Then, Elango et al. [9] added the idea of considering paths of dependencies, completing the K-partitioning method as introduced in Section 2 of our paper.

This leads to the work of Olivry et al. [17] that introduced the first automatic lower bound derivation tool, based on combining bounds obtained from the K-partitioning and the wavefront methods of proof. This tool includes several refinements to the bound derivation, by taking advantage of some specific properties. For example, if the projections use disjoint parts of the inset, the derived bound could be improved by a constant factor. A later extension [16] exploits the fact that some dimensions have very small sizes and applies this reasoning to convolutions.

7 Conclusion

In this paper, we introduced a new proof reasoning to derive data movement lower bounds, based on a pattern of dependencies, called the hourglass pattern. This pattern appears on several linear algebra kernels, such as the Modified Gram-Schmidt, the QR Householder decomposition (A2V and V2Q parts), the bidiagonal matrix reduction, and the Hessenberg matrix reduction. We managed to derive asymptotically tighter lower bounds for these kernels, compared to the previous classical methods. We also provided tiled algorithms for MGS and QR A2V, whose amount of data movement matches asymptotically the new lower bounds, up to a constant factor.

We integrated the hourglass detection and its associated proof in the automatic data movement lower bound derivation tool of Olivry et al., IOLB [17], whose implementation is freely available.

Individual contributions Fabrice identified the original problem on MGS and the intuition on how to attack this problem. The implementation in IOLB and the proof were written by Guillaume. The proof was refined by Lionel, who also provided a detailed study of the bound, and an experimental performance analysis. Julien identified the interesting kernels exhibiting an hourglass pattern and provided valuable insights on the linear algebra side. Lionel and Julien are also at the source of the upper bound proof in the appendix.

Acknowledgments Julien was partially supported for this work by NSF award #2004850.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM*, 59(6), Jan. 2013.

[4] O. Beaumont, L. Eyraud-Dubois, J. Langou, and M. Vérité. I/o-optimal algorithms for symmetric linear algebra kernels. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '22, page 423–433, New York, NY, USA, 2022. Association for Computing Machinery.

[5] G. Bilardi, M. Scquizzato, and F. Silvestri. A lower bound technique for communication in bsp. *ACM Transactions on Parallel Computing*, 4(3), feb 2018.

[6] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays – part 1, 2013.

[7] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2008.

[8] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[9] V. Elango, F. Rastello, L. Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 567–580, New York, NY, USA, 2015. Association for Computing Machinery.

[10] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data movement complexity of computational dags for parallel execution. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14,

page 296–306, New York, NY, USA, 2014. Association for Computing Machinery.

[11] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)*, pages 326–333. ACM, 1981.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 319–329, New York, NY, USA, 1988. Association for Computing Machinery.

[13] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[14] G. Kwasniewski, M. Kabic, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[15] R. LAPACK. `https://github.com/Reference-LAPACK/lapack`, 1992.

[16] A. Olivry, G. Iooss, N. Tollenaere, A. Rountev, P. Sadayappan, and F. Rastello. IOOpt: Automatic derivation of I/O complexity bounds for affine programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1187–1202, New York, NY, USA, 2021. Association for Computing Machinery.

[17] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, and F. Rastello. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 808—-822. ACM, 2020.

[18] L.-N. Pouchet and T. Yuki. PolyBench/C: The polyhedral benchmark suite, version 4.2, 2016. `http://polybench.sf.net`.

[19] D. Ranjan, J. E. Savage, and M. Zubair. Strong I/O lower bounds for binomial and FFT computation graphs. In B. Fu and D. Du, editors, *Computing and Combinatorics - 17th Annual International Conference, COCOON 2011, Dallas, TX, USA, August 14-16, 2011. Proceedings*, volume 6842 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2011.

[20] D. Ranjan, J. E. Savage, and M. Zubair. Upper and lower I/O bounds for pebbling r-pyramids. *J. Discrete Algorithms*, 14:2–12, 2012.

[21] J. Scott, O. Holtz, and O. Schwartz. Matrix multiplication i/o-complexity by path routing. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, page 35–45, New York, NY, USA, 2015. Association for Computing Machinery.

[22] T. M. Smith, B. Lowery, J. Langou, and R. A. van de Geijn. A tight i/o lower bound for matrix multiplication, 2019.

[23] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. 2010.

[24] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

A Data movement upper bound - tiled algorithms

In this Appendix, we present the tiled algorithm for the MGS and Householder A2V kernels. We compute the data movement of these algorithms to obtain an upper bound on the minimal amount of data movement needed, and we show how it relates to their corresponding lower bound.

A.1 Tiled algorithm for MGS

As shown in Section 4, we have an improved asymptotic lower bounds for MGS when $M \ll S$. For reference and completeness, we show in Figure 8 a tiled left-looking ordering, based on the ideas from [7, §9.1.4]. Then, we provide the proof that this ordering asymptotically matches this bound.

Let us show that, if we choose the block size B such that $(M+1)B < S$, then this algorithm satisfies the following properties:

1. it does not spill out of memory
2. the amount of "read" operations is $\frac{1}{2}\frac{M^2N^2}{S}$ (leading term).
3. the amount of "write" is $MN + \frac{1}{2}N^2$ (leading terms).
4. the total amount of I/O (read+write) is $\frac{1}{2}\frac{M^2N^2}{S}$ (leading term).

The input of this algorithm is an M-by-N matrix A. The outputs are the M-by-N matrix Q (stored in the array A) and the upper triangular matrix R. The first for-loop (line 1) moves along the columns of A by block. At step $j0$, we work with the block $A(0{:}M, j0{:}j0+B)$. We consider the left-looking variant of the algorithm. So, at the start of the $j0$ iteration, the block $A(0{:}M, j0{:}j0+B)$ contains the initial data of the matrix A, and at the end of the $j0$ iteration, the same block contains the final data of the matrix Q. The block $A(0{:}M, j0{:}j0+B)$ stays in memory during the $j0$ iteration, so we read it at the start of the $j0$ iteration and write it at the end of this iteration.

For each vector column on the left of this block of columns (the $A(0{:}M, i)$ where $i < j0$), we first need to project our block of A, $A(0{:}M, j0{:}j0+B)$, onto the orthogonal complement of this column. This is done by the first loop nest (line 3), which performs this projection one column at a time. This requires the program to load the column $A(0{:}M, i)$. Since we are using the left-looking version of the algorithm, we only need to "read" $A(0{:}M, i)$ and we do not need to "write" $A(0{:}M, i)$ back in memory.

Now, let us count the number of I/O accesses of this program. To fit the $(B+1)$ columns of A in cache (block $A(0{:}M, j0{:}j0+B)$ and column $A(0{:}M, i)$), we assume that B is chosen such that $MB + M < S$. Over the whole course of the algorithm, moving the blocks $A(1{:}M, j0{:}j0+B)$ into cache only happens once per block, so the I/O cost for these movements

```
for (j0 = 0; j0 < N; j0 += B) {
// read A(1:M,j0:j0+B)
   for (i = 0; i < j0; i += 1) {
//    read A(1:M,i)
      for (j = j0; ((j < j0 + B)&&(j < N)); j += 1) {
         R[i][j] = 0.0;
         for (k = 0; k < M; k += 1)
            R[i][j] += A[k][i] * A[k][j];
         for (k = 0; k < M; k += 1)
            A[k][j] -= A[k][i] * R[i][j];
      }
//    discard A(1:M,i)
   }
   for (j = j0; ((j < j0 + B)&&(j < N)); j += 1) {
      for (i = j0; i < j; i += 1) {
         R[i][j] = 0.0;
         for (k = 0; k < M; k += 1)
            R[i][j] += A[k][i] * A[k][j];
         for (k = 0; k < M; k += 1)
            A[k][j] -= A[k][i] * R[i][j];
      }
      R[j][j] = 0.0;
      for (k = 0; k < M; k += 1)
         R[j][j] += A[k][j] * A[k][j];
      R[j][j] = sqrt(R[j][j]);
      for (k = 0; k < M; k += 1)
         A[k][j] /= R[j][j];
   }
// write A(1:M,j0:j0+B)
}
```

Figure 8: Modified Gram-Schmidt - left-looking tiled code with $\mathcal{O}\left(\frac{M^2N^2}{S}\right)$ data movements, if the block size B satisfies $(M+1)B < S$.

is only MN read and MN write operations. The main I/O cost happens when reading the columns $A(0{:}M, i)$ one at a time, inside the very first two for-loops (lines 1 and 3, on iterations $j0$ and i).

Notice that the I/O benefit of the blocked algorithm occurs here: every time we move the column $A(1{:}M, i)$ in memory, we reuse it to project over B columns at once, instead of projecting over a single column in the unblocked version. This leads to a reduction of the I/O by a factor B. More precisely, the total amount of data movement involved in the reading of column $A(0{:}M, i)$ is:

$$\begin{aligned} & M \times |\{j0, i \mid 0 \leq j0 < N \text{ and } j0 \bmod B = 0 \text{ and } 0 \leq i < j0\}| \\ & \quad = M \times |\{j0, i \mid 0 \leq j0 < N/B \text{ and } 0 \leq i < j0.B\}| \\ & \quad = M \times \sum_{j0=0}^{N/B} j0 \times B \;\approx\; MB\left(\tfrac{N^2}{2B^2}\right) \;=\; \tfrac{1}{2}\tfrac{MN^2}{B}. \end{aligned}$$

The total data movement for the algorithm reading the columns $A(1:M, i)$, for $(i = 0; i < j0; i++)$, for $(j0 = 0; j0 < N; j0 += B)$ is:

$$\begin{aligned} & \textstyle\sum_{(j0=0;j0<N;j0+=B)} \sum_{(i=0;i<j0;i++)} M \\ & \quad = M \textstyle\sum_{(j0=0;j0<N;j0+=B)} j0 \\ & \quad \approx MB \textstyle\sum_{(j0=0;j0<\frac{N}{B};j0++)} j0 \\ & \quad \approx \tfrac{1}{2} MB \left(\tfrac{N}{B}\right)^2 = \tfrac{1}{2}\tfrac{MN^2}{B}. \end{aligned}$$

This is indeed a factor of B lower than the number of reads of the unblocked algorithm.

Because we have chosen B such that $M(B+1) < S$, we can take its maximal value to minimize the number of reads: $B = \lfloor \frac{S}{M} \rfloor - 1 \approx \frac{S}{M}$. Thus, the I/O cost of this tiled algorithm is, assuming $B \geq 1$:

$$I/O \approx \frac{1}{2}\frac{M^2N^2}{S}.$$

Additional remarks This algorithm works for any block size B, any M, any N ($N < M$), and any S. When we make the analysis of data movement, we further assume that $M(B+1) < S$. So, for the I/O analysis to be correct, we need at least $S > 2M$. In other words, we assume that at least two columns fit in cache.

Also, we do not consider R in the data movement or in the cache constraint. Once a value of R is computed, this value is used only once, and right after its computation. We can thus write these values as soon as we are done with them. Hence, (1) the total number of writes due to R is $N(N+1)/2 \approx N^2/2$. This $N^2/2$ "write" I/O is negligible for the algorithm (and cannot be optimized anyway), so we will neglect it; and (2) the requirement of cache for R is 1 for the whole computation, so we do not consider R at all for the cache usage.

Note that it is also possible to follow the same reasoning to write a tiled version of the right looking variant of the algorithm. We get a volume of I/O with a similar order of magnitude, but the constant is higher. The right-looking variant performs more I/O than the left-looking variant, but, in addition, writes are more expensive than reads, and the I/Os of the right-looking variant are dominated by writes, whereas, the I/Os of the left-looking variant are dominated by reads. So, in the sequential context, we prefer to use the left-looking variant. There are some advantages in using the right-looking variants in the sense that the updates can be done in parallel, so in a parallel multicore setting, for example, the right-looking variants would be interesting.

A.2 Tiled algorithm for Householder A2V

Figure 9 shows an implementation that matches asymptotically this bound. It is based on the same idea as the tiled algorithm for MGS in Section A.1, but adapted to the A2V algorithm.

In particular, we show that, if we chose B such that $(M+1)B < S$, then our algorithm satisfies the following properties:

1. it does not spill out of memory.
2. the amount of "read" is $\frac{1}{2}\frac{M^2N^2-MN^3/3}{S}$.
3. the amount of "write" is MN.
4. the total amount of I/O (read+write) is $\frac{1}{2}\frac{M^2N^2-MN^3/3}{S}$.

We have only show the leading terms for these previous I/O costs.

The input of our algorithm is the M-by-N matrix A. The outputs are the N-by-N upper triangular matrix R and the M-by-N unit lower triangular matrix V, that are computed in place in the M-by-N array A. The first for-loop (line 1), using iteration $k0$, moves along the column of A by block. At step $k0$, we work on the block $A(0{:}M, k0 : k0 + B)$. We consider the left-looking variant of the algorithm. So, at the start of the $k0$ iteration, the block $A(0{:}M, k0 : k0 + B)$ contains the initial data of the matrix A; and at the end of the $k0$ iteration, it contains the final data (V and R). This block remains in cache during the $k0$ iteration, so we read it at the start of the $k0$ iteration, and we write it at the end of this iteration.

For each vector column on the left of this block (the $A(0{:}M, j)$ where $j < k0$), we need to reflect our current block $A(j{:}M, k0{:}k0 + B)$ with the elementary Householder reflectors defined by $A(j + 1{:}M, j)$ and $tau[j]$, one column at a time. This requires the algorithm to load $A(j + 1{:}M, j)$. Since we are using the left-looking version of the algorithm, we do not need to "write" it back in memory.

```
1  for (k0 = 0; k0 < N; k0 += B) {
2  // read A(1:M,k0:k0+B)
3    for(j = 0; j < k0; j += 1){
4  //     read A(j:M,j)
5      for (k = k0; (k < k0 + B)&&(k < N); k += 1) {
6        tmp = A[j][k];
7        for(i = j + 1; i < M; i += 1)
8          tmp += A[i][j] * A[i][k];
9        tmp = tau[j] * tmp;
10       A[j][k] = A[j][k] - tmp;
11       for(i = j + 1; i < M; i += 1)
12         A[i][k] = A[i][k] - A[i][j] * tmp;
13     }
14 //     discard A(j:M,j)
15   }
16   for (k = k0; ((k < k0 + B)&&(k < N)); k += 1) {
17     for(j = k0; j < k; j += 1) {
18       tmp = A[j][k];
19       for(i = j + 1; i < M; i += 1)
20         tmp += A[i][j] * A[i][k];
21       tmp = tau[j] * tmp;
22       A[j][k] = A[j][k] - tmp;
23       for(i = j + 1; i < M; i += 1)
24         A[i][k] = A[i][k] - A[i][j] * tmp;
25     }
26     norma2 = 0.0;
27     for(i = k + 1; i < M; i += 1)
28       norma2 += A[i][k] * A[i][k];
29     norma = sqrt( A[k][k] * A[k][k] + norma2 );
30     A[k][k] = (A[k][k] > 0.0)?(A[k][k]+norma)
31                 :(A[k][k]-norma);
32     tau[k] = 2.0/
33         (1.0 + norma2/(A[k][k]*A[k][k]));
34     for(i = k + 1; i < M; i += 1)
35       A[i][k] /= A[k][k];
36     A[k][k] = (A[k][k] > 0.0)?(-norma):(norma);
37   }
38 // write A(1:M,k0:k0+B)
39 }
```

Figure 9: QR Householder A2V part - Tiled implementation, with $\mathcal{O}\left(\frac{M^2N^2}{S}\right)$ data movements, assuming $2M < S$.

Now, let us count the number of I/O accesses of this program. To fit the $(B+1)$ columns of A in cache, we assume that B is chosen such that $M(B+1) < S$. Over the whole course of the algorithm, moving the blocks $A(1{:}M, k0{:}k0+B)$ in cache happens only once per block, so the I/O for this is $2MN$ and is negligible. The main I/O cost happens when reading the columns $A(j+1{:}M, j)$, one at a time, inside the two very first for-loops (using iteration $k0$ and j).

Notice that the I/O gain of this blocked algorithm against the unblocked version occurs there: every time we load a column $A(j+1{:}M, j)$, we reuse it to reflect over B columns at once, instead of a single column for the unblocked version. This leads to a reduction of the I/O cost by a factor of B. More precisely, the total reads of columns $A(j+1{:}M, j)$ in the algorithm is:

$$\begin{aligned}
\textstyle\sum_{k0=0,\ k0+=B}^{N-1} \sum_{j=0}^{k0-1}(M-j) \ &\approx\ \textstyle\sum_{k0=0}^{N/B}\sum_{j=0}^{B.k0}(M-j) \\
&\approx\ \textstyle\sum_{k0=0}^{N/B}\left(B.k0.M - \frac{(B.k0)^2}{2}\right) \\
&=\ B.M.\left(\textstyle\sum_{k0=0}^{N/B} k0\right) - \frac{B^2}{2}.\left(\textstyle\sum_{k0=0}^{N/B} k0^2\right) \\
&\approx\ B.M.\frac{N^2}{2.B^2} - \frac{B^2}{2}.\frac{N^3}{3B^3} \ =\ \frac{MN^2}{2B} - \frac{N^3}{6B} \\
&\approx\ \frac{1}{2}\frac{MN^2 - N^3/3}{B}.
\end{aligned}$$

This is indeed a factor of B less than the unblocked algorithm.

Because we require B to satisfy $M(B+1) < S$, we choose $B = \lfloor \frac{S}{M} \rfloor - 1 \approx \frac{S}{M}$. So the I/O cost of our algorithm is, assuming $B \geq 1$,

$$I/O \approx \frac{1}{2}\frac{M^2N^2 - N^3/3}{S}.$$

Additional remark Just like for the MGS algorithm, the I/O analysis requires $S > 2M$, which means that we assume that at least two columns fit in cache.