# On Optimal Partitioning For Sparse Matrices In Variable Block Row Format

Peter Ahrens
pahrens@mit.edu
Massachusetts Institute of Technology
and Sandia National Laboratories
Cambridge, Massachusetts

Erik G. Boman
egboman@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico

## ABSTRACT

The Variable Block Row (VBR) format is an influential blocked sparse matrix format designed to represent shared sparsity structure between adjacent rows and columns. VBR consists of groups of adjacent rows and columns, storing the resulting blocks that contain nonzeros in a dense format. This reduces the memory footprint and enables optimizations such as register blocking and instruction-level parallelism. Existing approaches use heuristics to determine which rows and columns should be grouped together. We adapt and optimize a dynamic programming algorithm for sequential hypergraph partitioning to produce a linear time algorithm which can determine the optimal partition of rows under an expressive cost model, assuming the column partition remains fixed. Furthermore, we show that the problem of determining an optimal partition for the rows and columns simultaneously is NP-Hard under a simple linear cost model.

To evaluate our algorithm empirically against existing heuristics, we introduce the 1D-VBR format, a specialization of VBR format where columns are left ungrouped. We evaluate our algorithms on all 1626 real-valued matrices in the SuiteSparse Matrix Collection. When asked to minimize an empirically derived cost model for a sparse matrix-vector multiplication kernel, our algorithm produced partitions whose 1D-VBR realizations achieve a speedup of at least 1.18 over an unblocked kernel on 25% of the matrices, and a speedup of at least 1.59 on 12.5% of the matrices. The 1D-VBR representation produced by our algorithm had faster SpMVs than the 1D-VBR representations produced by any existing heuristics on 87.8% of the test matrices.

## CCS CONCEPTS

• **Theory of computation → Discrete optimization**; *Data structures design and analysis*; *Problems, reductions and completeness*; *Design and analysis of algorithms*; Dynamic programming; Vector / streaming algorithms; • **Mathematics of computing → Mathematical software performance**; Solvers; Hypergraphs; Graph algorithms; Computations on matrices.

## KEYWORDS

Sparse Matrices, Blocks, Block Sparsity, Partitioning, Variable Block Row, VBR, 1D-VBR

## 1 INTRODUCTION

Matrices that occur in practice are often **sparse**, meaning that most of their entries are zero. Several sparse matrix formats have been proposed that process and store only the nonzero entries in an effort to improve performance. For example, the Compressed Sparse Row (**CSR**) format stores a list of nonzero locations in each row. While most sparse formats perform better than dense formats, a considerable portion of the storage is used to record the individual locations of nonzeros. This contributes to the runtime of memory-bound kernels such as Sparse Matrix-Vector Multiply (**SpMV**) [46]. The complexity of these compressed formats makes it difficult to apply optimization techniques that are typical to dense settings, such as register blocking and instruction-level parallelism.

Because sparse matrices come from physical applications, many matrices have the property that nonzeros occur close together. **Blocked** sparse matrix formats have enjoyed a long history of study because they provide a way to use this observation to reduce the complexity of storing individual nonzero locations, opting to store rectangular regions of nonzeros in dense blocks instead. This allows implementations to store only the size and location of each block, avoid storage of each nonzero location within the block. The block can be processed as a small dense matrix, enabling dense performance engineering techniques.

Blocked formats have a long history of use, but one of the first formats to receive considerable study was the Variable Block Row (**VBR**) format [23, 24, 32–34, 41, 43], where similar adjacent rows and columns are grouped together. Unlike many formats which require blocks to be of fixed size, the number of rows or columns that may be grouped together varies across each dimension, producing variably sized blocks. Since blocks are produced by merging entire rows or columns, the blocks are aligned, which allows implementations to reuse elements of other kernel arguments along the direction of alignment. In general, producing bigger blocks means that less location information is needed, but as blocks get bigger, they may begin to cover more zeros that need to be stored explicitly in the dense storage. While the VBR format was motivated by scientific applications that produce matrices with clear block structure, we investigate the application of these techniques to arbitrary sparse matrices.

In Appendix A, we prove that the problem of determining optimal groupings of adjacent rows and columns for VBR format is NP-Hard under a simple linear cost model by reduction from the Maximum Cut problem [25, 29]. The remainder of the work focuses on the more constrained problem of determining an optimal grouping of the rows given a fixed grouping of columns.

At of the time of writing, only heuristic algorithms have been given to determine which rows or columns should be grouped together in either the full problem or our restricted case. In Section 6, we propose a novel modification to the implementation of the commonly used “overlap similarity” heuristic to ensure that it reads each element of the matrix only once.

Optimal algorithms have been proposed for the related, but not directly applicable, problems of sequential hypergraph and graph partitioning [5, 15, 26]. In Section 7, we adapt the algorithm due to Grandjean et. al. to produce a linear time, single pass, algorithm which can determine the optimal contiguous row groups under a fixed column grouping and an expressive cost model which applies directly to VBR and 1D-VBR. For example, we can use a cost model that directly minimizes the memory footprint of the resulting VBR format, or one of the widely used affine row-wise cost models for blocked sparse matrix-vector multiplication (SpMV) [9, 24]. Our algorithm runs in time $O(N + m \cdot w_{\max} + n)$, where $N$ is the number of nonzeros, $m$ is the number of rows, $w_{\max}$ is the maximum block dimension (treated as a small constant), and $n$ is the number of columns. Our algorithm operates directly on the sparse matrix in CSR format, taking only one pass over the matrix. In practice, benefits are not typically observed for increasing $w_{\max}$ beyond approximately 12 [40].

To test our partitioner empirically, we invent a specialization of the VBR sparse matrix format for the case where the columns are ungrouped. We refer to this new, restricted, format as 1D-VBR. Because SpMV has a long history of use as the target kernel for blocked formats, we use SpMV as our target kernel. SpMV is often used as a subroutine in iterative solvers, where a dense vector multiplies the same sparse matrix hundreds of times before a solution is found. Thus, a practical use case for our optimal 1D-VBR partitioner would be to run the partitioner and convert the matrix to 1D-VBR once, then offset the conversion cost with the savings obtained after multiplying the matrix many times in an iterative solver.

We test our optimal algorithm against existing heuristics using a Julia implementation of the 1D-VBR format [1] on the entire set of 1626 real-valued matrices in the SuiteSparse collection [13]. On 12.5% of the matrices we tested, our optimally-partitioned 1D-VBR format under an empirically-determined affine cost model exhibited an SpMV speedup of at least 1.59 over the CSR implementation. This partitioner was able to justify its cost within 74.8 multiplications on 25% of the matrices we tested. Using our algorithm to partition under one of our three studied cost models produced the fastest multiply on 43.7% of the tested matrices. Not only does this prove the effectiveness of our partitioner, it describes the extent of availability of block structure in practical sparse matrices.

## 2 PARTITIONS

Let $A$ be an $m \times n$ **matrix** with $N$ nonzeros, where $A_{i,j}$ corresponds to the value in the $i^{\text{th}}$ row and $j^{\text{th}}$ column. For convenience, we will use the interval notation $i : i'$ to represent the integer sequence $i, i+1, ..., i'$. We can then describe submatrices of $A$ as $A_{i:i',j:j'}$. We will use : to represent the full interval of valid indices. Thus, $A_{:,4}$ should be interpreted as the fourth column of $A$, and $A_{1:3,:}$ should be interpreted as the first three rows of $A$.

In practice, adjacent rows (and columns) often contain similar patterns of nonzero locations. To capitalize on this observation, we will group similar rows together, forming a **row part**. A $K$-**partition** $P$ of rows of $A$ assigns each row $i$ to one of $K$ **parts** $1 \leq P_i \leq K$. In this work, we will insist that our partitions stay **contiguous**, meaning that $P_{i-1} \leq P_i \leq P_{i+1}$ for all $1 \leq i \leq m$. Because our partitions are contiguous, we can represent a $K$-partition $P$ using vector $\Pi(P)$ of the $K + 1$ split positions. Thus, the rows in part $p$ lie in the range $\Pi_p : (\Pi_{p+1} - 1)$. Note that when the argument to a function is clear from context, we will omit the argument for brevity. We will use : to represent the **trivial partition**, which assigns each row to a distinct part, and $\vec{1}$ to represent the **unit partition**, which assigns each row to the same part.

Although our primary focus will be partitions of rows, the Variable Block Row (VBR) format uses a partition of rows and a partition of columns simultaneously. Thus, we also introduce notation for an $L$-partition $Q$ of columns, with the corresponding split vector $\Phi(Q)$. The partitions $P$ and $Q$ tile our matrix with $K \times L$ contiguous, non-overlapping, rectangular **blocks**. The block $(p, q)$ is of size $w_p \times u_q = (\Pi_{p+1} - \Pi_p) \times (\Phi_{q+1} - \Phi_q)$. Blocked formats store only **nonzero blocks**, or blocks that contain at least one nonzero of $A$. If we pick our partitions wisely, many blocks will be zero, and will not need to be stored.

We use $v_i(A)$ to refer to the set of nonzero locations in the $i^{th}$ row of $A$, and $e_j(A)$ to refer to the set of nonzero locations in the $j^{th}$ column of $A$.

$$v_i = \{j | A[i, j] \neq 0\}, e_j = \{i | A[i, j] \neq 0\}.$$

Similarly, let $\gamma_i(A, Q)$ be the set of column parts which contain nonzeros in the $i^{th}$ row, and let $\lambda_j(A, P)$ be the set of row parts which contain nonzeros in the $j^{th}$ column. Tersely,

$$\gamma_i = \{Q_j | A[i, j] \neq 0\}, \lambda_j = \{P_i | A[i, j] \neq 0\}.$$

Note that if $Q$ and $P$ are each the **trivial partition**, then $v_i(A) = \gamma_i(A, :)$ and $e_j(A) = \lambda_j(A, :)$.

Block partitioning algorithms sometimes focus on reordering the rows to group similar rows together [30, 35, 38]. If we allow our blocks to become large, the problem begins to resemble graph and/or hypergraph partitioning [10, 18]. While noncontiguous partitions allow for more expressive blocks, permuting a matrix or vector may be an expensive memory-intensive procedure. In some situations, the matrix may have already been reordered for numerical reasons, and the user might want to operate on the matrix without changing the column ordering. Furthermore, there are several matrices which do not need to be reordered to take advantage of similarities among adjacent rows. Partitioning problems often focus on reordering the rows to group the similar ones together. As a parting thought on the topic, research into the less complex contiguous partitioning problem may help to provide insight into the more general case.

## 3 SPARSE FORMATS

Sparse matrices are commonly stored in Compressed Sparse Row (CSR) format [36], which consists of three vectors *pos*, *idx*, and *val*. The length $m + 1$ vector *pos* stores the regions of *idx* and *val* corresponding to each row. The vectors *idx* and *val* (each of length $N$) store the sorted nonzero column locations and corresponding values of nonzeros, respectively. Assume we were to store $A$ in CSR format and wanted to determine the value $A_{i,j}$. We start by finding the unique index $l$ such that $A.pos[i] \leq l < A.pos[i + 1]$ and $A.idx[l] = j$. If no such $l$ exists, $A_{i,j} = 0$ and is not stored explicitly in CSR format. Otherwise, $A_{i,j} = A.val[l]$. Note that $A.pos[1]$ will

[1] Code is available at https://github.com/peterahrens/SparseMatrix1DVBCs.jl

always be 1 and $A.pos[m + 1]$ will always be $N + 1$. Storing $A$ in CSR format uses

$$s_{\text{CSR}}(A) = (m + 1)s_{\text{index}} + N \cdot s_{\text{index}} + N \cdot s_{\text{value}} \quad (1)$$

bits, where $s_{\text{index}}$ and $s_{\text{value}}$ are the sizes of the index and value types, in bits.

When adjacent rows or columns are used to represent different aspects of the same component of a system, the nonzeros in these rows or columns often appear in the same locations. For example, in many sparse matrices, adjacent nonzeros may represent partial derivatives of the same variable with respect to some set of important dependent variables. If different aspects are recorded for each component, the number of similar rows or columns may vary from component to component. Similarly, in matrices where adjacent rows or columns represent similar datapoints, the nonzeros tend to be clustered. These observations motivated the development of the Variable Block Row (VBR) format, which groups similar rows and columns together [32]. While VBR was motivated by applications like multiphysics simulations with several variables per mesh point, we focus on the more general setting of arbitrary sparse matrices, without assuming underlying meshes, PDEs, etc. VBR saves memory by storing only the locations of the resulting blocks instead of the locations of the individual nonzeros. Because the blocks produced by VBR are aligned along rows and columns, SpMV implementations can register block the corresponding sections of the input and output vectors, and use dense linear algebra to process each block. VBR is described in the SPARSKIT library [33, 34], the SparseBLAS specification [32], and the OSKI Sparse Kernel Interface [41, 42], and is used internally by the MKL Paradiso solver [2].

The VBR format imposes both a row $K$-partition $P$ and a column $L$-partition $Q$. However, it is more efficient to store the length $K + 1$ split vectors $\Pi$ and $\Phi$ than it is to store the length $m$ and $n$ partition vectors. Instead of storing individual nonzero locations, the VBR format uses the *idx* vector to store block indices (the indices record the parts corresponding to each block). Because the blocks are variably sized, the block locations in *val* are not aligned with the positions of the block indices in the *idx* array. Therefore, we use a vector *ptr* of block locations to record the starting index of each block in *val*. Assume we were to store $A$ in VBR format and wanted to determine the value of the entry $A_{i,j}$. Let $p$ and $q$ be the block indices of the index $(i, j)$ (the integers such that $\Pi[p] \leq i < \Pi[p+1]$ and $\Phi[q] \leq j < \Phi[q + 1]$). If we cannot find $l$ such that $A.pos[p] \leq l < A.pos[p + 1]$ and $A.idx[l] = q$, then $A_{i,j} = 0$ because the block $(p, q)$ is entirely zero and is not stored in explicitly VBR format. Otherwise, our block contains at least one nonzero and starts at position $A.ptr[l]$ in the *val* array. Our block has dimensions $w_p \times u_q = (\Pi[p+1]-\Pi[p])\times(\Phi[q+1]-\Phi[q])$. Because VBR stores nonzero blocks in a dense, row-major format, $A_{i,j} = A.val[A.ptr[l] + (i - A.\Pi[p]) \cdot u_q + (j - A.\Phi[q])]$.

Let $N_{\text{index}}(A, P, Q)$ be the number of nonzero blocks induced by $P$ and $Q$, so that

$$N_{\text{index}}(A, P, Q) = |\{(P_i, Q_j)|A_{i,j} \neq 0\}|. \quad (2)$$

We will use $N_{\text{index},p}(A, P, Q)$ to represent the number of nonzero blocks in the $p^{th}$ row part. Let $N_{\text{value}}(A, P, Q)$ be the number of entries contained in all nonzero blocks induced by $P$ and $Q$, such that

$$N_{\text{value}}(A, P, Q) = \sum_{(p,q)\in\{(P_i,Q_j)|A_{i,j}\neq 0\}} w_p \cdot u_q \quad (3)$$

We will use $N_{\text{value},p}(A, P, Q)$ to represent the number of elements in nonzero blocks in the $p^{th}$ row part. The VBR format uses six arrays, $\Pi$, $\Phi$, *pos*, *idx*, *ptr*, and *val*. Storing $A$ in VBR format uses

$$s_{\text{VBR}}(A, P, Q) = (2(K+1)+(L+1)+2N_{\text{index}})s_{\text{index}}+N_{\text{value}}s_{\text{value}} \quad (4)$$

bits.

In this work, we introduce a special case of the VBR format where the column partition is trivial, meaning that the columns are not grouped together, and we concern ourselves only with row partitioning. We call this special format 1D-VBR. Because $\Phi$ is trivial, we do not need to store it. Additionally, because blocks have only one column, block sizes are constant within each row part and the stride between blocks is constant within each row part. Thus, instead of storing the location each block in the val array, we need only store the location of the first block in each row part. Thus, we replace the *ptr* array with a new array of row offsets, *ofs*. Assume we were to store $A$ in 1D-VBR format and wanted to determine the value of the entry $A_{i,j}$. Let $p$ be the block index of the index $i$ (the integer such that $\Pi[p] \leq i < \Pi[p + 1]$). Notice that the block index of $j$ is $j$ itself, since we have not partitioned the columns. If we cannot find $l$ such that $A.pos[p] \leq l < A.pos[p + 1]$ and $A.idx[l] = j$, then $A[i, j] = 0$ because the block $(p, j)$ is entirely zero and is not stored explicitly in 1D-VBR format. Otherwise, our block contains at least one nonzero and because the blocks in row part $p$ are all of size $w_p = \Pi[p + 1] - \Pi[p]$, our block starts at position $A.ofs[p] + (l - A.pos[p]) \cdot w_p$ in the *val* array. Thus, $A_{i,j} = A.val[A.ofs[p] + (l - A.pos[p]) \cdot w_p + (i - A.\Pi[p])]$. The 1D-VBR format uses five arrays, $\Pi$, *pos*, *idx*, *ofs*, and *val*. Storing $A$ in 1D-VBR format uses

$$s_{\text{1D-VBR}}(A, P) = (3(K + 1) + N_{\text{index}})s_{\text{index}} + N_{\text{value}}s_{\text{value}} \quad (5)$$

bits.

## 3.1 Related Sparse Formats

Blocked sparse formats have enjoyed a long history of study. In lieu of providing an exhaustive overview of existing formats, we refer the reader to works such as [23, 31, 42] which provide summaries of several sparse blocking techniques. We focus only on the most relevant formats here.

The BCSR format tiles the matrix with fixed-size dense format blocks, storing nonzero block locations in CSR format [12, 14, 20–22]. BCSR is referred to as BSR in the Intel®Math Kernel Library [2]. Cost models developed for BCSR depend on the number of nonzero blocks, leading to the development of row-wise sampling algorithms to estimate the number of nonzero blocks [9, 20, 20, 22, 24, 27, 41, 42]. A nonzero-wise sampling algorithm was proposed in [3, 4, 44].

Generalizing to less constrained block decompositions, unaligned block formats continue to use fixed-size blocks, but relax alignment requirements. The SPARSKIT implementation of BCSR relaxes the column alignment of blocks, allowing blocks to shift along the block rows[34]. One could imagine a format which groups adjacent blocks in 1D-VBR block rows to achieve a similar format. The UBCSR format uses a number of fixed block sizes that can start at any entry in the matrix [43]. An intriguing approximation algorithm has been described for producing good fixed-sized, unaligned sparse matrix block decompositions (the UBCSR format) [39]. The CSR-SIMD format produces dense blocks inside the rows, putting successive groups of nonzeros into SIMD-register sized blocks for instruction level parallelism [11]. Note that SpMVs on CSR-SIMD formatted matrices cannot reuse loads from the input vector, whereas 1D-VBR uses only one load from the input for each block, no matter how large the block is. The Variable Block Length (VBL) format, originally proposed in [30] and referred to as VBL in [23], relaxes the constraint that the blocks inside rows must be of fixed length. Both VBL and CSR-SIMD can reduce the size of the matrix when nonzeros occur next to each other in the same row. The Variable Blocked-$\sigma$-SIMD Format (VBSF) is similar to CSR-SIMD, but allows the blocks to be merged across multiple rows, so the blocks have a fixed width but variable height. The DynB format relaxes all alignment and size constraints, allowing variably sized blocks to start at any entry of the matrix [31]. Algorithms for producing CSR-SIMD, VBSR, and DynB formats create their blocks with greedy algorithms that add adjacent elements into the block up to a density-related threshold. Because these formats make decisions on a block-by-block basis, it makes sense to convert the matrix to blocked format at the same time as the block decomposition is determined [11, 31].

## 4 BLOCKED SPMV

Algorithm 1 shows an example SpMV for a matrix in CSR format. Notice that processing each element of the matrix requires a load from $A.val$, $A.idx$, and $x$. The accesses into $y$, $A.idx$, and $A.val$ are sequential, but the accesses to $x$ are random.

ALGORITHM 1. *Given $m \times n$ matrix $A$ in CSR format and a length $n$ vector $x$, add $A \cdot x$ to the length $m$ vector $y$, in-place.*

```
1:  function SpMV-CSR(y, A, x)
2:      for $i \leftarrow 1$ to $m$ do
3:          $yy \leftarrow y[i]$
4:          for $l \leftarrow A.pos[i]$ to $A.pos[i+1]$ do
5:              $j \leftarrow A.idx[l]$
6:              $yy \leftarrow yy + A.val[l] \cdot x[j]$
7:          end for
8:          $y[i] \leftarrow y$
9:      end for
10: end function
```

Algorithm 2 shows an example SpMV kernel for a matrix stored in 1D-VBR format. Processing each stored element of $A$ requires a load from $A.val$, but we only need to load from $A.idx$ and $x$ once for each block (this data reuse is a benefit of producing aligned blocks, and a key property enjoyed by 1D-VBR but not by CSR-SIMD). Of course, the computations and the sequential loads are now processed with vector instructions. If our vector size does not divide our block size, we can simply pad our vectors as they are loaded from memory, without needing to pad the stored blocks. For example, if our blocks are of size 3, we can process them using vectors of size 4, with the fourth value of our $y$ vector register remaining undefined.

ALGORITHM 2. *Given $m \times n$ matrix $A$ in 1D-VBR format and a length $n$ vector $x$, add $A \cdot x$ to the length $m$ vector $y$, in-place. We intend for the subvectors produced by ranged index expressions to be processed by vectorized instructions, so that the statement*

$$yy \leftarrow yy + A.val[A.\Pi[K] : A.\Pi[K+1]] \cdot x[j]$$

*loads a whole block from $A.val$ into a vector register, multiplies it pointwise by the scalar in $x$, and adds the resulting vector pointwise to the vector register $yy$.*

```
1:  function SpMV-1D-VBR(y, A, x)
2:      for $p \leftarrow 1$ to $K$ do
3:          $yy \leftarrow y[A.\Pi[p] : A.\Pi[p+1]]$
4:          for $l \leftarrow A.pos[p]$ to $A.pos[p+1]$ do
5:              $j \leftarrow A.idx[l]$
6:              $yy \leftarrow yy + A.val[A.\Pi[p] : A.\Pi[p+1]] \cdot x[j]$
7:          end for
8:          $y[A.\Pi[i] : A.\Pi[i+1]] \leftarrow yy$
9:      end for
10: end function
```

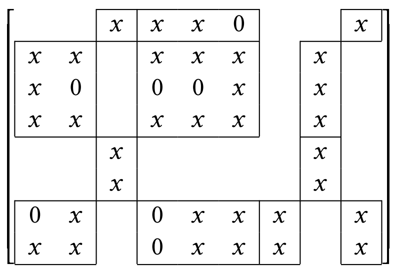
## 5 1D AND 2D PARTITIONING PROBLEMS

Because the blocks in a VBR format are stored in a dense format, we must trade off between a partition that uses larger blocks (and stores more explicit zeros) and a partition that uses smaller blocks (and stores more block locations). Practitioners often use cost models to measure the effect of performance parameters like block sizes. Several diverse cost models have been proposed for blocked sparse matrix formats [22, 28, 42]. While several of these models apply to VBR SpMV [24], we are not aware of any work which takes the next step to use the cost model to optimize a VBR partition. To simplify the presentation of our algorithms, we will keep our cost models simple. Our first cost model is simply the memory used by the representation, $s_{VBR}$, as described in (4). This model assumes that runtime will be directly proportional to the memory footprint of the induced VBR format. Because SpMV is a memory-bound kernel [46] and many sparse matrices do not fit in fast memory, we expect this model to work well for large matrices.

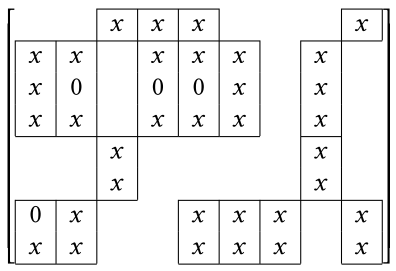
The form of our second cost model is taken from [24, (2)] and [9, (3)], which both model the time taken to compute a row part $p$ of weight $w_p$ as an affine function in the number of elements in the part. Cost models with similar forms have been proposed for similar blocked formats [20–22, 42]. The constant term $\alpha$ represents costs that occur once per row part, such as loading elements from $y$ and $A.\Pi$, etc. The coefficient on the linear term $\beta$ represents the cost of processing elements in the blocks. We use separate values of $\alpha_w$ and $\beta_w$ for each weight $w$ because the relationship between the number of rows in the row part and the constant and linear terms is not easily characterized. Thus,

$$t_{VBR}(A, P, Q) = \sum_{p=1}^{K} \alpha_{w_p} + \beta_{w_p} N_{\text{value},p}, \tag{6}$$

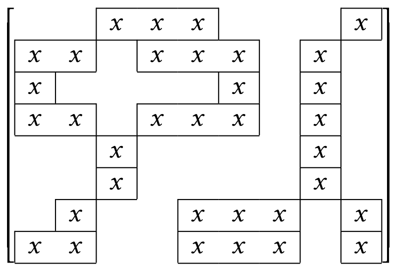
where $N_{\text{value},p}$ is the number of stored nonzero elements in the part. This is further simplified for 1D-VBR format since $u_q = 1$, $N_{\text{value},p} = w_p N_{\text{index},p}$, and we can move a constant factor $w_p$ into



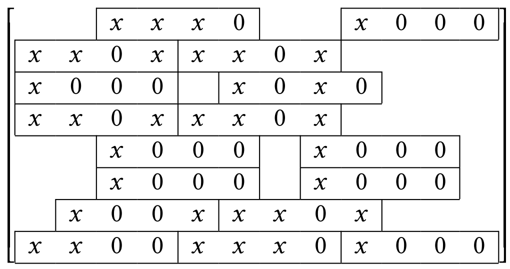
**(a) The stored entries of $A$ in VBR format. Here, $\Pi = [1, 2, 5, 7, 9]$ and $\Phi = [1, 3, 4, 7, 8, 9, 10]$.**



**(b) The stored entries of $A$ in 1D-VBR format. Here, $\Pi = [1, 2, 5, 7, 9]$.**



**(c) The stored entries of $A$ in 1D-VBL format.**



**(d) The stored entries of $A$ in CSR-SIMD format.**

**Figure 1: Various blocked sparse representations of a sample matrix $A$. Here, $x$ represents a nonzero value, 0 represents an explicitly stored 0 value, and each box represents a distinct stored block. Blank spaces correspond to implicit zeros of $A$. All of these formats store the nonzero blocks in a row-major order analogous to the way CSR stores nonzero entries.**

the linear term $\beta$, so that

$$t_{1D-VBR}(A, P) = \sum_{p=1}^{K} \left(\alpha_{w_p} + N_{\text{index},p}\beta_{w_p}\right). \tag{7}$$

For 1D-VBR, we produce values for $\alpha_{w_p}$ and $\beta_{w_p}$ by filling half of our L1 cache with a dense matrix stored in 1D-VBR format containing $n$ blocks of width $w_p$ in each row part. We then create a kernel which runs SpMV on this matrix 1000 times. Measuring the time for each $n \in 1 : 8$ allows us to fit $\alpha_{w_p}$ and $\beta_{w_p}$ with simple linear regression.

Our main problem can be stated as follows:

Problem 1 (Contiguous Row and Column Partitioning). *Given an $m \times n$ matrix $A$ and block size limit $w_{\max} \times u_{\max}$, find the $K$-partition $P$ and $L$-partition $Q$ minimizing the cost function $f(A, P, Q)$.*

We show in Appendix A that Problem 1 is NP-Hard for the very simple cost model

$$f(A, P, Q) = s \cdot N_{\text{index}} + N_{\text{value}} \tag{8}$$

where $s \geq 1$ is small constant and $w_{\max} \geq 2$. This cost model bears a striking similarity to our memory and runtime models. This cost model penalizes each nonzero block (index) with a cost of $s$, and each value in a nonzero block with a cost of 1, so this can represent the dominant terms in (4) and (6), our cost models for memory usage and runtime of the resulting VBR format. We further show that even minimizing the number of blocks used by VBR format is NP-Hard, which corresponds to the cost model

$$f(A, P, Q) = N_{\text{index}}. \tag{9}$$

The remainder of the paper will be focused on the much simpler Problem 2, for which we give an optimal linear-time algorithm for cost functions of the form

$$f(A, P, Q) = \sum_{p=1}^{K} g(w_p, N_{\text{index},p}, N_{\text{value},p}) \tag{10}$$

where $g$ is any positive-valued function. Because we can convert any row partitioning algorithm to a column partitioning algorithm by simply transposing our matrix first [17], without loss of generality we consider only the row partitioning case.

Problem 2 (Contiguous Row Partitioning). *Given an $m \times n$ matrix $A$, block size limit $w_{\max}$, and $L$-partition $Q$, find the $K$-partition $P$ minimizing the cost function $f(A, P, Q)$.*

We can minimize all of our cost functions using (10). As a warm up, notice that setting $g$ to the number of nonzero blocks in the block row,

$$g(w_p, N_{\text{index},p}, N_{\text{value},p}) = N_{\text{index},p}, \tag{11}$$

means that our algorithm will minimize (2), the number of blocks.

$$f(A, P, Q) = \sum_{p=1}^{K} g(w_p, N_{\text{index},p}, N_{\text{value},p}) = \sum_{p=1}^{K} N_{\text{index},p} = N_{\text{index}}$$

By setting

$$g(w_p, N_{\text{index},p}, N_{\text{value},p}) = (2 + 2N_{\text{index},p})s_{\text{index}} + N_{\text{value},p}s_{\text{value}}, \quad (12)$$

we have

$$\begin{aligned} f(A, P, Q) &= \sum_{p=1}^{K} g(w_p, N_{\text{index},p}, N_{\text{value},p}) \\ &= \sum_{p=1}^{K} (2 + 2N_{\text{index},p})s_{\text{index}} + N_{\text{value},p}s_{\text{value}} \\ &= (2k + 2N_{\text{index}})s_{\text{index}} + N_{\text{value}}s_{\text{value}} \\ &= s_{\text{VBR}} + (L + 3)s_{\text{index}}. \end{aligned}$$

Since $L$ is constant in this case, minimizing $f$ is equivalent to minimizing the memory usage of the resulting VBR format, $s_{\text{VBR}}$, described in (4).

Similarly, by setting

$$g(w_p, N_{\text{index},p}, N_{\text{value},p}) = (3 + N_{\text{index},p})s_{\text{index}} + N_{\text{value},p}s_{\text{value}}, \quad (13)$$

we can minimize the memory usage of the resulting 1D-VBR format, $s_{\text{1D-VBR}}$, described in (5).

If we set

$$g(w_p, N_{\text{index},p}, N_{\text{value},p}) = \alpha_{w_p} + \beta_{w_p} N_{\text{value},p} \quad (14)$$

we have that

$$\begin{aligned} f(A, P, Q) &= \sum_{p=1}^{K} g(w_p, N_{\text{index},p}, N_{\text{value},p}) \\ &= \sum_{p=1}^{K} \alpha_{w_p} + \beta_{w_p} N_{\text{value},p} \\ &= t_{VBR}, \end{aligned}$$

the affine compute cost model (6) for the VBR format.

Finally, if

$$g(w_p, N_{\text{index},p}, N_{\text{value},p}) = \alpha_{w_p} + \beta_{w_p} N_{\text{index},p}, \quad (15)$$

then we will minimize the affine compute cost model (7) for the 1D-VBR format.

## 6 HEURISTICS

Existing heuristics assume trivial column partitions when partitioning rows, and trivial row partitions when partitioning columns. Our heuristics are therefore phrased in terms of $v_i$, which is efficiently accessible in CSR format, since $A.idx$ is just a concatenation of sorted representations of each $v_i$. The cardinality of each $v_i$ is accessible via the differences between neighboring elements of $pos$.

$$v_i = \{j, A[i, j] \neq 0\}.$$

The VBR implementation in SPARSKIT groups identical rows and columns[33, 34]. The VBR implementations of OSKI and MKL PARADISO relax this notion and instead group rows which satisfy some similarity requirement [2, 41]. The OSKI algorithm for VBR examines each row $i'$ from top to bottom. Initially, all rows are ungrouped. Let $i$ be the first row in the group immediately preceding $i'$. OSKI will add $i'$ to $i$'s group if

$$\frac{|v_i \cap v_{i'}|}{\min(|v_i|, |v_i'|)} \geq \rho.$$

Otherwise, we start a new group with row $i'$. This process is repeated for the columns, producing $P$ and $Q$. The similarity metric is known as the **overlap similarity**. Another similarity metric, the cosine similarity, is sometimes used for greedy noncontiguous partitioning [35]. The OSKI code base uses a binary vector $h$ of length $n$ as a perfect hash table to calculate the size of the intersection, first setting $h_j$ to true for each $j \in v_i$, then iterating over elements of $v_{i'}$, checking to see if corresponding locations in $h$ have been set to true. When we start a new group, we must iterate through $v_i$ again to reinitialize $h$ to false. Once a partition has been produced, we iterate over the matrix again to fill the *pos* array, repeatedly filling and zeroing a binary hash vector to calculate the various important quantities.

Because $h$ will be used at most $m$ times, if we instead change $h$ to be a integer vector and store the value $i$ at each location of $h$ when calculating intersections with $v_i$, we need only iterate over $v_i$ once. Because we will build on this concept when introducing our optimal algorithm, we introduce the idea with our interpretation of the OSKI algorithm. Instead of checking whether $h_j$ has been set to true, we can check whether $h_j = i$. When we start a new group, we will use a different value of $i$ and avoid reinitialization of $h$. When checking if row $i'$ should be added to the current part, we can also add $v_{i'}$ to the hash table to avoid reading $v_{i'}$ twice. However, this may overwrite the entries which contain $i$ with $i'$, so if $h_j = i$, we can indicate that it was $i$ when we overwrote it by negating $i'$ before writing to $h$. Thus, our hash table can check the similarity between the two rows and add a new row at the same time, reducing the number of random accesses in our hash table to a minimum.

Our improved overlap algorithm is presented in Algorithm 3.

ALGORITHM 3. *Given an overlap similarity $\rho$, partition the rows of $m \times n$ matrix $A$ producing no part with more than $w_{\max}$ rows. Return $\Pi$, pos, and ofs.*

```
Require: $m > 1$, $w_{\max} > 1$, $0 < \rho \leq 1$.
1:  function OverlapPartitioner($A$, $w_{\max}$, $\rho$)
2:      Allocate uninitialized length-$n + 1$ vectors $\Pi$, $pos$, and $ofs$
3:      Allocate length-$m$ vector $h$ initialized to 0
4:      $\Pi[1] \leftarrow 1, pos[1] \leftarrow 1, ofs[1] \leftarrow 1$
5:      $i \leftarrow 1$
6:      $K \leftarrow 0$
7:      $d \leftarrow |v_1|$
8:      for $i' \leftarrow 2$ to $m$ do
9:          $d' \leftarrow d$
10:         $c \leftarrow 0$
11:         for $j \leftarrow v_{i'}$ in ascending order do
12:             if $h[j] = \pm i$ then
```

```
                c ← c + 1
                h[j] ← −i′
            else if i < h[j] then
                h[j] ← i′
            else if h[j] < −i then
                c ← c + 1
                h[j] ← −i′
            else
                d′ ← d′ + 1
                h[j] ← i′
            end if
        end for
        w ← i′ − i
        if w = w_max or c < ρ · min(|v_i|, |v′_i|) then
            K ← K + 1                          ▷ Start a new partition.
            Π[K + 1] ← i′
            i ← i′
            d ← |v_{i′}|
        else
            d ← d′                             ▷ Expand current partition.
        end if
        K ← K + 1
        w ← (n + 1) − i
        Π[K + 1] ← n + 1
    end for
    return P
end function
```

If we plan to use Algorithm 3 to produce 1D-VBR, we can play one last trick. While the value $d$ is currently unused in the algorithm, $d$ has the value $N_{\text{index},K}(A, P, :)$, the number of blocks in the current block row, on lines 28 or 36. Multiplying $d$ by $w$ produces $N_{\text{value}}K(A, P, :)$, or the number of values in the current block row. We can use the value of $d$ in these locations to build the *pos* and *ofs* arrays while we produce the partition, saving ourselves a pass over the matrix during construction of the 1D-VBR format.

Notice that when $\rho = 1$, Algorithm 3 only merges identical rows (this is the SPARSKIT heuristic). In this case, Algorithm 3 can be simplified to avoid a hash table by simply iterating over each pair of consecutive rows to check that they are equal. While this may read each row more than once, the reads are all sequential. We refer to this specialization as the StrictPartitioner. Because the row patterns in produced parts are equal, we can easily fill the *pos* and *ofs* arrays if we choose to convert to 1D-VBR.

## 7 OPTIMAL ALGORITHM

An optimal algorithm for the related problem of "restricted hypergraph partitioning" (producing contiguous partitions that reduce communication in parallel SpMV) is described by Grandjean et. al. [15]. However, this algorithm is described for cost functions which do not apply directly to Problem 2. Furthermore, this algorithm does not consider our fixed column partition. Since the algorithm is given as a reduction from the hypergraph problem to a graph problem, it requires multiple passes over the input. In situations where the runtime of the partitioner is justified with respect to the runtime savings of the target kernel, efficient algorithms that operate directly on the input matrix are desirable.

Since we are interested in both the number of blocks $N_{\text{index},p}$ and the number of values $N_{\text{value},p}$ in blocks, our algorithm uses two simultaneous hypergraph costs corresponding to each quantity. Thus, the algorithm we propose can be thought of as fusing the construction of two appropriate hypergraph problems for the fixed column partition together with the Grandjean et. al. reduction from the hypergraph to the graph formulation and the dynamic programming loop itself. We also add the part weight $w_p$ as a parameter to the cost function. The structure of our algorithm is also reminiscent of the structure of the algorithm proposed in [47].

Recall that Problem 2 asks us to compute an optimal row partition under some fixed column partition $Q$. We use dynamic programming, working upwards through the matrix $A$ and using a vector $h$ to remember the last row in which we saw each nonzero column part. This allows us to efficiently count the relevant values $N_{\text{index},p}$ and $N_{\text{value},p}$ for each candidate row part in our dynamic program. Our approach is shown in Algorithm 4. Recall that CSR format provides convenient iteration over $v_i$ in sorted order.

Algorithm 4. *Given an $m \times n$ sparse matrix $A$, a column partition $Q$, a maximum block size $w_{\max}$, and a positive-valued function $g$, compute a row partition $P$ minimizing the cost function*

$$f(A, P, Q) = \sum_{p=1}^{K} g(w_p, N_{index,p}, N_{value,p}) \tag{10}$$

```
function OptimalPartitioner(A, Q, Φ, w_max, g)
    Allocate uninitialized length-n + 1 vectors Π, Π′, c
    Allocate uninitialized length-n vectors Δ_index, Δ_value
    Allocate length-L vector h initialized to n + 1
    c[n + 1] ← 0
    for i ← m to 1 do                              ▷ Iterating Backwards!
        for j ← v_i in ascending order do
            q ← Q[j]
            u ← Φ[q + 1] − Φ[q]
            Δ_index[i] ← Δ_index[i] + 1
            Δ_value[i] ← Δ_value[i] + u
            Δ_index[h[q]] ← Δ_index[h[q]] − 1
            Δ_value[h[q]] ← Δ_value[h[q]] − u
            h[q] ← i
        end for
        d_index ← Δ_index[i]
        d_value ← Δ_value[i]
        Π′[i] ← i + 1
        c[i] ← g(1, d_index, d_value) + c[i + 1]
        for i′ ← i + 2 to min(i + w_max, n + 1) do
            d_index ← d_index + Δ_index[i′ − 1]
            d_value ← d_value + Δ_value[i′ − 1]
            w ← i′ − i
            if g(1, d_index, w · d_value) + c[i′] < c[i] then
                c[i] ← g(1, d_index, w · d_value) + c[i′]
                Π′[i] ← i′
            end if
        end for
    end for
    K ← 0
    i ← 1
    while i ≠ n + 1 do
```

```
33:         $i' \leftarrow \Pi'[i]$
34:         $K \leftarrow K + 1$
35:         $\Pi[K] \leftarrow i$
36:         $i \leftarrow i'$
37:     end while
38:     $\Pi[K+1] \leftarrow i$
39:     return $\Pi$
40: end function
```

## 7.1 Explanation

We can use dynamic programming because our problem has recursive substructure. Let $c_{i'}$ represent the cost of the best row partition of $A_{i':n,:}$ under column partition $Q$ according to $f$, as defined in (10). Then the cost of a row partition whose first part starts at $i$ and ends just before $i'$ is

$$g(i'-i, N_{\text{index}}(A_{i:(i'-1)}, \vec{1}, Q), N_{\text{value}}(A_{i:(i'-1)}, \vec{1}, Q)) + c_{i'}$$

Thus, the first row part in the minimal partition of $A_{i:n,:}$ ends at

$$\operatorname*{arg\,min}_{i<i'\leq i+w_{\max}} \; g(i'-i, N_{\text{index}}(A_{i:(i'-1)}, \vec{1}, Q), N_{\text{value}}(A_{i:(i'-1)}, \vec{1}, Q))+c_{i'}$$

This gives us our dynamic programming formulation. All that is left to explain is how we calculate these $N_{\text{index}}$ and $N_{\text{value}}$ quantities.

We have defined $N_{\text{index}}(A_{i:(i'-1)}, \vec{1}, Q)$ as the number of nonzero blocks induced by putting rows $i$ through $i'-1$ in the same row part. This is equivalent to the number of distinct column parts of nonzeros in rows $i$ through $i'-1$, or

$$d_{\text{index}, i'} = |\gamma_i \cup \ldots \cup \gamma_{i'-1}|.$$

We have defined $N_{\text{value}}(A_{i:(i'-1)}, \vec{1}, Q)$ as the total size of the nonzero blocks induced by putting rows $i$ through $i'-1$ in the same row part. Notice that $N_{\text{value}}(A_{i:(i'-1)}, \vec{1}, Q)/(i'-i)$ is equivalent to the sum of the sizes of each distinct column part of nonzeros in rows $i$ through $i'-1$, or

$$d_{\text{value}, i'} = \sum_{q \in \gamma_i \cup \ldots \cup \gamma_{i'-1}} u_q.$$

After completing the loop on line 7, $\Delta_{\text{index}}$ and $\Delta_{\text{value}}$ satisfy

$$\Delta_{\text{index}, i'} = d_{\text{index}, i'} - d_{\text{index}, (i'-1)}$$
$$\Delta_{\text{value}, i'} = d_{\text{value}, i'} - d_{\text{value}, (i'-1)}$$

Notice that as we expand a row part starting at row $i$, the set of nonzero column blocks that belong to our row part can only grow. If our candidate row part starts at $i$, we can think of $\Delta_{\text{index}, i'}$ as the number of additional nonzero column blocks that we encounter as we move the end of our row part from $i'-1$ to $i'$. Similarly, we can think of $\Delta_{\text{value}, i'}$ as the the number of additional columns in nonzero blocks that we encounter as we move the end of our row part from $i'-1$ to $i'$.

When we change our row start to $i-1$ in the next iteration of the loop on line 6, we need to update our $\Delta$ arrays. We use the vector $h$ to do this. We define $h_q$ as the most recent row $i' > i$ for which the column part $q$ contains a nonzero block. Notice that part $q$ contributes to the count of $d_{\text{index}, i'}$ and $d_{\text{value}, i'}$ precisely when $i' > h_q$. Thus, if row $i-1$ contains a nonzero in part $q$, then we must decrement $\Delta_{\text{index}, h_q}$ and increment $\Delta_{\text{index}, i}$, to reflect that this part will now be in all partitions starting at $i-1$. Similarly, we must update our $\Delta_{\text{value}}$ vector, except with the size $u_q$ of part $q$.

After the loop on line 7 updates $\Delta_{\text{index}}$ and $\Delta_{\text{value}}$, we can compute $d_{\text{index}}$ and $d_{\text{value}}$ for each candidate part in the loop on line 20 by simply adding the corresponding $\Delta$ quantities, and therefore compute the minimum-cost part.

Finally, at the start of the loop on line 32, $\Pi'_i$ contains the start of the next block $i'$ in an optimal partition of $A_{i:n,:}$. This cleanup loop follows each pointer starting at row 1 and sets $\Pi$ to its correct value, as well as computing $K$.

## 7.2 Runtime, Optimizations, and Extensions

The body of the loop at line 7 can be executed in constant time and will be repeated at most once for each nonzero in $A$. The body of the loop at line 20 can be executed in constant time and will be repeated at most $w_{\max}$ times for each row in $A$. Initialization takes $O(m+n)$ time. The cleanup loop is accomplished in $O(m)$ time. Thus, Algorithm 4 runs in $O(m \cdot w_{\max} + n + N)$ time.

Because the outer dynamic program loop on line 20 works backwards, all of the innermost loops access memory in storage order. While we show $\Pi$ and $\Pi'$ as separate vectors in Algorithm 4, an implementation can reuse the $\Pi'$ vector to store $\Pi$, overwriting it in place during the last loop on line 32.

This algorithm can be improved if $Q$ is the trivial partition and we intend to convert to 1D-VBR. If $Q$ is trivial, $Q[j] = j$, $u_q$ is always 1, and $d_{\text{index}} = d_{\text{value}}$, so we only need to compute one of the $d$ values and the corresponding $\Delta$ vectors.

If we store the values $d_{\text{index}}$ and $d_{\text{value}}$ corresponding to each potential part recorded with $\Pi'$, then we can compute the $pos$ and $ofs$ arrays for 1D-VBR format as we unravel the pointers in the loop on line 32, allowing us to allocate memory for the $idx$ and $val$ arrays and potentially parallelize the conversion step without having to walk the matrix again. Computing $pos$ and $ofs$ will also help to compute the $ptr$ array for VBR format.

If the blocks in our format are themselves sparse [7, 8, 19], we may be interested in a cost function $g$ which counts both the number of nonzero blocks and the number of reads from $x$ required for processing the block row. This corresponds to the value of $N_{\text{index}}$ for both a nontrivial column partition and the trivial column partition. If a cost model depends on $N_{\text{index}}$ or $N_{\text{value}}$ for more than one simultaneous column partition, we suggest using more than one copy of $h$, $\Delta$, and $d$ to optimize this expanded cost model.

## 8 CONVERSION

After producing a partition, we need to actually convert the matrix from CSR to VBR or 1D-VBR format. Because all of our partitioning algorithms can also compute $pos$ and $ofs$, we can allocate the exact amount of memory needed for $idx$, $val$, and possibly $ptr$, and all we need are the algorithms to fill them. A fast conversion algorithm is just as important as a fast partitioner, and we find that our conversion algorithms are similarly expensive to the partitioners we have proposed. If we need to parallelize the conversion algorithm, each process can start at the locations defined by $pos$ and $ofs$.

If all the rows in each group are identical, as is the case with partitions produced by the StrictPartitioner, then the nonzero patterns from the CSR representation can be copied directly from any row in each part to form the $idx$ array in 1D-VBR format. The $val$ array can be filled straightforwardly using offsets from $ofs$ and indexing expressions to access elements within each row.

If, however, all the rows in each group are not identical, then we must merge the nonzero patterns of each row in a part to produce the nonzero block patterns. If each CSR row contains a sorted list of the elements in $v_i$, then our goal is to form a sorted list of the elements in $\bigcup_{i|P_i=p} v_i$. This is a similar problem to merging $w$ sorted lists. Algorithms exist to solve such a problem in $O(\log(w)(|\bigcup_{i|P_i=p} v_i|) + \sum |v_i|)$ time [16, "HeapMerge"]. Since we also need to fill all $w \cdot (|\bigcup_{i|P_i=p} v_i|)$ entries of the *val* array with either nonzeros or explicit zeros, we instead use a linear search over the rows to find the minimum index, then iterate over the rows to fill the corresponding elements of *idx* and *val* [16, "LinearSearchMerge"]. The direct merge algorithm is the simpler choice when $w$ is small, which we have assumed it is. The algorithm for producing block rows in VBR or 1D-VBR format is similar enough to [16, "LinearSearchMerge"] that we omit it. It is enough to know that the number of operations performed by the conversion algorithm is proportional to the size of the resulting format.

## 9 RESULTS

We ran our programs on the "Haswell" partition of the "Cori" NERSC Supercomputer. We used a single core of a 16-core Intel®Xeon®Processor E5-2698 v3 running at 2.3 GHz with 32 KB of L1 cache per core, 256 KB of L2 cache per core, 41 MB of shared L3 cache, and 128 GB of memory. This CPU supports the AVX2 instruction set, meaning that it supports SIMD processing with 256 bit vector lanes, or 4 elements of 64 bits each.

All kernels were implemented[2] in Julia 1.4.1 [6]. Because Julia is compiled just-in-time, it enjoys powerful metaprogramming capabilities. This allowed us to create a custom SpMV subkernel for each block size in our 1D-VBR SpMV kernel. Hard-coding block sizes allows the compiler to perform important optimizations like loop unrolling. In all of our tests, our maximum block size was $w_{\max} = 8$ because we found that further increasing $w_{\max}$ did not increase performance. If the block size of a row part was 1, we used scalar instructions. Otherwise, we used one or two vectors of size 4 to process the row part. We used the SIMD.jl library to emit explicit LLVM vector instructions for each block size [37]. Since our matrices were real-valued, the value datatypes were Binary64 floating point numbers [1], and the index datatypes were 64 bit signed integers.

We benchmark with a warm cache, meaning that we run the kernel once before beginning to measure it. We run the Julia garbage collector before taking each sample. After warming up the cache, each kernel is sampled one million times or until 10 seconds of measurement time is exceeded (we allow the kernel to complete before stopping), whichever happens first. All benchmarks use the minimum sampled time.

Our experiments are motivated by the application of sparse iterative solvers of linear systems of equations, where the same matrix is multiplied by a vector many times during the solve. Therefore, we focus on the 1626 real-valued matrices in the SuiteSparse Matrix Collection [13]. Not all of these matrices have block structure. We view these results not just as an evaluation of our partitioners and cost functions, but also an exploration of the block distribution of real-world matrices.

We compare several partitioners. The "Reference" label refers to the case where no partitioner is used. The "Strict" label refers to the StrictPartitioner algorithm, a specialization of Algorithm 3 when $\rho = 1$. Note that we also use the specialized conversion routine for the "Strict" case. We use Algorithm 3 (the OverlapPartitioner) directly with 3 different values of $\rho$ (0.9, 0.8, and 0.7), each under the label "Overlap ($rho = \rho$)." Our OptimalPartitioner algorithm (Algorithm 4) is tested with 3 different cost models. The "Minimize Memory" label refers to minimizing the size of the 1D-VBR representation (13). The "Minimize Compute" label refers to minimizing the modeled computation time for 1D-VBR representation (15). The "Minimize Blocks" label refers to minimizing the number of blocks (11).

For each partitioner, we measure the time to partition the matrix, convert from CSR to 1D-VBR format, and to multiply the matrix by a vector. Since our partitioners are intended to accelerate an SpMV kernel, they must have a similar runtime. We therefore normalize our runtimes to a standard, unblocked CSR SpMV kernel. We used the same straightforward implementation that is used by the Julia standard library. We measure the memory usage and number of blocks in the resulting 1D-VBR formats, normalized to the memory usage and number of nonzeros in the CSR format. Table 1 shows the distribution of normalized running times for each partitioning method over all tested matrices.

Since we are using our algorithms in the context of a sparse iterative solver, where we partition once and multiply several times, using a partitioner only produces an overall speedup after a certain number of SpMV executions. If $t_{\text{1D-VBR partition}}$ and $t_{\text{1D-VBR convert}}$ are the measured times to partition and convert, and $t_{\text{1D-VBR multiply}}$ is the time to multiply once, then if we are to multiply $M$ times, the total time to perform $M$ multiplications is

$$t_{\text{1D-VBR total}} = t_{\text{1D-VBR partition}} + t_{\text{1D-VBR convert}} + M \cdot t_{\text{1D-VBR multiply}} \tag{16}$$

If the time required to multiply in CSR is $t_{\text{CSR multiply}}$, then partitioning is the faster approach only if one plans to perform

$$M_{\text{critical}} = \frac{t_{\text{1D-VBR partition}} + t_{\text{1D-VBR convert}}}{t_{\text{CSR multiply}} - t_{\text{1D-VBR multiply}}} \tag{17}$$

multiplications. We refer to the value $M_{\text{critical}}$ as the **critical point**.

Table 1 also shows the distribution of normalized memory usage, normalized blocks, and the critical points for each partitioning method over the test set. We see that minimizing our empirical cost model sometimes increases memory usage, but that modest memory savings may obtained by directly minimizing memory usage. Partitions with minimal modeled cost also create very few blocks, and tend to increase memory usage slightly. Minimizing our empirical cost model produces runtime savings quicker and more often than the other tested approaches.

Interestingly, simply minimizing the number of blocks is quite effective. However, minimizing our empirical cost model produces consistently faster multiply times. The OverlapPartitioner produced partitions slightly faster than our own OptimalPartitioner, but no partitioning method was as fast as the StrictPartitioner.

[2]Code is available at https://github.com/peterahrens/SparseMatrix1DVBCs.jl

Similarly, the specialized conversion routine for the StrictPartitioner was the fastest. The other partitioners used the same conversion routines, so the differences in conversion time reflect the sizes of the 1D-VBR representation, which we can verify by comparing the relevant columns of Table 1.

Figure 2 shows how often (as a percentage of tested matrices) each method produced the fastest total time $t_{\text{1D-VBR total}}$ (defined in (16)) for increasing values of $M$. This figure shows that no partitioning is worth it for situations requiring less than 5 multiplies, and that the StrictPartitioner was the only partitioner that ran fast enough to produce overall speedups in regimes requiring less than 10 multiplies. As the number of multiplies increases, we start to see several different partitioning techniques become practical, and by 100 multiplies, using the OptimalPartitioner with our modeled compute cost function (15) is often the best partitioning approach. If approximately 1000 multiplies are needed, partitioning becomes effective on almost half of the tested matrices. While minimizing the modeled compute cost was effective on matrices that fit inside L3 cache, minimizing memory was more effective on larger matrices. The L3 cache size of this machine is $1.2 \cdot 10^7$ bytes. Minimizing memory usage was the better cost model in 9.69% of matrices whose CSR representation used less than half of the L3 cache. However, minimizing memory was better on 61.6% of matrices whose CSR representation used more than half of the L3 cache.

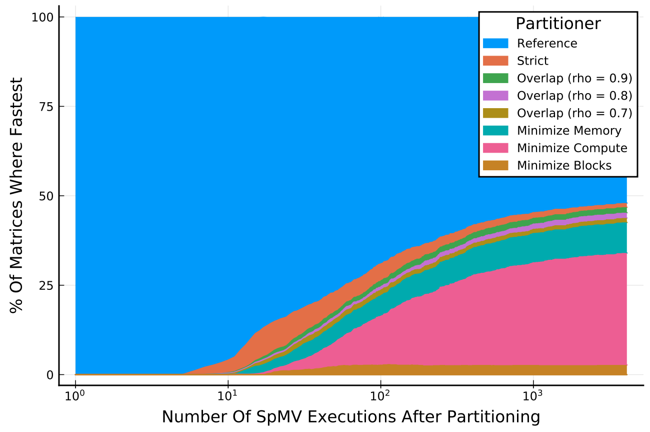


**Figure 2: A profile of how often each partitioner was the fastest. For each partitioner, we show the fraction of matrices in our test set for which that partitioner resulted in the lowest total time $t_{\text{1D-VBR total}}$ to partition, convert from CSR to 1D-VBR format, and multiply in 1D-VBR format $M$ times, as described in (16). The "Reference" partitioner corresponds to multiplying in unpartitioned CSR. The horizontal axis represents the value of $M$ in logarithmic scale. The vertical axis displays the fraction of test matrices as a percentage.**

## 10 CONCLUSIONS

We presented a fast and optimal algorithm for partitioning rows in a VBR matrix. We apply the algorithm to a novel specialization of VBR with only row partitions, 1D-VBR. We showed that our algorithm works well in practice on a large test set from the SuiteSparse collection. For iterative solvers, the same matrix is used in many iterations. For nonlinear solvers, a sequence of linear systems are often solved with the same sparsity pattern. We showed that the setup cost of partitioning and conversion can typically be amortized over many SpMV invocations.

We believe that a similar approach as Yaşar [45] can be taken to extend our optimal one dimensional algorithm to a two dimensional heuristic by applying it repeatedly, alternating partitioning the rows and columns. While we have proven the two dimensional problem to be NP-Hard to solve optimally, our proof did not exclude the existence of efficient approximation algorithms. Additionally, we hope to see this partitioner tested on other row and column aligned contiguous partitioning problems and cost models, such as block decompositions with sparse blocks [7, 8, 19].

## ACKNOWLEDGMENTS

This work was supported by a Department of Energy Computational Science Graduate Fellowship, DE-FG02-97ER25308. This work also funded by the Department of Energy's Exascale Computing Program (ECP). Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

## REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229
[2] 2020. *Developer Reference for IntelÂő Math Kernel Library - Fortran.* Technical Report 097. IntelÂő. 2556–2557 pages. https://software.intel.com/en-us/mkl-developer-reference-fortran
[3] Peter Ahrens. 2019. *A Parallel Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats.* Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/121653
[4] P. Ahrens, H. Xu, and N. Schiefer. 2018. A Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 546–556. https://doi.org/10.1109/IPDPS.2018.00064
[5] C.J. Alpert and A.B. Kahng. 1995. Multiway partitioning via geometric embeddings, orderings, and dynamic programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14, 11 (Nov. 1995), 1342–1358. https://doi.org/10.1109/43.469661
[6] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (Jan. 2017), 65–98. https://doi.org/10.1137/141000671
[7] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536313 ISSN: 1530-2075.
[8] Aydin BuluÃğ, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. ACM Press, 233. https://doi.org/10.1145/1583991.1584053
[9] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. 2007. Performance Optimization and Modeling of Blocked Sparse Kernels. *The International Journal of High Performance Computing Applications* 21, 4 (Nov. 2007), 467–484. https://doi.org/10.1177/1094342007083801

| Partitioner | Partition Time | | | | | Convert Time | | | | | Multiply Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 |
| **Reference** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| **Strict** | 0.233 | 0.399 | 0.597 | 0.783 | 1.01 | 2.62 | 2.89 | 3.31 | 3.88 | 4.27 | 0.976 | 1.14 | 1.35 | 1.54 | 1.66 |
| **Overlap (rho = 0.9)** | 2.39 | 2.9 | 3.55 | 4.18 | 4.57 | 3.14 | 3.55 | 4.25 | 5.29 | 6.23 | 0.711 | 0.967 | 1.27 | 1.45 | 1.59 |
| **Overlap (rho = 0.8)** | 2.41 | 2.9 | 3.56 | 4.18 | 4.57 | 3.24 | 3.65 | 4.42 | 5.42 | 6.53 | 0.724 | 0.972 | 1.25 | 1.44 | 1.59 |
| **Overlap (rho = 0.7)** | 2.4 | 2.91 | 3.58 | 4.19 | 4.57 | 3.28 | 3.78 | 4.59 | 5.56 | 6.68 | 0.719 | 0.965 | 1.23 | 1.42 | 1.59 |
| **Minimize Memory** | 2.66 | 3.33 | 4.83 | 6.33 | 7.57 | 3.23 | 3.53 | 4.03 | 4.77 | 5.41 | 0.656 | 0.931 | 1.19 | 1.39 | 1.54 |
| **Minimize Compute** | 2.74 | 3.54 | 5.27 | 7.01 | 8.47 | 4.83 | 5.99 | 7.21 | 8.62 | 9.9 | 0.628 | 0.846 | 1.02 | 1.19 | 1.35 |
| **Minimize Blocks** | 2.54 | 3.16 | 4.38 | 5.56 | 6.58 | 4.83 | 6.39 | 8.45 | 11.4 | 14.3 | 0.679 | 0.936 | 1.25 | 1.53 | 1.83 |

| Partitioner | Memory Used | | | | | Number of Blocks | | | | | Critical Point | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 | O. 1 | Q. 1 | Med. | Q. 3 | O. 7 |
| **Reference** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| **Strict** | 0.914 | 1.03 | 1.1 | 1.17 | 1.23 | 0.741 | 0.98 | 1 | 1 | 1 | 170 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| **Overlap (rho = 0.9)** | 0.85 | 1 | 1.13 | 1.23 | 1.32 | 0.288 | 0.538 | 0.79 | 0.95 | 1 | 27.3 | 273 | $\infty$ | $\infty$ | $\infty$ |
| **Overlap (rho = 0.8)** | 0.852 | 1 | 1.13 | 1.24 | 1.32 | 0.28 | 0.525 | 0.754 | 0.916 | 0.999 | 29 | 335 | $\infty$ | $\infty$ | $\infty$ |
| **Overlap (rho = 0.7)** | 0.863 | 1.01 | 1.14 | 1.25 | 1.33 | 0.276 | 0.519 | 0.732 | 0.892 | 0.997 | 30 | 244 | $\infty$ | $\infty$ | $\infty$ |
| **Minimize Memory** | 0.738 | 0.929 | 1.04 | 1.13 | 1.17 | 0.35 | 0.61 | 0.784 | 0.939 | 0.997 | 21.8 | 120 | $\infty$ | $\infty$ | $\infty$ |
| **Minimize Compute** | 0.85 | 1.2 | 1.46 | 1.67 | 1.84 | 0.25 | 0.412 | 0.626 | 0.74 | 0.807 | 23.9 | 74.8 | $\infty$ | $\infty$ | $\infty$ |
| **Minimize Blocks** | 0.89 | 1.31 | 1.77 | 2.24 | 2.59 | 0.23 | 0.365 | 0.575 | 0.698 | 0.746 | 31 | 187 | $\infty$ | $\infty$ | $\infty$ |

**Table 1: The first table summarizes the distribution of the normalized times required to partition, convert from CSR to 1D-VBR format, and multiply in 1D-VBR format for over each matrix in our test set of matrices. The "Reference" partitioner corresponds to multiplying in unpartitioned CSR, and all times measured for each matrix are normalized to this CSR multiply time. The second table summarizes memory usage and number of blocks in the resulting 1D-VBR formats. The memory usage and number of blocks for each matrix are normalized to the corresponding attributes of the unpartitioned CSR representation. We also display the distribution of critical points for each partitioner over all the matrices. The critical point is defined in** (17) **as the number of multiplies needed to justify partitioning. The "O. 1", "Q. 1", "Med", "Q. 3", and "O. 7" columns refer to the least octile, least quartile, median, greatest quartile, and greatest octile, respectively, of the corresponding distribution.**

[10] U. V. Catalyurek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (July 1999), 673–693. https://doi.org/10.1109/71.780863

[11] Xinhai Chen, Peizhen Xie, Lihua Chi, Jie Liu, and Chunye Gong. 2018. An efficient SIMD compression format for sparse matrix-vector multiplication. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4800. https://doi.org/10.1002/cpe.4800

[12] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM SIGPLAN Notices* 45, 5 (May 2010), 115. https://doi.org/10.1145/1837853.1693471

[13] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (Nov. 2011), 1–25. https://doi.org/10.1145/2049662.2049663

[14] Ryan Eberhardt and Mark Hoemmen. 2016. Optimization of Block Sparse Matrix-Vector Multiplication on Shared-Memory Parallel Architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 663–672. https://doi.org/10.1109/IPDPSW.2016.42

[15] Anael Grandjean, Johannes Langguth, and Bora UÃğar. 2012. On Optimal and Balanced Sparse Matrix Partitioning Problems. In *2012 IEEE International Conference on Cluster Computing*. 257–265. https://doi.org/10.1109/CLUSTER.2012.77 ISSN: 2168-9253.

[16] W.A. Greene. 1991. k-way merging and k-ary sorts. In *[Proceedings] 1991 Symposium on Applied Computing*. 197–. https://doi.org/10.1109/SOAC.1991.143874 ISSN: null.

[17] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (Sept. 1978), 250–269. https://doi.org/10.1145/355791.355796

[18] Bruce Hendrickson and Tamara G. Kolda. 2000. Graph Partitioning Models for Parallel Computing. *Parallel Comput.* 26, 12 (Nov. 2000), 1519–1534. https://doi.org/10.1016/S0167-8191(00)00048-X

[19] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712 event-place: Washington, District of Columbia.

[20] Eun-Jin Im. 2000. *Optimizing the Performance of Sparse Matrix-Vector Multiplication.* Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2000/5556.html

[21] Eun-Jin Im and Katherine Yelick. 2001. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Computational Science âĂŤ ICCS 2001 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 127–136. https://doi.org/10.1007/3-540-45545-0_22

[22] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications* 18, 1 (Feb. 2004), 135–158. https://doi.org/10.1177/1094342004041296

[23] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2009. A Comparative Study of Blocking Storage Methods for Sparse Matrices on Multicore Architectures. In *2009 International Conference on Computational Science and Engineering*. IEEE, Vancouver, BC, Canada, 247–256. https://doi.org/10.1109/CSE.2009.223

[24] V. Karakasis, G. Goumas, and N. Koziris. 2009. Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In *2009 International Conference on Parallel Processing*. 356–364. https://doi.org/10.1109/ICPP.2009.21

[25] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20âĂŞ22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger (Eds.). Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[26] Brian W. Kernighan. 1971. Optimal Sequential Partitions of Graphs. *Journal of the ACM (JACM)* 18, 1 (Jan. 1971), 34–40. https://doi.org/10.1145/321623.321627

[27] B.C. Lee, R.W. Vuduc, J.W. Demmel, and K.A. Yelick. 2004. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. IEEE, 169–176 vol.1. https://doi.org/10.1109/ICPP.2004.1327917

[28] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. 2007. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing* 18, 3 (May

2007), 297–311. https://doi.org/10.1007/s00200-007-0038-9

[29] Christos H. Papadimitriou and Mihalis Yannakakis. 1991. Optimization, approximation, and complexity classes. *J. Comput. System Sci.* 43, 3 (Dec. 1991), 425–440. https://doi.org/10.1016/0022-0000(91)90023-X

[30] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99)*. ACM, New York, NY, USA. https://doi.org/10.1145/331532.331562 event-place: Portland, Oregon, USA.

[31] Javed Razzaq, Rudolf Berrendorf, Jan P. Ecker, Soenke Hack, Max Weierstall, and Florian Manuss. 2017. The DynB Sparse Matrix Format Using Variable Sized 2D Blocks for Efficient Sparse Matrix Vector Multiplications with General Matrix Structures. *International Journal On Advances in Intelligent Systems* 10, 1 and 2 (June 2017), 48–58. http://thinkmind.org/index.php?view=article&articleid=intsys_v10_n12_2017_5

[32] Karin Remington and Roldan Pozo. 1996. *NIST Sparse BLAS: UserâĂŹs Guide.* Technical Report. NIST. ftp://gams.nist.gov/pub/karin/spblas/uguide.ps.gz

[33] Youcef Saad. 1990. *SPARSKIT: A basic tool kit for sparse matrix computations.* Technical Report. https://ntrs.nasa.gov/search.jsp?R=19910023551

[34] Youcef Saad. 1994. *SPARSKIT: a basic tool kit for sparse matrix computations - Version 2.* https://www-users.cs.umn.edu/~saad/PDF/RIACS-90-20.pdf

[35] Yousef Saad. 2003. Finding Exact and Approximate Block Structures for ILU Preconditioning. *SIAM Journal on Scientific Computing* 24, 4 (Jan. 2003), 1107–1123. https://doi.org/10.1137/S1064827501393393

[36] Yousef Saad. 2003. *Iterative methods for sparse linear systems* (2nd ed.). SIAM, Philadelphia.

[37] Erik Schnetter, Takafumi Arakaki, Valentin Churavy, Kristoffer Carlsson, Nicolau Leal Werneck, Steve Kelly, Gunnar FarnebÃďck, Miguel Raz GuzmÃąn Macedo, Matt Bauman, Kenta Sato, and Elliot Saba. 2019. eschnett/SIMD.jl: v2.8.0. https://doi.org/10.5281/zenodo.3356766

[38] Manu Shantharam, Anirban Chatterjee, and Padma Raghavan. 2011. Exploiting dense substructures for fast sparse matrix vector multiplication. *The International Journal of High Performance Computing Applications* 25, 3 (Aug. 2011), 328–341. https://doi.org/10.1177/1094342011414748

[39] Virginia Vassilevska and Ali Pinar. 2004. Finding Nonoverlapping Dense Blocks of a Sparse Matrix. (Feb. 2004). https://escholarship.org/uc/item/2s3680h5

[40] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. 2002. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. IEEE, 26–26. https://doi.org/10.1109/SC.2002.10025

[41] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (Jan. 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[42] Richard W. Vuduc. 2004. *Automatic performance tuning of sparse matrix kernels.* Ph.D. Dissertation. University of California, Berkeley, CA, USA. http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

[43] Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *High Performance Computing and Communications (Lecture Notes in Computer Science)*, Laurence T. Yang, Omer F. Rana, Beniamino Di Martino, and Jack Dongarra (Eds.). Springer, Berlin, Heidelberg, 807–816. https://doi.org/10.1007/11557654_91

[44] Helen Xu. 2018. *Fill Estimation for Blocked Sparse Matrices and Tensors.* Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/117816

[45] Abdurrahman YaÅ§ar and ÃIJmit V. ÃĞatalyÃijrek. 2019. Heuristics for Symmetric Rectilinear Matrix Partitioning. *arXiv:1909.12209 [cs]* (Sept. 2019). http://arxiv.org/abs/1909.12209 arXiv: 1909.12209.

[46] A. N. Yzelman. 2015. Generalised Vectorisation for Sparse Matrix: Vector Multiplication. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA$^3$ '15)*. ACM, New York, NY, USA, 6:1–6:8. https://doi.org/10.1145/2833179.2833185

[47] Louis H. Ziantz, Can C. ÃŰzturan, and Boleslaw K. Szymanski. 1994. Run-time optimization of sparse matrix-vector multiplication on SIMD machines. In *PARLE'94 Parallel Architectures and Languages Europe (Lecture Notes in Computer Science)*, Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, Berlin, Heidelberg, 313–322. https://doi.org/10.1007/3-540-58184-7_111

## A FINDING OPTIMAL VBR PARTITIONS IS NP-HARD

In this section, we show that Problem 1, finding the row and column partition which maximizes a cost model $f(A, P, Q)$ of the VBR representation of some matrix $A$, is NP-Hard by reduction from the Maximum Cut problem, one of Karp's 21 NP-Complete problems[25, 29].

We restate the Maximum Cut problem here for convenience.

Problem 3 (Maximum Cut). *Given an undirected graph $G = (V, E)$ with $m$ nodes and $n$ edges, split $V$ into two sets $C_1 \subset V$ and $C_2 \subset V$ where $C_1 \cap C_2 = \emptyset$ and the number of edges between $C_1$ and $C_2$ is maximized.*

Theorem 1. *Problem 1 is NP-Hard for any $w_{\max} \geq 2$ and cost functions of the form*

$$f(A, P, Q) = s \cdot N_{index} + N_{value} \tag{8}$$

*where $s \geq 1$ is constant.*

Proof. Assume we are given an instance of Maximum Cut (Problem 3). We first define a matrix $A$ in terms of $G$ and $s$, then show a correspondence between a class of partitions of $A$ and cuts in $G$. Finally, we show that the $P$ and $Q$ which optimize (8) correspond to a maximum cut through $G$.

Let $A$ be an $rm \times rn$ matrix of zeros and nonzeros, where nonzeros are represented with $x$. Unless stated otherwise, entries of $A$ are defined to be zero. Fix an ordering of the edges of $G$, and let $e_j = (i_1, i_2)$ where $i_1 < i_2$ be the $j^{th}$ edge in this ordering of $G$. We will insert an $l \times l$ gadget into $A$ at each endpoint of $e_j$, where

$$l = 3 + r_1 + (1 + r_1)r_2 + 2(1 + r_1)r_3 \tag{18}$$
$$r_1 = \lfloor s + 1 \rfloor \tag{19}$$
$$r_2 = 32 \tag{20}$$
$$r_3 = \lceil 28s - 10 \rceil \tag{21}$$

These constants depend on the relative weights that $s$ assigns to each block and the size of each block. They are larger to make the proof shorter; making them large allows us to upper bound $r_3$ by $28s - 9$ when calculating the cost of each gadget. To give an example of some smaller constants, if $s = 1$, we can use $r_1 = 2, r_2 = 3, r_3 = 1$.

If we think of $A$ as being tiled with $l \times l$ tiles, we insert the gadget $B_1$ at the intersection of the $i_1^{th}$ tile row and $j^{th}$ tile column.

$$B_1 = \begin{bmatrix}
x & & & \cdots & x & \cdots & x & \cdots & & \cdots \\
& x & & \cdots & x & \cdots & & \cdots & & \cdots \\
& & x & \cdots & x & \cdots & & \cdots & x & \cdots \\
\vdots & \vdots & \vdots & & & & & & & \\
x & x & x & & & & & & & \\
\vdots & \vdots & \vdots & & & & & & & \\
x & & & & & & & & & \\
\vdots & \vdots & \vdots & & & & & & & \\
& & x & & & & & & & \\
\vdots & \vdots & \vdots & & & & & & &
\end{bmatrix}$$

We insert the gadget $B_2$ at the intersection of the $i_2^{th}$ tile row and $j^{th}$ tile column.

$$B_2 = \begin{bmatrix} & & x & \cdots & x & \cdots & x & \cdots & & \cdots \\ & x & & \cdots & x & \cdots & & \cdots & & \cdots \\ x & & & \cdots & x & \cdots & & \cdots & x & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ x & & & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ & & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

where the upper left patterns occur once, the patterns second to the right and the bottom are repeated $r_2$ times, and the two rightmost and bottommost patterns are repeated $r_3$ times. All patterns are followed by $r_1$ rows or columns of zeros. Figure 3 gives an example of $A$ for some $G$.

The gadgets are identical except for the upper left $3 \times 3$ pattern. Thus, the patterns on the top of the gadgets are column-aligned and the patterns on the left are row-aligned across gadget rows and gadget columns. We refer to the resulting fully zero $r_1$ rows (resp. columns) as **filler** rows (resp. columns).

We start by arguing that it is never optimal to produce a partition with a row part that contains both filler rows and non-filler rows. A symmetric argument holds for the columns.

First, consider the case where the row part contains filler rows on the top or bottom. Separating these rows from the part reduces the sizes $N_{\text{value}}$ of the blocks in that part without changing the number $N_{\text{index}}$ of blocks, so the part cannot have been optimal.

Second, if the row part does not start or end with a filler row, it must contain filler rows. Since the filler rows in $A$ occur in contiguous groups of size $r_1$, this part must contain such a group. Consider a block in this part. If the block contains nonzeros on only one side of the filler rows, then separating the rows strictly reduces the size of the block without adding any new blocks. If the block contains nonzeros on both sides of the filler rows, then removing the rows creates a block, but deletes at least $r_1$ explicitly stored zero values. Since $r_1 > s$, separating these filler rows still reduces the cost of the partition, so it cannot have been optimal.

Therefore, optimal partitions do not merge different patterns together. We won't concern ourselves with whether the filler rows have been merged together, since it doesn't change the cost function. Since the patterns on top consist of only one column, and the patterns on the side consist of only one row, the only undetermined piece of our optimal partition is the partition of the first three rows and columns of each gadget row and gadget column, respectively.

There remains only four cases for the rows. Either each row lies in a separate part, all rows share a part, the first two rows share a part, or the last two rows share a part. A symmetric argument holds for the columns.

Sections A.1 and A.2 exhaustively check that for all cases where both the first three rows and columns of a gadget have two parts each ($P, Q \in \{[1, 1, 2, \ldots], [1, 2, 2, \ldots]\}$),

$$f(B_1, P, Q) \leq 146 + 263s + 112s^2 \tag{22}$$

and that in all the other cases,

$$f(B_1, P, Q) \geq 147 + 263s + 112s^2 \tag{23}$$

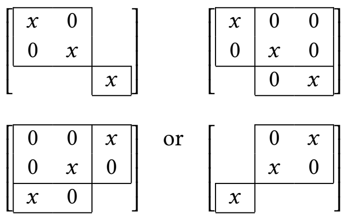
The exhaustive proofs for $B_2$ are symmetric, so we omit them.

Assume we start with a row and column partition where only the first three rows and columns of each gadget share parts in the partition. For every row or column part with three members, we split off one row or column into a different part. For any case where the first three rows or columns of a gadget row or column all belong to different parts, we merge two of the rows or columns. For every gadget in our initial partition whose first three rows and columns had two parts each, it's blocks will be unchanged. For every other gadget, the cost will be strictly reduced. Thus, optimal partitions only merge pairs of rows and columns, and these pairs occur in the first three rows or columns of each gadget row or column. In this case, the cost of every subassembly is the same except for the upper left $3 \times 3$ pattern of each gadget. Therefore, the remainder of the argument focuses on these assemblies.

At this point, we can establish a correspondence between cuts in the graph and partitions. Let $(C_1, C_2)$ be a cut in the graph. We will define a row partition $P$ corresponding to this cut. Unless stated otherwise, rows in this partition are assigned to distinct parts. If a vertex $i$ lies in $C_1$, then we merge the first and second rows of the corresponding gadget row. If our vertex $i$ lies in $C_2$, then we merge the second and third rows of the gadget row. Consider the gadgets corresponding to an edge $e_j = (i_1, i_2)$. Notice that if vertices $i_1$ and $i_2$ lie in the same part, $C_1$ for example, we have one of the following situations:

$v_i \in V_1, P = (1, 1, 2)$

$v_{i'} \in V_1, P = (1, 1, 2)$

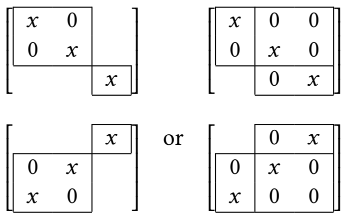


$e_j, Q = (1, 1, 2)$ $\quad$ $e_j, Q = (1, 2, 2)$

The cost of either arrangement is $f = 13 + 5s$. However, if vertices $i_1$ and $i_2$ lie in different parts, $C_1$ and $C_2$, for instance, the situation is as follows:

$v_i \in V_1, P = (1, 1, 2)$

$v_{i'} \in V_2, P = (1, 2, 2)$



$e_j, Q = (1, 1, 2)$ $\quad$ $e_j, Q = (1, 2, 2)$

Since these are the only two gadgets in the column corresponding to this edge, we are free to choose a column partition. The above partition of minimal cost has a cost of $f = 10 + 4s$, less than the case where the vertices shared an edge.

Thus, the cost of an optimal column partition corresponding to the row partition representation of a cut can be expressed as a constant minus $3 + s$ times the number of cut edges. Since there is a bijection between cuts and our "pairwise" row partitions (one of which we know to be optimal), producing an optimal partition of rows and columns is equivalent to finding the maximum cut in $G$. If we treat $s$ as a constant, our reduction imposes only a constant factor of overhead in $m$ and $n$, and Problem 3 is reducible to Problem 1 in polynomial time. □

Notice that our proof of Theorem 1 makes no assumption on the size of $w_{\max}$, using only $w_{\max} \geq 2$ and enforcing pairwise partitions with the cost function. We chose this proof technique since the cost function is realistic and in some situations there may be no limit on the sizes of blocks ($w_{\max} = n$). However, it requires large gadgets when $s$ is large. Alternative gadgets can be used in situations where $s$ is large by instead using $w_{\max}$ to constrain the sizes of blocks and choosing different gadgets. This leads us to Theorem 2, which corresponds to the case where $s$ is large enough that $N_{\text{value}}$ would be considered negligibly small.

Theorem 2. *Problem 1 is NP-Hard for any $w_{\max} \geq 2$ and the cost function*

$$f(A, P, Q) = N_{index}. \tag{9}$$

Proof. The proof is similar to that of Theorem 1, but because $w_{\max}$ is the only factor limiting the size of blocks, we will use different gadgets. Since the rest of the proof is so similar, we only describe the new gadgets.

Let $B_1$ be a $2w_{\max} + 1 \times 2w_{\max} + 1$ matrix whose upper left $w_{\max} + 1 \times w_{\max} + 1$ subregion is nonzero, save for the upper right and lower left entries of the subregion, and zero everywhere else. Let $B_2$ be the same as $B_1$, except $B_2$'s upper left and lower right entries of the dense subregion are zero and the upper right and lower left entries of the upper left subregion are nonzero. For example, if $w_{\max} = 3$, our gadgets are defined as:

$$B_1 = \begin{bmatrix} x & x & x & & \cdots \\ x & x & x & x & \cdots \\ x & x & x & x & \cdots \\ & x & x & x & \cdots \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}, \; B_2 = \begin{bmatrix} & x & x & x & \cdots \\ x & x & x & x & \cdots \\ x & x & x & x & \cdots \\ x & x & x & & \cdots \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

Our cost function asks us to minimize the total number of blocks. Since the dense region is of size $w_{\max} + 1 \times w_{\max} + 1$, each gadget must contribute at least three blocks to this cost function. This can be achieved with one horizontal and one vertical split in the dense region that isolates one of the zeros on the corners. Any other decomposition with a single horizontal and single vertical split produces four blocks. Thus, when vertices $i_1$ and $i_2$ lie on the same side of the cut, this corresponds to one of the following situations (we keep our example value of $w_{\max} = 3$):

$$v_i \in V_1, P = (1, 1, 1, 2) \quad \left[\begin{array}{|ccc|c|} \hline x & x & x & 0 \\ x & x & x & x \\ x & x & x & x \\ \hline 0 & x & x & x \\ \hline \end{array}\right] \qquad \left[\begin{array}{|c|ccc|} \hline x & x & x & 0 \\ x & x & x & x \\ x & x & x & x \\ \cline{1-4} & x & x & x \\ \cline{2-4} \end{array}\right]$$

$$v_{i'} \in V_1, P = (1, 1, 1, 2) \quad \left[\begin{array}{|ccc|c|} \hline 0 & x & x & x \\ x & x & x & x \\ x & x & x & x \\ \hline x & x & x & \\ \cline{1-3} \end{array}\right] \quad \text{or} \quad \left[\begin{array}{|c|ccc|} \hline 0 & x & x & x \\ x & x & x & x \\ x & x & x & x \\ \hline x & x & x & 0 \\ \hline \end{array}\right]$$

$$e_j, Q = (1, 1, 1, 2) \qquad\qquad e_j, Q = (1, 2, 2, 2)$$

When vertices $i_1$ and $i_2$ like on different sides of the cut, we have:

$$v_i \in V_1, P = (1, 1, 1, 2) \quad \left[\begin{array}{|ccc|c|} \hline x & x & x & 0 \\ x & x & x & x \\ x & x & x & x \\ \hline 0 & x & x & x \\ \hline \end{array}\right] \qquad \left[\begin{array}{|c|ccc|} \hline x & x & x & 0 \\ x & x & x & x \\ x & x & x & x \\ \cline{1-4} & x & x & x \\ \cline{2-4} \end{array}\right]$$

$$v_{i'} \in V_2, P = (1, 2, 2, 2) \quad \left[\begin{array}{|ccc|c|} \hline 0 & x & x & x \\ \hline x & x & x & x \\ x & x & x & x \\ x & x & x & 0 \\ \hline \end{array}\right] \quad \text{or} \quad \left[\begin{array}{c|ccc|} \cline{2-4} & x & x & x \\ \hline \multicolumn{1}{|c|}{x} & x & x & x \\ \multicolumn{1}{|c|}{x} & x & x & x \\ \multicolumn{1}{|c|}{x} & x & x & 0 \\ \hline \end{array}\right]$$

$$e_j, Q = (1, 1, 1, 2) \qquad\qquad e_j, Q = (1, 2, 2, 2)$$

By choosing the correct column partition, the case where vertices $i_1$ and $i_2$ lie on different sides of the cut can be made to use only 6 blocks. When these vertices are on the same side of the cut we require 7 blocks. □

## A.1 Exhaustively Checking (22)

*A.1.1* $P = [1, 2, 2, \ldots], Q = [1, 2, 2, \ldots]$.

$$\begin{bmatrix} \boxed{x} & & & \cdots & \boxed{x} & \cdots & \boxed{x} & \cdots & & \cdots \\ & \boxed{\begin{matrix} x & 0 \\ 0 & x \end{matrix}} & & \begin{matrix} \cdots \\ \cdots \end{matrix} & \boxed{\begin{matrix} x \\ x \end{matrix}} & \begin{matrix} \cdots \\ \cdots \end{matrix} & & \begin{matrix} \cdots \\ \cdots \end{matrix} & \boxed{\begin{matrix} 0 \\ x \end{matrix}} & \begin{matrix} \cdots \\ \cdots \end{matrix} \\ \vdots & \vdots \quad \vdots & & & & & & & & \\ \boxed{x} & \boxed{x \quad x} & & & & & & & & \\ \vdots & \vdots \quad \vdots & & & & & & & & \\ \boxed{x} & & & & & & & & & \\ \vdots & \vdots \quad \vdots & & & & & & & & \\ & \boxed{0 \quad x} & & & & & & & & \\ \vdots & \vdots \quad \vdots & & & & & & & & \end{bmatrix}$$

$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (5 + 6r_2 + 6r_3) + (2 + 4r_2 + 4r_3)s \\ &= 197 + 6\lceil 28s - 10 \rceil + (130 + 4\lceil 28s - 10 \rceil)s \\ &\leq 143 + 262s + 112s^2 \\ &\leq 146 + 263s + 112s^2 \end{aligned}$$

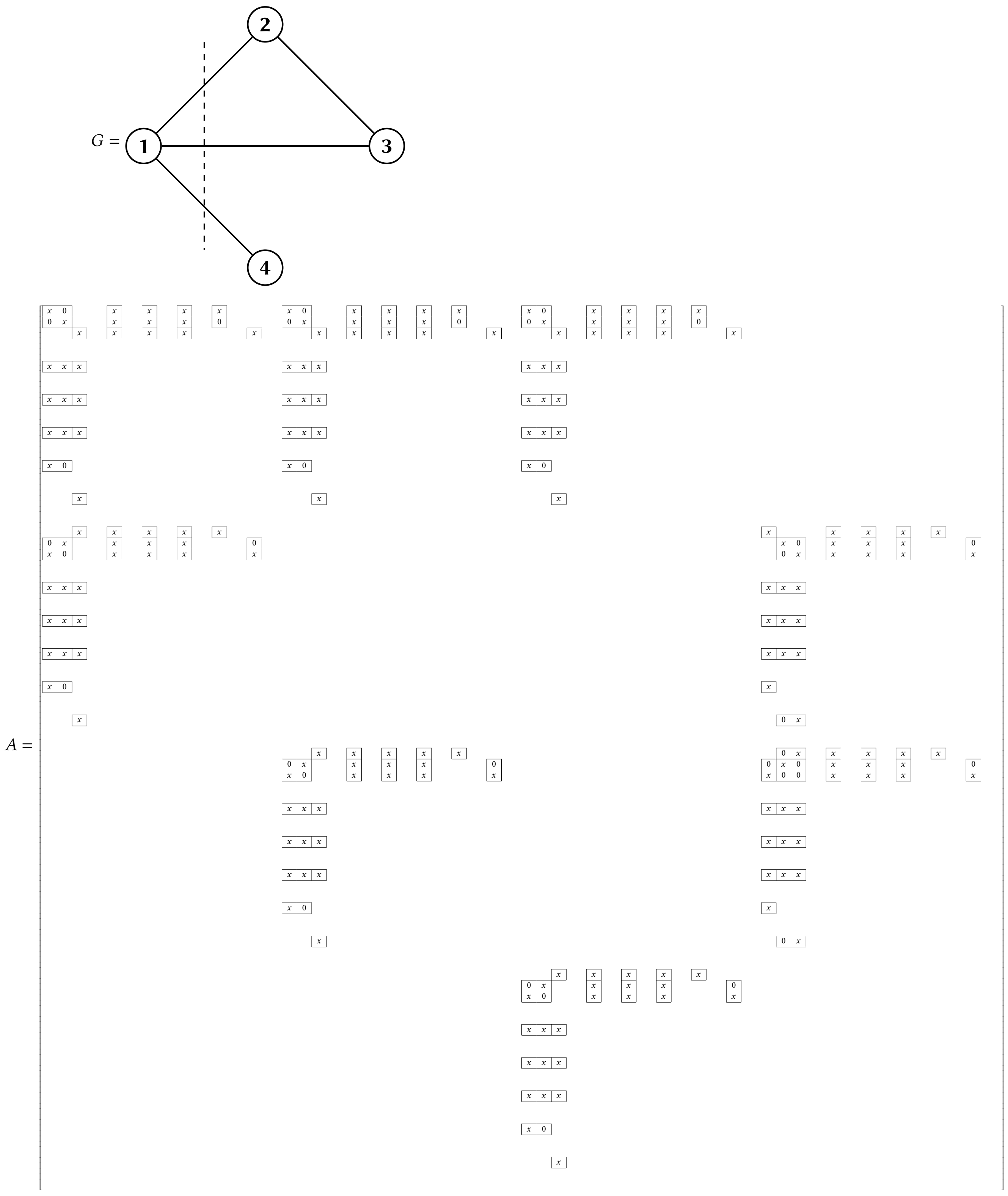


**Figure 3: An example of our reduction for a simple graph $G$. The maximum cut in $G$ is shown, and the corresponding optimal partition of $A$ under the cost function where $s = 1$ is shown. Notice that gadgets corresponding to edges that cross the cut cost less than the gadgets of edges that do not cross the cut.**

*A.1.2* $P = [1, 2, 2, \dots], Q = [1, 1, 2, \dots]$.

$$\begin{bmatrix} x & 0 & & \cdots & x & \cdots & \boxed{x} & \cdots & & \cdots \\ 0 & x & 0 & \cdots & x & \cdots & & \cdots & 0 & \cdots \\ 0 & 0 & x & \cdots & x & \cdots & & \cdots & x & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ x & 0 & & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ & & \boxed{x} & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (8 + 6r_2 + 6r_3) + (3 + 4r_2 + 4r_3)s \\ &= 200 + 6\lceil 28s - 10 \rceil + (131 + 4\lceil 28s - 10 \rceil)s \\ &\le 146 + 263s + 112s^2 \end{aligned}$$

*A.1.3* $P = [1, 1, 2, \dots], Q = [1, 2, 2, \dots]$.

$$\begin{bmatrix} x & 0 & 0 & \cdots & x & \cdots & x & \cdots & & \cdots \\ 0 & x & 0 & \cdots & x & \cdots & 0 & \cdots & & \cdots \\ & 0 & x & \cdots & \boxed{x} & \cdots & & \cdots & \boxed{x} & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ \boxed{x} & & & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ & 0 & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (8 + 6r_2 + 6r_3) + (3 + 4r_2 + 4r_3)s \\ &= 200 + 6\lceil 28s - 10 \rceil + (131 + 4\lceil 28s - 10 \rceil)s \\ &\le 146 + 263s + 112s^2 \end{aligned}$$

*A.1.4* $P = [1, 1, 2, \dots], Q = [1, 1, 2, \dots]$.

$$\begin{bmatrix} x & 0 & & \cdots & x & \cdots & x & \cdots & & \cdots \\ 0 & x & & \cdots & x & \cdots & 0 & \cdots & & \cdots \\ & & \boxed{x} & \cdots & \boxed{x} & \cdots & & \cdots & \boxed{x} & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & \boxed{x} & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ x & 0 & & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ & & \boxed{x} & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (5 + 6r_2 + 6r_3) + (2 + 4r_2 + 4r_3)s \\ &= 197 + 6\lceil 28s - 10 \rceil + (130 + 4\lceil 28s - 10 \rceil)s \\ &\le 143 + 262s + 112s^2 \\ &\le 146 + 263s + 112s^2 \end{aligned}$$

## A.2 Exhaustively Checking (23)

*A.2.1* $P = [1, 1, 1, \dots], Q = [1, 1, 1, \dots]$.

$$\begin{bmatrix} x & 0 & 0 & \cdots & x & \cdots & x & \cdots & 0 & \cdots \\ 0 & x & 0 & \cdots & x & \cdots & 0 & \cdots & 0 & \cdots \\ 0 & 0 & x & \cdots & x & \cdots & 0 & \cdots & x & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ x & 0 & 0 & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ 0 & 0 & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

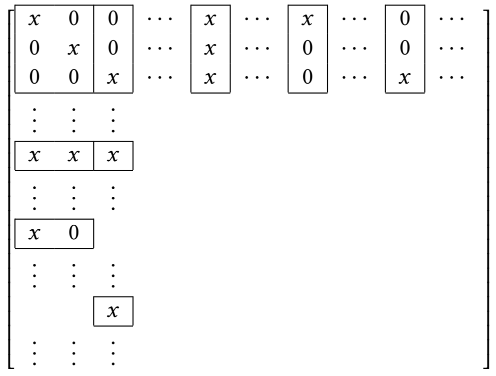
$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (9 + 6r_2 + 12r_3) + (1 + 2r_2 + 4r_3)s \\ &= 201 + 12\lceil 28s - 10 \rceil + (65 + 4\lceil 28s - 10 \rceil)s \\ &\ge 81 + 361s + 112s^2 \\ &\ge 147 + 263s + 112s^2 \end{aligned}$$

*A.2.2* $P = [1, 1, 1, \dots], Q = [1, 2, 2, \dots]$.

$$\begin{bmatrix} x & 0 & 0 & \cdots & x & \cdots & x & \cdots & 0 & \cdots \\ 0 & x & 0 & \cdots & x & \cdots & 0 & \cdots & 0 & \cdots \\ 0 & 0 & x & \cdots & x & \cdots & 0 & \cdots & x & \cdots \\ \vdots & \vdots & \vdots & & & & & & & \\ x & x & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ \boxed{x} & & & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \\ & 0 & x & & & & & & & \\ \vdots & \vdots & \vdots & & & & & & & \end{bmatrix}$$

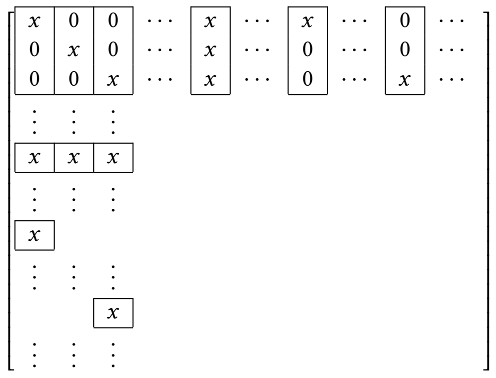
$$\begin{aligned} f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\ &= (9 + 6r_2 + 9r_3) + (2 + 3r_2 + 4r_3)s \\ &= 201 + 9\lceil 28s - 10 \rceil + (98 + 4\lceil 28s - 10 \rceil)s \\ &\ge 111 + 310s + 112s^2 \\ &\ge 147 + 263s + 112s^2 \end{aligned}$$

*A.2.3* $P = [1,1,1,\dots], Q = [1,1,2,\dots]$.



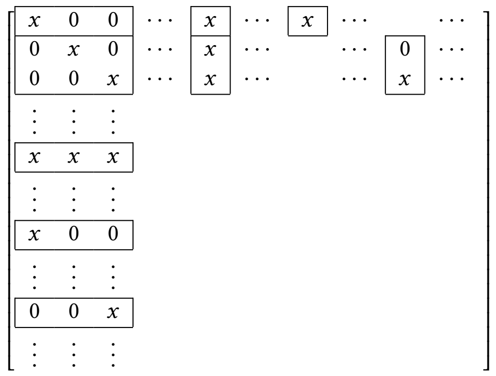
$$\begin{aligned} f(B_1,P,Q) &= N_{\text{value}} + N_{\text{index}}s \\ &= (9 + 6r_2 + 9r_3) + (2 + 3r_2 + 4r_3)s \\ &= 201 + 9\lceil 28s - 10\rceil + (98 + 4\lceil 28s - 10\rceil)s \\ &\geq 111 + 310s + 112s^2 \\ &\geq 147 + 263s + 112s^2 \end{aligned}$$

*A.2.4* $P = [1,1,1,\dots], Q = [1,2,3,\dots]$.



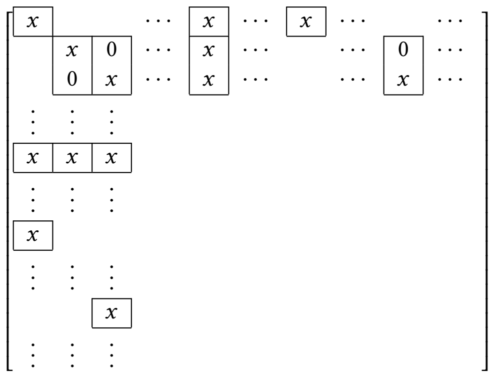
$$\begin{aligned} f(B_1,P,Q) &= N_{\text{value}} + N_{\text{index}}s \\ &= (9 + 6r_2 + 8r_3) + (3 + 4r_2 + 4r_3)s \\ &= 201 + 8\lceil 28s - 10\rceil + (131 + 4\lceil 28s - 10\rceil)s \\ &\geq 121 + 315s + 112s^2 \\ &\geq 147 + 263s + 112s^2 \end{aligned}$$

*A.2.5* $P = [1,2,2,\dots], Q = [1,1,1,\dots]$.



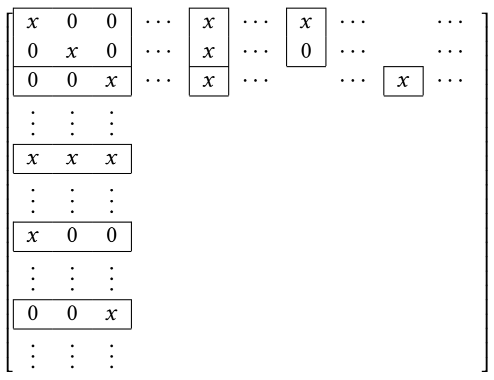
$$\begin{aligned} f(B_1,P,Q) &= N_{\text{value}} + N_{\text{index}}s \\ &= (9 + 6r_2 + 9r_3) + (2 + 3r_2 + 4r_3)s \\ &= 201 + 9\lceil 28s - 10\rceil + (98 + 4\lceil 28s - 10\rceil)s \\ &\geq 111 + 310s + 112s^2 \\ &\geq 147 + 263s + 112s^2 \end{aligned}$$

*A.2.6* $P = [1,2,2,\dots], Q = [1,2,3,\dots]$.



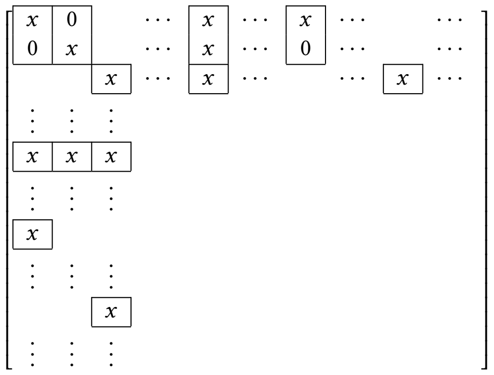
$$\begin{aligned} f(B_1,P,Q) &= N_{\text{value}} + N_{\text{index}}s \\ &= (5 + 6r_2 + 5r_3) + (3 + 5r_2 + 4r_3)s \\ &= 197 + 5\lceil 28s - 10\rceil + (163 + 4\lceil 28s - 10\rceil)s \\ &\geq 147 + 263s + 112s^2 \end{aligned}$$

*A.2.7* $P = [1,1,2,\dots], Q = [1,1,1,\dots]$.



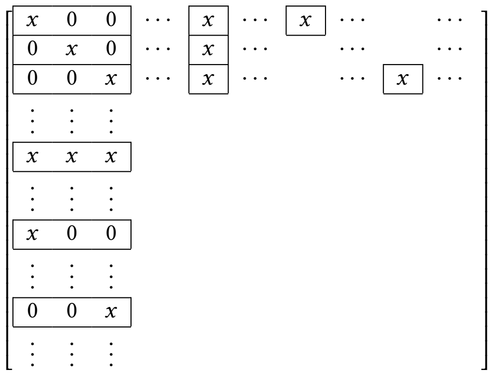
$$\begin{aligned} f(B_1,P,Q) &= N_{\text{value}} + N_{\text{index}}s \\ &= (9 + 6r_2 + 9r_3) + (2 + 3r_2 + 4r_3)s \\ &= 201 + 9\lceil 28s - 10\rceil + (98 + 4\lceil 28s - 10\rceil)s \\ &\geq 111 + 310s + 112s^2 \\ &\geq 147 + 263s + 112s^2 \end{aligned}$$

*A.2.8* $P = [1, 1, 2, \ldots], Q = [1, 2, 3, \ldots]$.



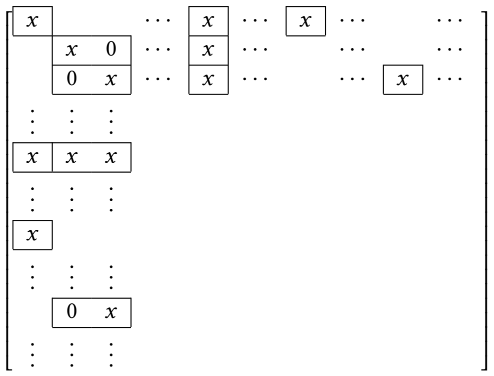
$$
\begin{aligned}
f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\
&= (5 + 6r_2 + 5r_3) + (3 + 5r_2 + 4r_3)s \\
&= 197 + 5\lceil 28s - 10\rceil + (163 + 4\lceil 28s - 10\rceil)s \\
&\geq 147 + 263s + 112s^2
\end{aligned}
$$

*A.2.9* $P = [1, 2, 3, \ldots], Q = [1, 1, 1, \ldots]$.



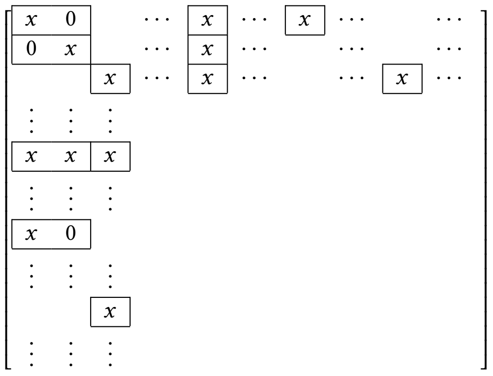
$$
\begin{aligned}
f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\
&= (9 + 6r_2 + 8r_3) + (3 + 4r_2 + 4r_3)s \\
&= 201 + 8\lceil 28s - 10\rceil + (131 + 4\lceil 28s - 10\rceil)s \\
&\geq 121 + 315s + 112s^2 \\
&\geq 147 + 263s + 112s^2
\end{aligned}
$$

*A.2.10* $P = [1, 2, 3, \ldots], Q = [1, 2, 2, \ldots]$.



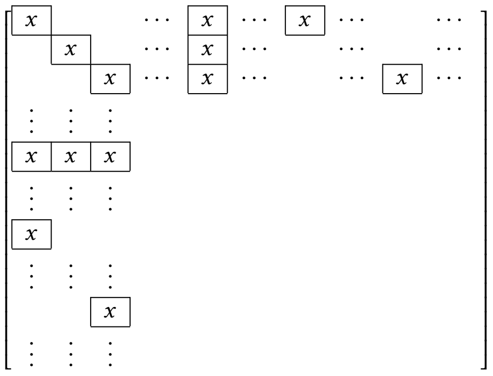
$$
\begin{aligned}
f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\
&= (5 + 6r_2 + 5r_3) + (3 + 5r_2 + 4r_3)s \\
&= 197 + 5\lceil 28s - 10\rceil + (163 + 4\lceil 28s - 10\rceil)s \\
&\geq 147 + 263s + 112s^2
\end{aligned}
$$

*A.2.11* $P = [1, 2, 3, \ldots], Q = [1, 1, 2, \ldots]$.



$$
\begin{aligned}
f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\
&= (5 + 6r_2 + 5r_3) + (3 + 5r_2 + 4r_3)s \\
&= 197 + 5\lceil 28s - 10\rceil + (163 + 4\lceil 28s - 10\rceil)s \\
&\geq 147 + 263s + 112s^2
\end{aligned}
$$

*A.2.12* $P = [1, 2, 3, \ldots], Q = [1, 2, 3, \ldots]$.



$$
\begin{aligned}
f(B_1, P, Q) &= N_{\text{value}} + N_{\text{index}} s \\
&= (9 + 6r_2 + 4r_3) + (3 + 6r_2 + 4r_3)s \\
&= 201 + 4\lceil 28s - 10\rceil + (195 + 4\lceil 28s - 10\rceil)s \\
&\geq 161 + 267s + 112s^2 \\
&\geq 147 + 263s + 112s^2
\end{aligned}
$$