A Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats

Peter Ahrens, Helen Xu, and Nicholas Schiefer
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, Massachusetts USA
pahrens@mit.edu, hjxu@mit.edu, and schiefer@mit.edu

***Abstract*—Many sparse matrices and tensors from a variety of applications, such as finite element methods and computational chemistry, have a natural aligned rectangular nonzero block structure. Researchers have designed high-performance blocked sparse operations which can take advantage of this sparsity structure to reduce the complexity of storing the locations of nonzeros. The performance of a blocked sparse operation depends on how well the block size reflects the structure of nonzeros in the tensor. Sparse tensor structure is generally unknown until runtime, so block size selection must be efficient. The *fill* is a quantity which, for some block size, relates the number of nonzero blocks to the number of nonzeros. Many performance models use the fill to help choose a block size. However, the fill is expensive to compute exactly.**

We present a sampling-based algorithm called PHIL to estimate the fill of sparse matrices and tensors in any format. We provide theoretical guarantees for sparse matrices and tensors, and experimental results for matrices. The existing state-of-the-art fill estimation algorithm, which we will call OSKI, runs in time linear in the number of elements in the tensor. The number of samples PHIL needs to compute a fill estimate is unrelated to the number of nonzeros and depends only on the order (number of dimensions) of the tensor, desired accuracy of the estimate, desired probability of achieving this accuracy, and number of considered block sizes. We compare PHIL and OSKI on a suite of 42 matrices. On most inputs, PHIL estimates the fill at least 2 times faster and often more than 20 times faster than OSKI. PHIL consistently produced accurate estimates — in all cases that we tested PHIL was faster and/or more accurate than OSKI. Finally, we find that PHIL and OSKI produce comparable speedups in multicore blocked sparse matrix-vector multiplication (SpMV) when the block size was chosen using fill estimates in a model due to Vuduc et al.

***Keywords*-Fill Estimation; Sparse Matrix; Sparse Tensor; Block Sparse; Block Size Selection; Randomized Algorithm; Sampling Algorithm; Autotuning; Performance Engineering; Performance Model; Numerical Linear Algebra; Phil**

I. INTRODUCTION

Matrices and tensors (multidimensional generalizations of matrices) are considered sparse when they contain far more zero entries than nonzero entries. Sparse matrices and tensors allow performance engineers to write algorithms and data structures with complexity proportional to the number of nonzero entries, leading to substantial improvements in performance over dense implementations.

Sparse matrices and tensors have applications across a diverse range of domains [1], [2]. For example, sparse tensors have applications in review systems [3], quantum chemistry [4], and natural language processing [5]. Sparse matrix-vector multiplication is one of the most heavily used numerical kernels in scientific computing. Parallel implementations of this numerical kernel are usually limited by memory bandwidth [6], [7].

Sparse storage formats provide benefits over dense storage by only storing and operating upon the nonzeros. The increased complexity of data structures that can describe the irregular locations of nonzeros in these formats, however, poses a significant challenge to algorithm designers and performance engineers. Several storage formats for matrices and tensors reduce this complexity by taking advantage of structural patterns in the locations of nonzeros [2], [7]–[10].

We focus on regular **blocked** formats, which store aligned rectangular dense blocks of nonzeros instead of storing the nonzeros individually. Blocked formats reduce memory traffic and improve the efficiency of parallel sparse operations. Computations over dense blocks also admit more performance optimizations than computations over individual nonzeros [9]. Several sparse matrices and tensors in scientific computing lend themselves naturally to blocked structures. For example, sparse matrices arising from finite element methods [11] and sparse tensors arising in quantum chemistry [12] both exhibit regular block structure.

The performance of a blocked sparse operation depends on how the architecture responds to a block size and how well the block size reflects the structure of the sparse tensor. Thus, block size choice is critical to the performance of any blocked storage format. Vuduc et al. show that choosing the correct blocking can speed up sparse matrix-vector multiplication by a more than a factor 2 on matrices with a blocked structure [13]. Since zeros in the dense blocks must be stored explicitly, an ideal blocking scheme would perform well on the given architecture while minimizing the “filling in”, or explicit representation, of zeros. Im et. al. proposed a performance model of blocked sparse matrix multiplication which depends on a quantity called the **fill**, or the ratio of introduced zeros to the original number of nonzeros [14]. Many subsequent performance models for matrices have been formulated in terms of the fill or directly related quantities [6], [11], [13]–[20]. In the absence of an efficient fill estimation algorithm, block size selection for sparse tensors has been limited to empirical search [21].

The structure of the sparse tensor is generally not known before runtime. Thus, block size selection must occur at runtime and therefore be efficient. Computing the fill dominates the cost of block size selection and is too costly to compute exactly for all potential block sizes, taking more than hundreds of times the cost of a sparse matrix-vector multiplication. Previously, Vuduc et. al. described an algorithm, which we call OSKI, for estimating the fill of a sparse tensor [11]. OSKI estimates the fill by computing the exact fill on a random selection of rows and then averaging. However, the fill may vary substantially between rows, leaving OSKI vulnerable to several cases of pathological inputs. No theoretical analysis of OSKI has been given, and we show several real-world example matrices on which OSKI consistently produces erroneous results.

A. Contributions

We describe PHIL, the first fill estimation algorithm with provable guarantees for sparse matrices and tensors. At a high level, PHIL repeatedly samples a nonzero entry in the tensor, finds neighboring nonzeros, then computes the number of nonzero elements each block containing that entry for all relevant block sizes.

OSKI runs in time linear in the number of nonzeros and is described only for matrices in CSR format. We provide an exact bound on the number of samples that *does not depend* on the number of nonzeros in the tensor. As long as the tensor storage format allows fast (sublinear in the size of the input) access to elements of the tensor, PHIL runs in time sublinear in the number of nonzeros. However, PHIL does not require a specific tensor storage format.

Given a tensor of order R (a tensor with R dimensions) and a maximum block size B, PHIL only needs $B^{2R}\ln(2B^R/\delta)/(2\epsilon^2)$ samples to compute a result to within ϵ relative error with probability at least $1-\delta$. In addition to the time taken to find the neighboring nonzeros, each sample (for all B^R block sizes) can be processed with $(R+1)(2B)^R$ integer additions and B^R floating point divisions and additions. We later explain how PHIL can be extended to consider arbitrarily large block sizes by limiting attention to multiples of some base block size.

We experimentally evaluate PHIL and OSKI on a suite of sparse matrices. We demonstrate that in almost all cases PHIL provides more accurate estimates than OSKI in half the time, often outperforming OSKI by more than a factor of 20. PHIL consistently provided accurate results even when OSKI produced results with a complete loss of accuracy. In all cases we tested, PHIL was faster and/or more accurate than OSKI. We used the Tensor Algebra Compiler (TACO) to generate parallel blocked sparse matrix vector multiplication kernels [22]. PHIL and OSKI produced fill estimates that resulted in almost identical sparse matrix-vector multiplication times when the performance model proposed by Vuduc et al. was used to select a block size [13].

II. BACKGROUND

In this section we introduce tensor notation, various sparse tensor representations, and blocking schemes. We conclude the section by describing the *fill estimation problem* and related previous work.

A. Tensors

Throughout this paper, we discuss order-R tensors in a particular orthogonal basis. That is, tensors are R-dimensional arrays of elements over some field $\mathbb{F}$, usually the real or complex numbers. We denote tensors by capital script letters $\mathcal{A}$ and vectors by lowercase boldface letters $\mathbf{a}$.

The element of a order-R tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$ addressed by a coordinate made up of R indices $(i_1, i_2, \ldots, i_R)$ where $1 \leq i_r \leq I_r$ is denoted $\mathcal{A}[i_1, i_2, \ldots, i_R]$. For compactness of notation, we sometimes specify a coordinate as an R-component vector $\mathbf{i} = (i_1, i_2, \ldots, i_R)$. We represent the range of indices $i, i+1, \ldots, i'$ with the syntax $i : i'$. We represent a range of coordinates with the syntax $\mathbf{i} : \mathbf{i}'$, meaning $(i_1 : i'_1) \times \cdots \times (i_R : i'_R)$. Subtensors are formed when we fix a subset of coordinates. We also use : without bounds to indicate all elements along a particular dimension. Thus, the middle $n/2$ columns of a matrix $\mathcal{A} \in \mathbb{F}^{n \times n}$ would be written $\mathcal{A}[:, n/4 : 3n/4]$.

We denote the number of nonzero entries in a tensor $\mathcal{A}$ as $k(\mathcal{A})$. When we compare a vector to a scalar, our comparison is true if and only if the comparison is true for each entry of the vector pointwise. For convenience, we occasionally redefine the starting coordinate of a tensor. Thus, $\mathcal{A} \in \mathbb{F}^{\mathbf{I}:\mathbf{I}'}$ is an $(I'_1 - I_1 + 1) \times \cdots \times (I'_R - I_R + 1)$ tensor whose smallest coordinate is $\mathbf{I}$ and largest coordinate is $\mathbf{I}'$.

B. Sparse Tensor Representations

Most sparse formats store only the coordinates which correspond to nonzeros and the nonzero values themselves. While we discuss a few specific formats, note that our algorithm applies to any sparse tensor format which admits iteration over nonzero coordinates.

The simplest sparse matrix and tensor format is *Coordinate (COO)* [2]. In this format, all coordinates which correspond to nonzeros are stored in an unordered list. Entries are stored in sorted order of their coordinates.

Perhaps the most popular sparse matrix format is the *Compressed Sparse Row (CSR)* [8] sparse matrix format. In CSR format, the indices of nonzeros in each row are stored in sorted order. Each row has an associated list of coordinates of nonzeros. The nonzeros are stored in a single array with the same ordering as their coordinates. CSR can be extended to a tensor format in many ways [2], such as *Compressed Sparse Fiber (CSF)* [22], [23]. In CSF format, each coordinate i is stored in a tree structure where a node in level r represents an index i_r which corresponds to a set of nonzeros. CSR is the matrix case of CSF.

To decrease the complexity of storing the coordinates of individual nonzeros, performance engineers may store blocks of nearby nonzeros together. Blocked formats can reduce the memory usage of sparse operations by reducing the complexity of locating nonzeros. Programmers and compilers can optimize linear algebra on small dense blocks using standard techniques such as loop unrolling, register and cache blocking, and instruction-level parallelism. The effectiveness of these optimizations depends heavily on the structure of the tensor and the blocked storage format [9], [24].

Proposed blocked storage formats are diverse, altering parameters such as the size and alignment of blocks, or the storage format for locations of blocks and nonzeros within blocks [9]. Some formats involve reordering to improve the block structure of the tensor (in this case, blocks may not represent contiguous entries in the original tensor) [8], [10].

In this paper, we focus on regular blocking for simplicity, where the aligned rectangular blocks are of equal size and represent contiguous entries in the original tensor. For our experiments, we will use a simple variant of CSR called *Blocked Compressed Sparse Row (BCSR)* [8], where the locations of the nonzero blocks are recorded using CSR format. The BCSR format can be extended naturally to *(BCSF)* format to support higher-dimensional tensors as well [21], [22]. In BCSR and BCSF, each block is stored in a dense format, with zeros represented explicitly, and only blocks which contain nonzeros are stored.

C. Regular Blocking

Definition II.1. A *blocking scheme* $\mathbf{b}$ of a tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$ is parameterized by a vector $\mathbf{b} = (b_1, b_2, \ldots, b_R)$ of block sizes. The blocking scheme induced by $\mathbf{b}$ is a partition of $\mathcal{A}$ into R-dimensional subtensors with b_r entries along the r^{th} dimension. Thus, a nonzero at coordinate $\mathbf{i}$ would be stored at the block coordinate

$$\left(\left\lceil \frac{i_1}{b_1} \right\rceil, \left\lceil \frac{i_2}{b_2} \right\rceil, \ldots, \left\lceil \frac{i_R}{b_R} \right\rceil\right).$$

We present an example of a blocking scheme in a sparse matrix in Figure 1. Blocked formats like BCSR may fill in the empty slots of nonempty blocks with explicit zeros.

D. Fill Estimation

Since the performance of blocked sparse tensor operations depends on the block size and the structure of the tensor, our goal is to choose the block size that gives the best performance for our given tensor. In blocked sparse formats that store dense blocks, larger blocks generally allow more opportunities for performance optimization. However, if the blocks do not capture the structure of the tensor, we will waste time computing with explicitly represented zeros.

We want to find a blocking scheme that includes all of the nonzero entries of $\mathcal{A}$ in very few blocks. Thus, we are interested in the number of blocks containing a nonzero under the blocking scheme $\mathbf{b}$, which we denote $k_{\mathbf{b}}(\mathcal{A})$. Notice that $k_{\mathbf{1}}(\mathcal{A}) = k(\mathcal{A})$, since tiling $\mathcal{A}$ into unit-size blocks will have exactly one non-empty block for every nonzero. The *fill* is a metric which uses the number of nonzero blocks to formally express this notion of blocking scheme quality:

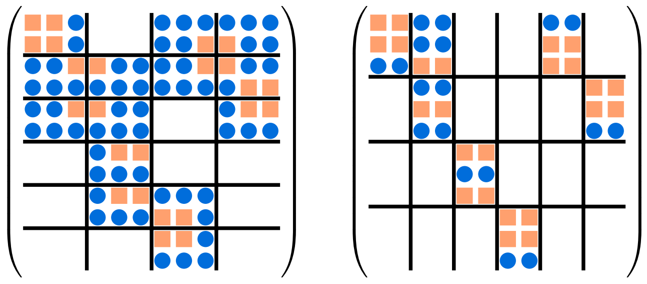


Figure 1. On the left, a sparse matrix before blocking. On the right, the same sparse matrix after blocking. The squares denote nonzero elements and circles are explicit zeros that are introduced due to the storage format. In this example, the blocking scheme $\mathbf{b} = (2, 3)$ and $k_{\mathbf{b}}(\mathcal{A}) = 12$. The number of nonzero elements $k(\mathcal{A}) = 30$, so the *fill* $f_{\mathbf{b}}(\mathcal{A}) = (2 \times 3 \times 12)/30 = 2.4$.

Definition II.2 ([14]). The *fill* of a tensor $\mathcal{A}$ with respect to a particular blocking scheme $\mathbf{b}$ is the ratio

$$f_{\mathbf{b}}(\mathcal{A}) = \frac{b_1 b_2 \cdots b_R k_{\mathbf{b}}(\mathcal{A})}{k(\mathcal{A})}.$$

That is, the fill is the ratio of the number of entries in nonempty blocks in the blocking scheme $\mathbf{b}$ of $\mathcal{A}$ to the number of nonzeros in $\mathcal{A}$. Where it is clear which tensor we refer to, we often write the fill as $f_{\mathbf{b}}$. For a fixed number of nonzeros, the fill $f_{\mathbf{b}}(\mathcal{A})$ is directly proportional to the number of nonzero blocks $k_{\mathbf{b}}(\mathcal{A})$.

The fill was first defined by Im et. al., and later used in several BCSR matrix-vector multiply performance prediction models [11], [13]–[18]. The fill has also been used to select block sizes for sparse triangular solve and sparse $\mathcal{A}^T\mathcal{A}\mathbf{x}$ [11]. The number of nonzero blocks (proportional to the fill) has been used in performance models for general blocked format sparse matrix-vector multiply [6], [19], [20]. Block size selection remains a difficult problem for tensors as it is difficult to estimate the fill, so developers have adopted empirical search techniques [21]. An estimate of the fill could easily be added as an additional feature in feature-based machine learning approaches to sparse kernel performance modeling [25].

As an example, we explain the simple performance model for blocked sparse matrix-vector multiply given in [13]. There are more accurate performance models which still depend on the fill, but our focus is on fill estimation and not performance modeling. It was later shown that, when the fill was known exactly, performance of the resulting blocking scheme was optimal or near-optimal (within 5%) [11].

Once per machine, we compute a profile of how the machine performs for each block size. Let $P_{\mathbf{b}}$ be the performance of the machine (in flop/s) on a dense matrix stored with blocking scheme $\mathbf{b}$. $P_{\mathbf{b}}$ is a measure of how efficiently we can process nonzeros when nonzeros are stored in blocks of size $\mathbf{b}$. We can estimate the performance of the machine on the BCSR format of $\mathcal{A}$ as $P_{\mathbf{b}}/f_{\mathbf{b}}(\mathcal{A})$, then choose a block size which maximizes the estimated performance.

For dense blocks in matrices, we care only about block sizes $b_1 \times b_2$ that are small enough to fit b_1 input, b_2 output, and at least one matrix element in registers. This usually limits our attention to $b_1, b_2 \leq 12$ [11]. Thus, our problem is to quickly compute an estimate of the fill for these block sizes with reasonable accuracy.

Problem II.1 (Fill Estimation)**.** Given a tensor $\mathcal{A}$ and a maximum block size B, compute a (randomized) approximation $F_{\mathbf{b}}(\mathcal{A})$ with error at most $\epsilon > 0$ such that

$$(1-\epsilon)f_{\mathbf{b}}(\mathcal{A}) \leq F_{\mathbf{b}}(\mathcal{A}) \leq (1+\epsilon)f_{\mathbf{b}}(\mathcal{A})$$

for all (square or rectangular) block sizes $\mathbf{b} \leq B$, with probability at least $1-\delta$ where $0 < \delta < 1$. Equivalently, we want to compute a random variable $F_{\mathbf{b}}(\mathcal{A})$ such that

$$\Pr\left[\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} > \epsilon\right] \leq \delta\,.$$

Since $f_{\mathbf{b}}(\mathcal{A})$ differs from $k_{\mathbf{b}}(\mathcal{A})$ by a multiplicative factor of $b_1 b_2 \cdots b_R/k(\mathcal{A})$ (which can easily be computed in constant time), estimating the fill is equivalent to estimating the number of nonzero blocks.

E. Previous Work

Exact computation of the fill for many block sizes is computationally intractable in comparison to the cost of a sparse matrix-vector multiplication. There has been a recent attempt to parallelize the computation on matrices [26]. However, it was only able to provide competitive results by drastically reducing the number of quantities estimated.

To our knowledge, only one previously proposed algorithm estimates the fill instead of computing it exactly [11], [15]. Since the algorithm is implemented in the Optimized Sparse Kernel Interface (OSKI) library, we will refer to it as OSKI [17]. For each block row size $1 \leq b_1 \leq B$, OSKI samples a fraction of block rows. For each sampled block row, OSKI computes the fill exactly for all block column sizes $1 \leq b_2 \leq B$ simultaneously. OSKI does this by iterating through coordinates $\mathbf{i}$ of nonzeros in the block row and using a perfect hash table for each block column size to record the number of unique block column coordinates $\lceil i_2/b_2 \rceil$ seen. The fraction of block rows evaluated is specified by a parameter σ which is usually set to 0.02.

Although OSKI can estimate the fill of most matrices, it does not give predictable results. In our work, we show that it is vulnerable to special cases. To our knowledge, there are no theoretical guarantees on the accuracy of OSKI, and no existing algorithm which estimates the fill of tensors.

III. The Algorithm

We begin with a high-level summary of PHIL, our sampling-based fill estimation algorithm. Suppose we want to estimate the fill of a sparse tensor $\mathcal{A}$ given a maximum block size B. PHIL repeatedly samples a coordinate $\mathbf{i}$ of a nonzero with replacement from $\mathcal{A}$. For each blocking scheme $\mathbf{b} \leq B$, it computes the number $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ of nonzero entries in *the block that* $\mathbf{i}$ *appears in* under the blocking scheme $\mathbf{b}$, which it uses to estimate the fill.

PHIL computes $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ efficiently by using prefix sums to minimize redundant work. Once we find the coordinates of all nonzeros near $\mathbf{i}$, we use multidimensional prefix sums (cumulative sums) to compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all blocking schemes $\mathbf{b} \leq B$ in less than $(R+1)(2B)^R$ integer additions. Note that B and R are both expected to be small, and we are computing B^R separate quantities.

We define $F_{\mathbf{b}}$, a quantity proportional to the average of the reciprocals $1/z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$, and show that $F_{\mathbf{b}}$ is an *unbiased estimator* for the fill $f_{\mathbf{b}}$ (a random variable with expectation equal to the fill). In Theorem III.1 we give a concentration bound for $F_{\mathbf{b}}$, showing that PHIL solves the fill approximation problem as long as we use enough samples. We include a proof and discussion of Theorem III.1 in Section IV.

Theorem III.1. *If we sample at least*

$$S \geq S_0 = \frac{B^{2R}}{2\epsilon^2} \ln\left(\frac{2B^R}{\delta}\right)$$

samples with replacement, then

$$\Pr\left[\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \leq \epsilon\right] \geq 1-\delta\,.$$

The required number of samples S_0 is independent of the number of nonzeros $k(\mathcal{A})$. S_0 depends only on the desired accuracy and desired probability of attaining such accuracy. The required number of samples is constant with respect to the problem size. This is a clear advantage for large tensors where performance engineering matters the most.

A. Fill Estimation

We describe how PHIL computes an unbiased estimator for the fill. First, we introduce a few important definitions for working with blocking schemes on tensors:

Definition III.1. The *head* of a block is the unique coordinate in the block with the lowest index along all dimensions. For any coordinate $\mathbf{i}$, let $h_{\mathbf{b}}(\mathbf{i})$ denote the head of $\mathbf{i}$'s block under the blocking scheme $\mathbf{b}$. Similarly, the *tail* of a block is the unique coordinate in the block with the highest index along all dimensions. For any coordinate $\mathbf{i}$, let $t_{\mathbf{b}}(\mathbf{i})$ denote the tail of $\mathbf{i}$'s block under $\mathbf{b}$.

Let $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ be defined on each coordinate $\mathbf{i}$ of a nonzero of $\mathcal{A}$ as:

$$x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = \frac{1}{z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})} = \frac{1}{k(\mathcal{A}[h_{\mathbf{b}}(\mathbf{i}) : t_{\mathbf{b}}(\mathbf{i})])}\,,$$

where $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is the number of nonzeros in the block of $\mathbf{i}$ under blocking scheme $\mathbf{b}$. Thus, $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is the reciprocal of the number of nonzeros in $\mathbf{i}$'s block.

PHIL averages $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over S coordinates $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$ sampled with replacement from the set of coordinates of nonzeros in $\mathcal{A}$. The average of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all $\mathbf{i}$ is closely related to the fill, so we compute the fill estimate $F_{\mathbf{b}}$ as:

Definition III.2. For all $\mathbf{b} \leq B$:

$$F_{\mathbf{b}} := \frac{b_1 b_2 \cdots b_R}{S} \sum_{j=1}^{S} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$$

Theorem III.2. *For any blocking scheme* $\mathbf{b}$*, the random variable* $F_{\mathbf{b}}$ *is an unbiased estimator for the fill: that is,* $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}(\mathcal{A})$.

Proof: Notice that the sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all of the nonzeros $\mathbf{i}$ within a particular block is 1 if the block is not empty. Thus, the sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all nonzeros $\mathbf{i}$ in $\mathcal{A}$ is equal to $k_{\mathbf{b}}(\mathcal{A})$, the number of blocks that contain nonzeros. Thus, we may multiply the average of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over $\mathbf{i}$ by $b_1 b_2 \cdots b_R$ to obtain an estimator of $f_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$, by definition. ■

Next, we describe how PHIL computes $F_{\mathbf{b}}$ in Algorithm III.1.

Algorithm III.1. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, $\mathbf{i}$, and B, compute an approximation to $f_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all block sizes $\mathbf{b} \leq B$. Note that $\mathcal{A}$ may be stored in a sparse format, whereas all other tensors are stored in a dense format.

```
Require:
    $0 \leq \delta \leq 1, \quad \epsilon > 0, \quad B \geq 1$
 1: function ESTIMATEFILL($\mathcal{A}$, $B$, $\epsilon$, $\delta$)
 2:     $\mathcal{Y} \in \mathbb{R}^{B \times \cdots \times B}$
 3:     $\mathcal{F} \in \mathbb{R}^{B \times \cdots \times B}$
 4:     $S \leftarrow \left\lceil \frac{B^{2R}}{2\epsilon^2} \ln\left(\frac{2B^R}{\delta}\right) \right\rceil$.
 5:     $\mathcal{Y} \leftarrow 0$
 6:     for $\mathbf{i} \in$ sample of size $S$ with replacement from the
        nonzero coordinates of $\mathcal{A}$ do
 7:         $\mathcal{Y} \leftarrow \mathcal{Y} + \text{COMPUTE}\mathcal{X}(\mathcal{A}, B, \mathbf{i})$
 8:     for $\mathbf{b} \in B \times \cdots \times B$ do
 9:         $\mathcal{F}[\mathbf{b}] \leftarrow \frac{b_1 b_2 \cdots b_R \mathcal{Y}[\mathbf{b}]}{s}$
10:     return $\mathcal{F}$
Ensure:
    $(1-\epsilon) f_{\mathbf{b}}(\mathcal{A}) \leq \mathcal{F}[\mathbf{b}] \leq (1+\epsilon) f_{\mathbf{b}}(\mathcal{A})$ with probability
    at least $(1-\delta)$.
```

B. COMPUTE$\mathcal{X}$

We could compute $x_{\mathbf{b}}(\mathcal{A}, i)$ for a sample coordinate $\mathbf{i}$ by looking up how many nonzeros are in the block corresponding to $\mathbf{i}$ and returning the reciprocal. However, finding the number of nonzeros in a block takes time linear in the number of nonzeros in that block (in addition to the cost of finding these coordinates) and therefore could potentially take time B^R in an order-R tensor.

PHIL *reuses* the computations of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for the same $\mathbf{i}$ over different blocking schemes $\mathbf{b}$. After finding the locations of all the nonzeros within a B radius of a nonzero at coordinate $\mathbf{i}$, we can compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all $\mathbf{b} \leq B$ at the same time. This is described in Algorithm III.2 and illustrated in Figure 2.

Algorithm III.2. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, $\mathbf{i}$, and B, compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all blocking schemes $\mathbf{b} \leq B$. Note that $\mathcal{A}$ may be stored in a sparse format, whereas all other tensors are stored in a dense format.

```
Require:
    $\mathcal{A}[\mathbf{i}] \neq 0, \quad B \geq 1$
 1: function COMPUTE$\mathcal{X}$($\mathcal{A}$, $\mathbf{i}$, $B$)
 2:     $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B:\mathbf{i}+B-1}$
 3:     $\mathcal{Z}_0 \leftarrow 0$
 4:     for $\mathbf{j} \in \text{NONZEROSINRANGE}(\mathcal{A}, \mathbf{i}-B, \mathbf{i}+B-1)$ do
 5:         $\mathcal{Z}_0[\mathbf{j}] \leftarrow 1$
 6:     for $r \in 1:R$ do
 7:         for $j \in i_r - B + 1 : i_r + B - 1$ do
 8:             $\mathcal{Z}_0[\underbrace{:, \ldots, :, j}_{r}, :, \ldots, :] \leftarrow \mathcal{Z}_0[\underbrace{:, \ldots, :, j}_{r}, :, \ldots, :] + \mathcal{Z}_0[\underbrace{:, \ldots, : j-1}_{r}, :, \ldots, :]$
 9:     for $b_1 \in 1:B$ do
10:         $\mathcal{Z}_1 \leftarrow \mathcal{Z}_0[t_{b_1}(i_1), \underbrace{:, \ldots, :}_{r-1}] - \mathcal{Z}_0[h_{b_1}(i_1)-1, \underbrace{:, \ldots, :}_{r-1}]$
11:         for $b_2 \in 1:B$ do
12:             $\mathcal{Z}_2 \leftarrow \mathcal{Z}_1[t_{b_2}(i_2), \underbrace{:, \ldots, :}_{r-2}] - \mathcal{Z}_1[h_{b_2}(i_2)-1, \underbrace{:, \ldots, :}_{r-2}]$
    ⋮
13:                 for $b_R \in 1:B$ do
14:                     $\mathcal{Z}_R \leftarrow \mathcal{Z}_{R-1}[t_{b_R}(i_R)] - \mathcal{Z}_{R-1}[h_{b_R}(i_R)-1]$
15:                     $\mathcal{X}[\mathbf{b}] \leftarrow \frac{1}{\mathcal{Z}_R}$
Ensure:
    $\mathcal{X}[\mathbf{b}] \leftarrow x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$
```

The main idea behind COMPUTE$\mathcal{X}$ is to create a tensor $\mathcal{Z}_0$ corresponding to the number of nonzeros of $\mathcal{A}$ in subtensors surrounding $\mathbf{i}$. We can use the differences in the number of nonzeros in the subtensors to find the number of nonzeros in the desired block.

More formally, we construct some $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B:\mathbf{i}+B-1}$ such that $\mathcal{Z}_0[\mathbf{j}]$ is equal to the number of nonzeros in the subtensor $\mathcal{A}[\mathbf{i} - B : \mathbf{j}]$. In one dimension, we can compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ as $\mathcal{Z}_0[t_{\mathbf{b}}(\mathbf{i})] - \mathcal{Z}_0[h_{\mathbf{b}}(\mathbf{i}) - 1]$. In two dimensions, we can compute $z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ as $\mathcal{Z}_0[t_{\mathbf{b}}(\mathbf{i})] - \mathcal{Z}_0[t_{b_1}(i_1), h_{b_2}(i_2) - 1] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, t_{b_2}(i_2)] + \mathcal{Z}_0[h_{\mathbf{b}}(\mathbf{i}) - 1]$.

The core of COMPUTE$\mathcal{X}$ is the computation of $\mathcal{Z}_0$. We initialize $\mathcal{Z}_0[\mathbf{j}]$ to 1 if $\mathcal{A}[\mathbf{j}] \neq 0$ and 0 otherwise. Then, we take a prefix sum along each dimension in turn. After the first prefix sum, $\mathcal{Z}_0[\mathbf{j}]$ is the number of nonzeros in $\mathcal{A}[i_1 - B : j_1, j_2, \ldots, j_R]$. After the r^{th} prefix sum, $\mathcal{Z}_0[\mathbf{j}]$ is the number of nonzeros in $\mathcal{A}[i_1 - B : j_1, \ldots, i_r - B : j_r, j_{r+1}, \ldots, j_R]$. After the R^{th} prefix sum, we have computed $\mathcal{Z}_0$.

We find the number of nonzeros in each block ($z_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$) using differences between elements of $\mathcal{Z}_0$. For each value of b_1, we set $\mathcal{Z}_1[j_2, \ldots, j_R]$ to the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) : t_{b_1}(i_1), i_2 - B : j_2, \ldots, i_R - B : j_R]$ as $\mathcal{Z}_0[t_{b_1}(i_1), j_2, \ldots, j_R] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, j_2, \ldots, j_R]$.

Having computed $\mathcal{Z}_1$ for a particular value of b_1, then for each value of b_2 we take differences between elements of $\mathcal{Z}_1$ to compute $\mathcal{Z}_2$, where $\mathcal{Z}_2[j_3, \ldots, j_R]$ is the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) : t_{b_1}(i_1), h_{b_2}(i_2) : t_{b_2}(i_2), i_3 - B : j_3, \ldots, i_R - B : j_R]$. Continuing in this way, $\mathcal{Z}_R$ is just the scalar $z_{\mathbf{b}}(\mathcal{A}, \mathbf{j})$.

Each prefix sum takes at most $(2B)^R$ additions to compute, and we compute R prefix sums. In the final loop, $\mathcal{Z}_r$ is of size $(2B)^{R-r}$. We must compute $\mathcal{Z}_r$ exactly B^r times. Therefore, the block difference computation incurs $\sum_{r=1}^{R} 2^{-r}(2B)^R$ subtractions. Thus, ComputeX uses at most $(R+1)(2B)^R$ integer additions to compute $\mathcal{Z}$.

C. NonzerosInRange

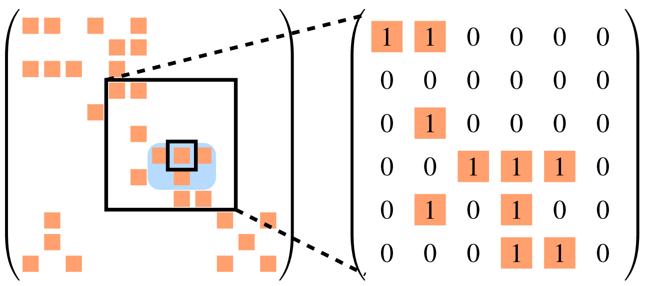
Since $\mathcal{A}$ may be stored in an arbitrary sparse format, we abstract the process of finding the coordinates of nonzeros within a certain range into an algorithm called NonzerosInRange. NonzerosInRange$(\mathcal{A}, \mathbf{j}, \mathbf{j}')$ returns a list of all $\mathbf{i} \in \mathbf{j} : \mathbf{j}'$ such that $\mathcal{A}[\mathbf{i}] \neq 0$.

The implementation of NonzerosInRange depends on the initial format of the sparse matrix $\mathcal{A}$. We discuss two implementations to show why this routine should not be costly in theory or practice.

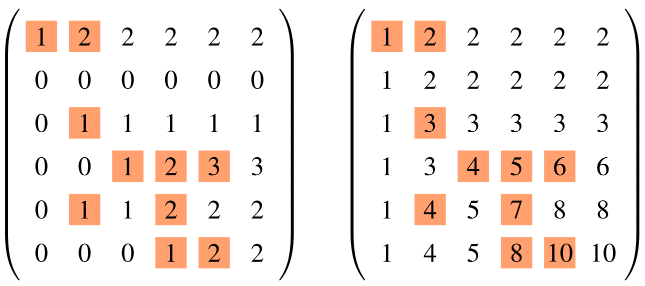
If $\mathcal{A}$ is a matrix in CSR format (where coordinates of nonzeros in each row are stored in sorted order of their column index), then using a binary search within each row provides an $O(B \log_2(I_2) + B^2)$ time implementation, where the B^2 term reflects the maximum number of coordinates that may need to be returned. This search technique generalizes to tensors in CSF format, yielding an $O\left(\sum_{r=2}^{R} B^{r-1} \log_2(I_r) + B^R\right)$ time implementation.

We now describe an implementation of NonzerosInRange for a tensor $\mathcal{A}$ stored in any other format (e.g. COO). Before we run EstimateFill, we block the entire tensor $\mathcal{A}$ into blocks of size $B \times \cdots \times B$ and store the blocks in a sparse format (without explicit zeros). We store each block that contains at least one nonzero in a hash table. NonzerosInRange is only ever called with ranges of size $2B \times \cdots \times 2B$ and only needs to look up the 3^R blocks which might contain zeros in the target range, scan through these blocks to find nonzeros which are actually in the target range, and return these nonzeros. The entire algorithm has a setup time of $O(k(\mathcal{A}))$ and an individual query time of $O(3^R B^R)$.

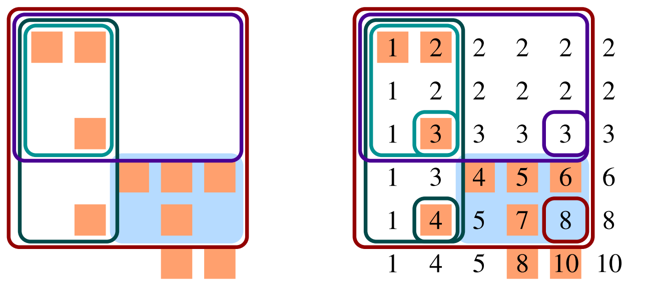
Figure 2. Here we visualize the execution of ComputeX as it computes one element of its output X. Specifically, we show how it computes $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = \mathcal{X}[\mathbf{b}]$. In this example, our maximum block size is $B = 3$ and our nonzero of interest is $\mathbf{i} = (7, 8)$. Continuing our example in Figure 1, we will show computation of $\mathcal{X}$ only for the blocking scheme $\mathbf{b} = (2, 3)$. Our goal is to compute the reciprocal of the number of nonzero elements in $\mathbf{i}$'s block (depicted by the shaded region).



(a) First, ComputeX uses NonzerosInRange to find the nonzeros within a box of size $2B$ around $\mathbf{i}$. Then, it creates a matrix of the same size as the box and fills it with 0 where there are zeros in the original matrix and 1 where there are nonzeros.



(b) Next, ComputeX performs a prefix sum on the rows and then columns of the matrix. Notice that element $\mathbf{j}$ of the matrix is now equal to the number of nonzero elements in the box extending from the upper left of the matrix to element $\mathbf{j}$.



(c) Finally, ComputeX computes the number of elements in the desired block by subtracting the number of nonzeros in each medium sized box from the large box, and adding back in the small box to avoid double-counting. Since all of these boxes begin in the upper left corner of our matrix, the number of nonzeros in these boxes are given by the prefix sum results in their lower right corners. The difference operation tells us that the shaded region contains $8 - 4 - 3 + 3 = 4$ nonzeros. Thus, $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = 1/4$. At this point, it is easy to compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for different $\mathbf{b}$ by repeating the difference operation with different blocks.

IV. Analysis

We want to select the number of samples, S, as small as possible for efficiency while still having provable guarantees on the concentration of our unbiased estimator $\sum_j x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)/S$. We use Hoeffding's inequality [27] as a concentration bound for sampling with replacement.

Theorem IV.1 (Hoeffding's inequality). *Let $X_1, X_2, \ldots, X_M$ be M independent random variables bounded such that $0 \leq X_j \leq 1$. Let $\overline{X} = \frac{1}{M}\sum_{j=1}^{M} X_j$ be their mean. Then for any $t \geq 0$,*

$$\Pr\left[\left|\overline{X} - \mathbb{E}[X]\right| \geq t\right] \leq 2\exp(-2Mt^2)\,.$$

For any blocking scheme $\mathbf{b}$ and any tensor element $\mathbf{i}$, the value $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ is a random variable bounded between 0 and 1. Furthermore, since the entries $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$ are sampled independently from among the nonzeros, the random variables $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_1), x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_S)$ are independent. Therefore, we obtain our concentration bound from Theorem IV.1:

Theorem IV.2 (Restatement of Theorem III.1). *If we sample at least*

$$S \geq S_0 = \frac{B^{2R}}{2\epsilon^2}\ln\left(\frac{2B^R}{\delta}\right)$$

samples with replacement, then

$$\Pr\left[\max_{\mathbf{b}\leq B}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \leq \epsilon\right] \geq 1 - \delta\,.$$

Proof: We have $F_{\mathbf{b}} = b_1 b_2 \cdots b_R (1/S)\sum_{j=1}^{S} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$ by definition. By Theorem III.2, $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}$. Since each examined block contains at least 1 and at most B^R nonzeros, $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_1), x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_S)$ are independent and bounded between $1/B^R$ and 1. Similarly, $k_b(\mathcal{A})/k(\mathcal{A})$ in Definition II.2 is bounded to the same range. By Theorem IV.1,

$$\Pr\left[\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \epsilon\right] = \Pr\left[\left|\frac{F_{\mathbf{b}} - \mathbb{E}[F_{\mathbf{b}}]}{b_1 b_2 \cdots b_R}\right| \geq \epsilon\frac{f_{\mathbf{b}}}{b_1 b_2 \cdots b_R}\right]$$
$$\leq 2\exp\left(-2S\left(\frac{\epsilon k_b(\mathcal{A})}{k(\mathcal{A})}\right)^2\right) \leq 2\exp\left(\frac{-2S\epsilon^2}{B^{2R}}\right),$$

since $F_{\mathbf{b}}$ is $b_1 b_2 \cdots b_R$ times an average of S values, each of which is at least $1/B^R$. By the union bound over the B^R possible blocking schemes $\mathbf{b}$,

$$\Pr\left[\max_{\mathbf{b}\leq B}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \epsilon\right] \leq 2B^R\exp\left(\frac{-2S\epsilon^2}{B^{2R}}\right)\,.$$

Therefore, if $S \geq S_0 = \frac{B^{2R}}{2\epsilon^2}\ln\left(\frac{2B^R}{\delta}\right)$,

$$\Pr\left[\max_{\mathbf{b}\leq \mathbf{B}}\frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \epsilon\right] \leq \delta\,.$$

■

Note that this bound is constant with respect to the number of nonzeros $k(\mathcal{A})$, which is highly advantageous when $S \ll k(\mathcal{A})$. Obtaining a high probability bound with $\delta \leq 1/k(\mathcal{A})^w$ for some w would indeed require dependence on $k(\mathcal{A})$, albeit only logarithmically. However, in practice a small constant δ such as 0.01 likely suffices.

If strong guarantees are desired, such as with matrix ($R = 2$) settings of $B = 12$, $\epsilon = 0.1$ and $\delta = 0.01$, it is possible that the number of required samples (10,645,998) exceeds the number of nonzeros in smaller matrices. This is fundamental to bounds based on sampling with replacement. If we sample without replacement, we can apply a recent result using the Hoeffding-Serfling inequality to obtain a bound which scales with the number of nonzeros [28]. This bound is more complicated to describe, and requires the implementation to generate samples without replacement. Furthermore, this bound would still require sampling a significant fraction of the nonzeros.

Instead, we suggest that implementers who need strong guarantees on small problems use an efficient exact algorithm or lower the maximum block size B (in our example, $B = 4$ needs only 103,308 samples). We show in the next section that PHIL empirically provides far more accurate estimates than the worst-case guaranteed theoretical bound. In practice, for $B = 12$, running PHIL with $\epsilon = 3$ and $\delta = 0.01$ (11,829 samples) results in a mean maximum relative error of at most 0.05 for all cases we tested. PHIL still produces an accurate estimate even when run with relaxed guarantees.

V. Results

We implemented[1] PHIL for sparse matrices in CSR format in C, which can efficiently execute the dense integer and floating point operations in Algorithm III.2. We compare PHIL to the competing algorithm described in [11], which we will refer to as OSKI. We use a test suite inspired by matrices from [11] designed to test fill estimation. We also include some synthetic matrices we generated to test worst-case behavior. Our test suite is summarized in the first few columns of Table 3. We find that PHIL computes the fill more accurately in much less time than OSKI for a wide range of matrices in our test suite. We also find that when using optimized BCSR matrix-vector multiplication routines generated by the Tensor Algebra Compiler (TACO) [22] and the performance model given in [13] (described in Section II), the estimates produced by PHIL yield BCSR matrix-vector multiply performance comparable to the performance obtained using estimates from OSKI.

A. System

We ran all of our experiments on a node with two sockets, each with a 12-core Intel® Xeon™ Processor E5-2695 v3

[1]Our code is available under the BSD 3-clause license at https://github.com/peterahrens/FillEstimation/releases/tag/IPDPS2018

"Ivy Bridge" at 2.4 GHz. Each core has 32 KB of L1 cache and 256 KB of L2 cache. Each socket has 30 MB of shared L3 cache.

TACO generates parallel BCSR kernels for each block size which we ran on one socket with 12 threads. Both PHIL and OSKI implementations run serially and use the `mt19937` random number generator from the C++ Standard Library.

B. Test Cases

In Figure 3, we test our implementation on a suite of 42 matrices inspired by the test set in [11]. All but two are from the University of Florida Sparse Matrix Collection [1]. These matrices were chosen to represent a variety of application domains and block structures.

In Figure 4, we focus on four of these matrices, two of which were used by Vuduc et al. to measure OSKI [11]. We describe two pathological cases we invented to induce worst-case behavior in PHIL and OSKI, respectively.

`pathological_PHIL` is a matrix designed to bring out the worst in our PHIL algorithm. Let $\mathcal{A}$ be a worst case tensor for some blocking scheme $\mathbf{b}$. Assume for contradiction that there are nonzero blocks which are not completely full and contain more than one nonzero. We can add nonzeros to more than half full blocks and remove nonzeros from more than half empty blocks to increase the *variance* of each of each sample $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$. This increases the variance of the PHIL estimator $F_{\mathbf{b}}(\mathcal{A})$, which increases the probability that it will lie farther from its mean. Thus, our worst case matrix has only completely full blocks and blocks with only one nonzero. One can show formally that the variance of $F_{\mathbf{b}}$ is maximized when these two types of blocks occur in equal number. For our concrete test case, we create a $10,000 \times 10,000$ matrix with $10,000$ full 12×12 blocks and $10,000$ sparse 12×12 blocks. PHIL should perform poorly on this matrix.

`pathological_OSKI` is a matrix designed to bring out the worst in the OSKI algorithm. Because OSKI samples rows with equal probability, hiding many blocks which look different from the rest of the matrix in a single row should cause OSKI to perform poorly. This matrix is of size $100,000 \times 100,000$, and the first 6 rows are dense, while all other rows have only a single nonzero in the first column.

C. Metrics

Since autotuning algorithms typically run at runtime before execution of the tuned operation, the speedups gained by autotuning must be weighed against the execution time of the algorithm. Since our example operation to autotune is sparse matrix-vector multiplication, we normalize the time taken to perform fill estimation by the time it takes to perform a parallel CSR matrix-vector multiply without blocking.

We use the simple performance model described by Vuduc. et al. in [13] and summarized in Section II to select a block size. Since the modeled performance is proportional to the fill, we judge the quality of a fill estimate using the maximum relative error.

Definition V.1. The maximum relative error of a fill estimate f is

$$\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}}.$$

Assume that for some fill estimates f the maximum relative error is ϵ. Since the performance model is proportional to the fill, our approximate performance model is accurate to within a factor of $(1+\epsilon)$ from the true performance model that uses the true fill F. We choose the block size maximizing our approximate model. Consider the best guess block size which maximizes the true model. Since our approximate models of both the chosen block size and the best guess is accurate to within a factor of $(1+\epsilon)$ from the true model, the true modeled performance of our chosen block size is at most a factor of $(1+\epsilon)^2$ from the true modeled performance of the best guess. We therefore measure the mean over several trials of the maximum relative error over all block sizes. Note that if the maximum relative error is greater than 1, this represents a complete loss of accuracy, as a bogus algorithm which returns 0 for the estimated fill of all block sizes would achieve a better maximum relative error.

D. Experiments

Figure 3 compares the the estimation algorithms in terms of runtime, mean maximum relative error, and the resulting BCSR matrix-vector multiply time of the selected block sizes on our suite of 42 matrices with fixed values of ϵ, δ, and σ. The block sizes are chosen using the performance model in [13] and all blocked and non-blocked matrix-vector multiplies are performed using TACO [22]. The data shows that in most cases, PHIL was more accurate and much faster than OSKI. PHIL always produced results with a mean maximum relative error less than .05, while in a few cases OSKI produced results with a mean maximum relative error which was worse or much worse than 1.

Since PHIL uses a fixed number of samples, normalizing the runtime of PHIL shows that PHIL takes longer relative to the parallel CSR matrix-vector multiplication time on smaller matrices. However, on the larger matrices (when autotuning is most important) PHIL usually takes at most 10 matrix-vector multiplies, outperforming OSKI by factors of 10 to 40.

Both the PHIL and OSKI estimates led to remarkably similar BCSR matrix-vector multiplication times. It may be possible to improve the chosen block sizes with a more complex performance model [18], but our focus is on estimating the fill and not on modeling the performance of sparse kernels.

Figure 3 also shows that for a fixed setting of parameters, the runtime and relative error of our fill estimation algorithms varies substantially from matrix to matrix (although the relative error of PHIL is consistently small). We wish to compare the algorithms across all settings of parameters. Therefore, Figure 4 shows the mean maximum relative error as a function of the runtime of the estimation algorithm on four different matrices. Both axes use logarithmic scale.

Figure 4 shows that PHIL provides better estimates of the fill than OSKI for any amount of time invested. On these four matrices, PHIL is both more efficient and more accurate than OSKI. We ran both PHIL and OSKI for longer on the pathological cases in order to see them produce good estimates. On `pathological_PHIL`, PHIL performs better than OSKI, but the difference is smaller than on the practical matrices. On `pathological_OSKI`, OSKI fails to estimate the fill in any reasonable time.

VI. CONCLUSION

PHIL estimates the fill of a sparse matrix at least 2 times faster than OSKI on most of our real-world inputs and provides useful estimates of the fill even in pathological test cases. We found that PHIL and OSKI produced comparable speedups in blocked sparse matrix-vector multiply in most cases using their recommended parameters. PHIL produced far more accurate estimates of the fill than its worst-case accuracy guarantee.

Sampling techniques are useful in autotuning since we can often sacrifice some accuracy in the heuristics for a faster autotuner. As libraries for numerical computation evolve and autotuning moves from compile-time to run-time implementations, developers will need efficient heuristics [29]. This work indicates broader potential for sampling techniques in the design of autotuned numerical software. The creation of faster sampling algorithms with provable guarantees will allow library developers to write software that can more accurately specialize to user data and provide the best possible performance for their application and hardware.

A. Future Work

A major motivation in the design of our algorithm was to express the problem as a dense set of operations that can be computed efficiently. Future work includes a parallel, optimized implementation of PHIL and an extension to handle sparse tensors in multiple storage formats. COMPUTE$\mathcal{X}$ should benefit from instruction-level parallelism. PHIL should also easily extend to a multicore implementation. Because PHIL computes different samples independently and PHIL only reads from the sparse tensor, each thread can compute a local sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ which we can reduce at the end.

B. Extensions

Some formats store their blocks in a sparse format [7], [10]. These blocks are usually much larger than the blocks mentioned in this paper, but we can extend an algorithm for Problem II.1 to estimate the fill of larger block sizes by limiting our attention to multiples of some base block size.

Problem VI.1 (Coarse Fill Estimation)**.** Given a tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$, a base block size $\mathbf{q}$, and a maximum multiplier B, compute an approximation $F_{\mathbf{b}}(\mathcal{A})$ accurate to within a factor of ϵ for all $\mathbf{b}$ where $b_r = b'_r q_r$ and $1 \leq \mathbf{b}' \leq B$ with probability $1 - \delta$.

We can create a tensor $\mathcal{A}' \in \mathbb{F}^{I'_1 \times I'_2 \times \cdots \times I'_R}$ where $\mathcal{A}'[\mathbf{j}]$ is the number of nonzeros in block $\mathbf{j}$ of $\mathcal{A}$ under the blocking scheme $\mathbf{q}$. Notice that $f_{\mathbf{b}'}(\mathcal{A}') = f_{\mathbf{b}}(\mathcal{A})$, so a solution to Problem II.1 on $\mathcal{A}'$ is a solution to Problem VI.1 on $\mathcal{A}$. Since $k(\mathcal{A}') \leq k(\mathcal{A})$, $\mathbf{I}' \leq \mathbf{I}$, and we can construct $\mathcal{A}'$ in $O(k(\mathcal{A}))$ time, most algorithms (including PHIL) that solve Problem II.1 can solve Problem VI.1 with an addition of $O(k(\mathcal{A}))$ to their asymptotic running time.

ACKNOWLEDGMENT

This research was supported in part by an Intel ISTC grant, a Darpa XDATA grant, Aramco, NSF Grants DMS-1312831, 1314547 and 1533644, as well as a DOE Computational Science Graduate Fellowship DE-FG02-97ER25308, a National Physical Sciences Consortium Fellowship, and an Akamai Presidential Fellowship. Thanks to Jiajia Li, Richard Vuduc, Charles Leiserson, Saman Amarasinghe, and David Karger for the helpful discussions.

REFERENCES

[1] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

[2] B. W. Bader and T. G. Kolda, "Efficient MATLAB Computations with Sparse and Factored Tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, Jan. 2008.

[3] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text." ACM Press, 2013, pp. 165–172.

[4] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, Dec. 2014.

[5] A. Carlson, J. Betteridge, B. Kisiel, and B. Settles, "Toward an Architecture for Never-Ending Language Learning." vol. 5, 2010, p. 3.

[6] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrixvector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, Mar. 2009.

[7] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks." ACM Press, 2009, p. 233.

[8] A. Pinar and M. T. Heath, "Improving Performance of Sparse Matrix-Vector Multiplication," in *Supercomputing, ACM/IEEE 1999 Conference*, Nov. 1999, pp. 30–30.

Matrix Information			$B = 12$						$B = 4$					
			Normalized Time to Estimate Fill		Mean Maximum Relative Error		Normalized TACO SpMV Time (Vuduc et al. Model)		Normalized Time to Estimate Fill		Mean Maximum Relative Error		Normalized TACO SpMV Time (Vuduc et al. Model)	
Name	NNZ (k)	Size (m + n)	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI	PHIL	OSKI
Domain: 2D/3D Problem														
nd24k	28,715,634	144,000	8.707	104.8	0.030	0.014	0.710	0.710	3.126	9.740	0.007	0.002	0.715	0.715
BenElechi1	13,150,496	491,748	9.983	88.40	0.022	0.011	0.761	0.761	3.286	11.33	0.003	0.003	0.766	0.766
kim2	11,330,020	913,952	10.55	99.07	0.033	0.006	1.0*	1.0*	3.481	14.33	0.010	0.002	1.0*	1.0*
nd6k	6,897,316	36,000	21.32	77.71	0.031	0.027	0.702	0.702	6.693	8.896	0.006	0.004	0.726	0.726
nd3k	3,279,690	18,000	46.77	83.11	0.030	0.038	0.704	0.704	14.23	9.505	0.006	0.007	0.573	0.573
Domain: Computational Fluid Dynamics														
atmosmodl	10,319,760	2,979,504	7.232	99.00	0.023	0.007	1.0*	1.0*	2.483	20.88	0.008	0.001	1.0*	1.0*
3dtube	3,213,618	90,660	42.54	93.89	0.022	0.069	0.566	0.566	12.30	11.37	0.008	0.015	0.571	0.571
Domain: Graph														
hugetric-00010	19,771,708	13,185,530	1.180	52.72	0.009	0.005	1.0*	1.0*	0.474	15.72	0.004	0.001	0.996	0.996
kron_g500-logn17	10,228,360	262,144	4.256	35.95	0.004	0.045	1.0*	1.0*	1.642	4.537	0.001	0.012	1.0*	1.0*
flickr	9,837,214	1,641,756	0.975	9.203	0.006	0.038	1.0*	1.0*	0.370	1.636	0.002	0.012	0.743	0.743
pdb1HYS	4,344,765	72,834	29.52	76.72	0.024	0.040	0.523	0.523	8.821	9.095	0.006	0.012	0.514	0.514
fl2010	2,346,294	968,962	9.022	28.63	0.006	0.009	1.0*	1.0*	3.129	7.381	0.002	0.003	0.486	0.486
in2010	1,281,716	534,142	27.90	48.48	0.008	0.015	1.000	1.000	10.14	13.13	0.002	0.004	1.0*	1.0*
ok2010	1,274,148	538,236	37.00	62.39	0.007	0.012	1.000	1.000	13.08	17.45	0.002	0.004	1.0*	1.0*
Domain: Linear Programming														
spal	46,168,124	331,899	1.991	35.54	0.015	0.025	0.948	0.948	0.848	3.779	0.005	0.008	0.575	0.575
rail4284	11,284,032	1,101,178	6.869	47.63	0.017	0.375	1.0*	1.0*	2.567	4.960	0.007	0.124	1.0*	1.0*
degme	8,127,528	844,916	11.14	77.90	0.017	0.082	1.0*	1.0*	3.655	9.981	0.005	0.074	1.0*	1.0*
gupta1	2,164,210	63,604	34.21	51.24	0.024	0.516	1.0*	1.0*	11.46	6.641	0.008	0.215	1.0*	1.0*
pds-100	1,096,002	670,820	57.17	78.18	0.004	0.027	1.0*	1.0*	20.55	19.81	0.002	0.010	1.0*	1.0*
Domain: Mathematical Optimization														
largebasis	5,560,100	880,040	15.62	86.99	0.025	0.015	1.0*	1.0*	4.572	15.14	0.008	0.004	1.0*	1.0*
exdata_1	2,269,501	12,002	24.40	33.45	0.033	4.518	0.539	0.533	7.181	3.575	0.004	0.019	0.565	0.545
Domain: Power Network														
TSOPF_RS_b2383	16,171,169	76,240	8.238	63.53	0.034	0.072	0.782	0.750	3.417	6.908	0.005	0.007	0.791	0.791
kkt_power	14,612,663	4,126,988	3.591	66.57	0.008	0.014	1.000	1.000	1.458	15.00	0.004	0.003	1.0*	1.0*
Domain: Structural														
af_shell10	52,672,325	3,016,130	2.664	101.3	0.026	0.004	0.859	0.859	1.236	19.79	0.007	0.002	1.0*	1.0*
ldoor	46,522,475	1,904,406	2.946	90.50	0.024	0.011	0.817	0.817	1.154	12.47	0.007	0.005	1.0*	1.0*
Emilia_923	41,005,206	1,846,272	3.402	94.47	0.021	0.010	0.812	0.812	1.330	13.16	0.006	0.003	0.795	0.795
inline_1	36,816,342	1,007,424	3.299	71.53	0.019	0.014	0.757	0.757	1.304	9.360	0.007	0.004	0.786	0.786
F1	26,837,113	687,582	2.917	46.61	0.018	0.019	0.923	0.923	1.136	6.089	0.006	0.006	0.640	0.640
af_shell9	17,588,875	1,009,710	6.927	88.24	0.024	0.007	0.808	0.808	2.386	12.44	0.007	0.004	0.955	0.955
halfb	12,387,821	449,234	10.77	89.29	0.026	0.029	0.949	0.951	3.637	11.60	0.007	0.009	0.933	0.938
troll	11,985,111	426,906	12.16	95.70	0.024	0.029	0.786	0.786	4.125	12.42	0.006	0.008	0.824	0.824
pwtk	11,634,424	435,836	10.73	86.56	0.035	0.018	0.883	0.883	3.478	10.75	0.007	0.007	0.887	0.887
fcondp2	11,294,316	403,644	10.54	79.80	0.022	0.029	0.669	0.669	3.579	10.14	0.005	0.008	0.697	0.697
crankseg_1	10,614,210	105,608	11.69	71.01	0.023	0.050	0.808	0.808	4.290	8.386	0.008	0.018	0.915	0.914
m_t1	9,753,570	195,156	14.52	92.26	0.020	0.039	0.754	0.754	4.880	10.64	0.005	0.013	0.726	0.726
gearbox	9,080,404	307,492	13.45	82.71	0.022	0.034	0.717	0.717	4.475	10.60	0.007	0.009	0.693	0.693
ship_001	4,644,230	69,840	27.40	85.27	0.029	0.062	0.984	0.988	8.653	9.923	0.008	0.024	1.0*	1.0*
s3dkt3m2	3,753,461	180,898	33.18	94.30	0.034	0.021	0.876	0.875	9.185	12.36	0.007	0.009	0.864	0.863
ct20stif	2,698,463	104,658	46.08	88.01	0.025	0.069	0.770	0.773	14.24	11.76	0.008	0.022	0.821	0.824
nasasrb	2,677,324	109,740	46.92	88.56	0.019	0.042	0.616	0.617	13.89	11.60	0.005	0.018	0.735	0.735
Domain: Synthetic														
pathological_PHIL	1,449,856	239,988	61.10	79.58	0.045	0.073	0.983	0.983	18.88	15.84	0.006	0.010	1.0*	1.0*
pathological_OSKI	699,994	200,000	22.98	14.93	0.013	3.666	0.634	0.634	8.548	4.483	0.005	1.800	0.794	0.800

Figure 3. Over 42 different matrices, we show the mean estimation time, mean maximum relative error (Definition V.1), and the resulting mean parallel sparse matrix-vector multiply (SpMV) time in BCSR format when the performance model in [13] is used to select a block size. Times are normalized to the mean time taken to perform one parallel sparse matrix-vector multiply (SpMV) on the unblocked CSR matrix. All means are the average of 100 trials. All blocked and non-blocked matrix-vector multiplies are performed using TACO [22]. Highlighted cells show the better result between PHIL and OSKI. The left group of columns corresponds to a maximum considered block size $B = 12$. The right group of columns corresponds to a maximum considered block size of $B = 4$. The parameters to PHIL are $\epsilon = 3$ and $\delta = 0.01$ when $B = 12$, and they are $\epsilon = 0.25$ and $\delta = 0.01$ when $B = 4$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting) for all cases. To create the performance matrix P for the performance model, we timed BCSR matrix-vector multiplication performance for 100 trials on a 1000×1000 dense matrix. The matrices are organized by application domain and described in more detail in Section V-B. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since most autotuners will try both an unblocked CSR format and the predicted best block size with BCSR format, they may choose to use CSR if no speedup is observed and so these results are listed as 1.0.

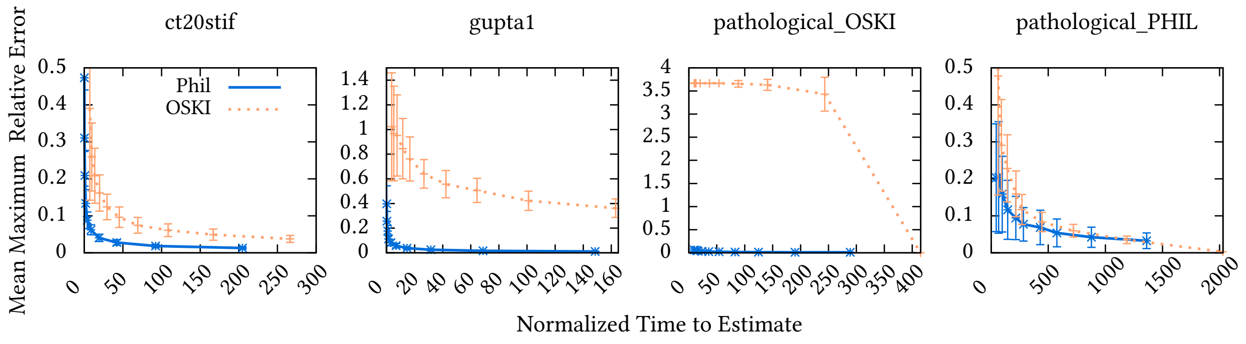


Figure 4. Mean maximum relative error (Definition V.1) as a function of mean estimation time (normalized to the mean time it takes to perform a parallel sparse matrix-vector multiplication in CSR format using TACO [22]) for four matrices. Both axes use logarithmic scale. All means are the average of 100 trials. The error bars reflect one standard deviation above and below the mean. The blue solid line represents PHIL and the orange dotted line represents OSKI. Each point is a separate setting for the parameters. `3dtube` is a matrix arising from the application of finite element analysis to a computational fluid dynamics problem. This matrix consists mostly of 3×3 dense blocks (96% of nonzeros reside in these blocks). `gupta1` is the matrix representation of a linear programming problem, and has no obvious block structure. The pathological matrices are described in more detail in Section V-B. Note that errors above 1 represent a complete loss of accuracy.

[9] V. Karakasis, G. Goumas, and N. Koziris, "A Comparative Study of Blocking Storage Methods for Sparse Matrices on Multicore Architectures." IEEE, 2009, pp. 247–256.

[10] A. N. Yzelman, "Generalised Vectorisation for Sparse Matrix: Vector Multiplication," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA^3 '15. New York, NY, USA: ACM, 2015, pp. 6:1–6:8.

[11] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, CA, USA, Jan. 2004.

[12] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices." ACM Press, 2015, pp. 1–8.

[13] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply." IEEE, 2002, pp. 26–26.

[14] E.-J. Im and K. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Computational Science ICCS 2001*, G. Goos, J. Hartmanis, J. van Leeuwen, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2073, pp. 127–136.

[15] E.-J. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jun. 2000.

[16] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, Feb. 2004.

[17] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, pp. 521–530, Jan. 2005.

[18] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance Optimization and Modeling of Blocked Sparse Kernels," *The International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 467–484, Nov. 2007.

[19] V. Karakasis, G. Goumas, and N. Koziris, "Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels," in *2009 International Conference on Parallel Processing*, Sep. 2009, pp. 356–364.

[20] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM SIGPLAN Notices*, vol. 45, no. 5, p. 115, May 2010.

[21] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication." IEEE, May 2015, pp. 61–70.

[22] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The Tensor Algebra Compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017.

[23] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor." ACM Press, 2015, pp. 1–7.

[24] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, May 2007.

[25] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication," *ACM SIGPLAN Notices*, vol. 48, no. 6, p. 117, Jun. 2013.

[26] D. Langr, I. Šimeček, and T. Dytrych, "Block Iterators for Sparse Matrices," Oct. 2016, pp. 695–704.

[27] W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *Journal of the American Statistical Association*, vol. 58, no. 301, p. 13, Mar. 1963.

[28] R. Bardenet and O.-A. Maillard, "Concentration inequalities for sampling without replacement," *Bernoulli*, vol. 21, no. 3, pp. 1361–1385, Aug. 2015.

[29] J. Dongarra and V. Eijkhout, "Self-Adapting Numerical Software for Next Generation Applications," *The International Journal of High Performance Computing Applications*, vol. 17, no. 2, pp. 125–131, May 2003.