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

by

Willow Ahrens

B.S. Electrical Engineering and Computer Science
Major in Computer Science, Minor in Mathematics
University of California, Berkeley, 2016

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

at the

Massachusetts Institute of Technology

February 2019

Signature of Author: _____

Department of Electrical Engineering and Computer Science
January 2, 2019

Certified By: _____

Alan Edelman
Professor of Applied Mathematics
Computer Science and AI Laboratories
Applied Computing Group Leader

Accepted By: _____

Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

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

by

Willow Ahrens

Submitted to the Department of Electrical Engineering and Computer Science on January 2, 2019 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

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 parallelize Phil, and refer to the parallel implementation as PPhil. We compare Phil, PPhil, and OSKI on a suite of 42 matrices. On average, PPhil was able to produce a fill estimate in 1.3810 times the time it took to compute one sparse matrix vector multiply, which was 61.176 times faster than OSKI. The maximum error generated by Phil was 0.0480, while OSKI sometimes produced estimates with a complete loss of accuracy. 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.

Much of the work presented in this thesis appears in [1], a paper coauthored with Helen Xu and Nicholas Schiefer. The parallel algorithm PPhil and its implementation are novel contributions of this thesis. Helen's masters thesis is also based on [1], and adds additional test matrices [2].

Thesis Supervisor: Alan Edelman

Title: Professor of Applied Mathematics

Acknowledgments

I'd first like to thank my collaborators Helen Xu and Nicholas Schiefer. I've had the good fortune to rely on them as homework buddies through difficult qualifying courses, collaborators on an awesome paper, and friends who share incredible emoji 😂. I also want to thank my parents James Ahrens and Christine Sweeney for their love and keen advice, and all my friends in the MIT Theory Group, the MIT Commit Group, the MIT Supertech Group, the Julia Lab, the MIT Hobby Shop, and the MIT Glass Lab. I thank my advisor Alan Edelman for his encouragement, perspective, and great conversations. Further thanks to Jiajia Li, Richard Vuduc, Fredrik Kjolstad, Charles Leiserson, Saman Amarasinghe, and David Karger for the helpful discussions.

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. 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.

1 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 [3, 4]. For example, sparse tensors have applications in review systems [5], quantum chemistry [6], and natural language processing [7]. 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 [8, 9].

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 [4, 9–12].

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 [11]. Several sparse matrices and tensors in scientific computing lend themselves naturally to blocked structures. For example, sparse matrices arising from finite element methods [13] and sparse tensors arising in quantum chemistry [14] 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 [15]. 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 [16]. Many subsequent performance models for matrices have been formulated in terms of the fill or directly related quantities [8, 13, 15–22]. In the absence of an efficient fill estimation algorithm, block size selection for sparse tensors has been limited to empirical search [23].

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 [13]. 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.

1.1 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. We also show how to parallelize PHIL, and will refer to our multicore parallel implementation of PHIL as PPHIL.

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 parallelize PHIL using OpenMP and separate samples. An initial attempt at parallelization was made in [2] but did not produce estimates of the same quality as PHIL. Our implementation PPHIL achieves better speedups and for the first time produces estimates of identical quality to PHIL.

We experimentally evaluate PHIL, PPHIL, and OSKI on a suite of sparse matrices and both "Ivy Bridge" and "Haswell" architectures. As expected, PHIL and PPHIL produced estimates of the same quality. The maximum error generated by PPHIL was 0.0480, while OSKI sometimes produced estimates with a complete loss of accuracy. On average, PPHIL was able to produce a fill estimate in 1.3810 times the time it took to compute one sparse matrix vector multiply, which was 61.176 times faster than OSKI. When estimating the fill of all blocks with size less than 12, PPHIL was an average of 6.6154 times faster than PHIL on "Ivy Bridge" and 9.6680 times faster than PHIL on "Haswell". We used the Tensor Algebra Compiler (TACO) to generate parallel blocked sparse matrix vector multiplication kernels [24]. PHIL, PPHIL, 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 [15].

2 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.

2.1 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 $\mathbf{I}'$.

2.2 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)* [4]. 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)* [10] 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 [4], such as *Compressed Sparse Fiber (CSF)* [24, 25]. 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 [11, 26].

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 [11]. 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) [10, 12].

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)* [10], 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 [23, 24]. In BCSR and BCSF, each block is stored in a dense format, with

zeros represented explicitly, and only blocks which contain nonzeros are stored.

2.3 Regular Blocking

Definition 2.1. 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.

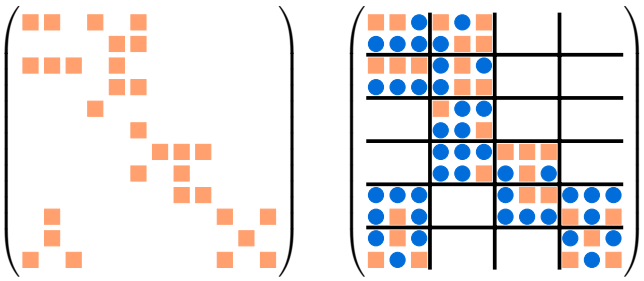


Figure 1: On the left, a sparse matrix before blocking. On the right, the same sparse matrix after blocking. The squares denote nonzero elements and circles are explicit zeros that are introduced due to the storage format. In this example, the blocking scheme $\mathbf{b} = (2, 3)$ and $k_{\mathbf{b}}(\mathcal{A})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

2.4 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:

Definition 2.2 ([16])**.** 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 [13, 15–20]. The fill has also been used to select block sizes for sparse triangular solve and sparse $\mathcal{A}^T\mathcal{A}\mathbf{x}$ [13]. The number of nonzero blocks (proportional to the fill) has been used in performance models for general blocked format sparse matrix-vector multiply [8, 21, 22]. Block size selection remains a difficult problem for tensors as it is difficult to estimate the fill, so developers have adopted empirical search techniques [23]. An estimate of the fill could easily be added as an additional feature in feature-based machine learning approaches to sparse kernel performance modeling [27].

As an example, we explain the simple performance model for blocked sparse matrix-vector multiply given in [15]. 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%) [13].

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 $b_1, b_2 \leq 12$ [13]. Thus, our problem is to quickly compute an estimate of the fill for these block sizes with reasonable accuracy.

Problem 2.1 (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.

2.5 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 [28]. 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 [13, 17]. Since the algorithm is implemented in the Optimized Sparse Kernel Interface (OSKI) library, we will refer to it as OSKI [19]. 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.

Estimating the fill of a single blocking scheme $\mathbf{b}$ can be thought of as a special case of the *count-distinct* problem, where the goal is to estimate the number of distinct elements in a stream. Consider the stream where each element is the block coordinate of a nonzero coordinate in $\mathcal{A}$. The number of distinct elements in this stream is equal to $k_{\mathbf{b}}(\mathcal{A})$, which is directly proportional to $f_{\mathbf{b}}(\mathcal{A})$. However, an optimal solution to the count-distinct problem must examine the entire stream (all the nonzeros) and use $\Omega(1/\epsilon^2 + \log(k(\mathcal{A})))$ space to achieve ϵ accuracy [29]. Thus, solving Problem 2.1 using B^R separate instances of the count-distinct problem would take at least $\Omega(B^R k(\mathcal{A}))$ time. However, the count-distinct problem may still prove useful to those who wish to estimate the fill of a very small number of large block sizes [30].

3 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 3.1 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 3.1 in Section 4.

Theorem 3.1. *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.

3.1 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 3.1. 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 3.2. For all $\mathbf{b}\leq B$:

$$F_{\mathbf{b}}:=\frac{b_1b_2\cdots b_R}{S}\sum_{j=1}^{S}x_{\mathbf{b}}(\mathcal{A},\mathbf{i}_j)$$

Theorem 3.2. *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 $b_1b_2\cdots b_R$ to obtain an estimator of $f_{\mathbf{b}}(\mathcal{A},\mathbf{i})$, by definition. □

We describe how PHIL computes $F_{\mathbf{b}}$ in Algorithm 3.1.

Algorithm 3.1. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_R}$ and B, compute an approximation to $f_{\mathbf{b}}(\mathcal{A})$ 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:

$0 \leq \delta \leq 1\,, \quad \epsilon > 0\,, \quad B \geq 1$

function EstimateFill($\mathcal{A}$, B, ϵ, δ)
  $\mathcal{Y} \in \mathbb{R}^{B \times \cdots \times B}$
  $\mathcal{F} \in \mathbb{R}^{B \times \cdots \times B}$
  $S \leftarrow \left\lceil \frac{B^{2R}}{2\epsilon^2} \ln\left(\frac{2B^R}{\delta}\right) \right\rceil$.
  $\mathcal{Y} \leftarrow 0$
  for $\mathbf{i} \in$ sample of size S with replacement from the nonzero coordinates of $\mathcal{A}$ **do**
    $\mathcal{Y} \leftarrow \mathcal{Y} +$ Compute$\mathcal{X}(\mathcal{A}, B, \mathbf{i})$
  for $\mathbf{b} \in B \times \cdots \times B$ **do**
    $\mathcal{F}[\mathbf{b}] \leftarrow \frac{b_1 b_2 \cdots b_R \mathcal{Y}[\mathbf{b}]}{s}$
  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 $(1-\delta)$.

3.2 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 3.2 and illustrated in Figure 2.

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

Algorithm 3.2. 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:
$\mathcal{A}[\mathbf{i}] \neq 0\,, \quad B \geq 1$

function COMPUTE$\mathcal{X}$($\mathcal{A}$, $\mathbf{i}$, B)
  $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B:\mathbf{i}+B-1}$
  $\mathcal{Z}_0 \leftarrow 0$
  for $\mathbf{j} \in$ NONZEROSINRANGE$(\mathcal{A}, \mathbf{i}-B, \mathbf{i}+B-1)$ **do**
    $\mathcal{Z}_0[\mathbf{j}] \leftarrow 1$
  for $r \in 1:R$ **do**
    for $j \in i_r - B + 1 : i_r + B - 1$ **do**

$$\mathcal{Z}_0[\underbrace{:,\ldots,:,j}_{r},:,\ldots,:] \leftarrow \mathcal{Z}_0[\underbrace{:,\ldots,:,j}_{r},:,\ldots,:] + \mathcal{Z}_0[\underbrace{:,\ldots,:j-1}_{r},:,\ldots,:]$$

  for $b_1 \in 1:B$ **do**

$$\mathcal{Z}_1 \leftarrow \mathcal{Z}_0[t_{b_1}(i_1),\underbrace{:,\ldots,:}_{r-1}] - \mathcal{Z}_0[h_{b_1}(i_1)-1,\underbrace{:,\ldots,:}_{r-1}]$$

    for $b_2 \in 1:B$ **do**

$$\mathcal{Z}_2 \leftarrow \mathcal{Z}_1[t_{b_2}(i_2),\underbrace{:,\ldots,:}_{r-2}] - \mathcal{Z}_1[h_{b_2}(i_2)-1,\underbrace{:,\ldots,:}_{r-2}]$$

$\vdots$

      for $b_R \in 1:B$ **do**
        $\mathcal{Z}_R \leftarrow \mathcal{Z}_{R-1}[t_{b_R}(i_R)] - \mathcal{Z}_{R-1}[h_{b_R}(i_R)-1]$
        $\mathcal{X}[\mathbf{b}] \leftarrow \frac{1}{\mathcal{Z}_R}$

Ensure:
$\mathcal{X}[\mathbf{b}] \leftarrow x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$

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 $\mathcal{Z}$.

3.3 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 $O(3^R B^R)$.

4 Analysis

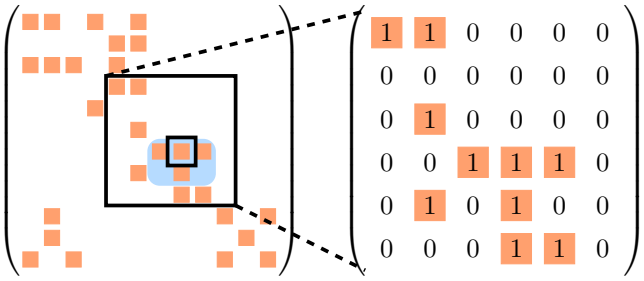
We want to select the number of samples, S, as small as possible for efficiency while still having provable guarantees on the concentration of our unbiased estimator $\sum_j x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)/S$. We use Hoeffding's inequality [31] as a concentration bound for sampling with replacement.

Theorem 4.1 (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 $t \geq 0$,*

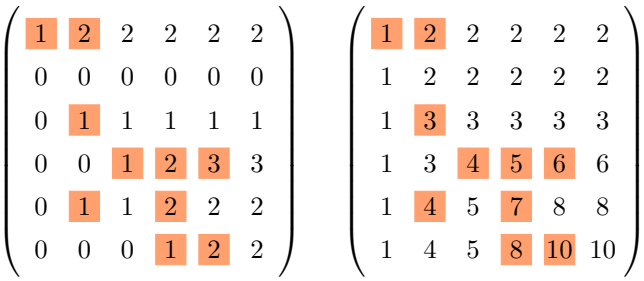
$$\Pr\left[\left|\overline{X} - \mathbb{E}[X]\right| \geq t\right] \leq 2\exp(-2Mt^2).$$

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 4.1:

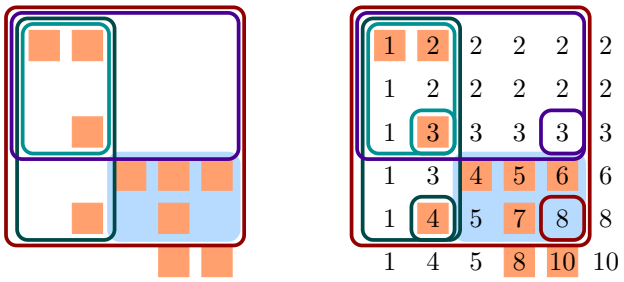
Figure 2: Here we visualize the execution of ComputeX as it computes one element of its output X. Specifically, we show how it computes $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) = \mathcal{X}[\mathbf{b}]$. In this example, our maximum block size is $B = 3$ and our nonzero of interest is $\mathbf{i} = (7, 8)$. 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.

Theorem 4.2 (Restatement of Theorem 3.1)**.** *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 3.2, $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}$. 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 2.2 is bounded to the same range. By Theorem 4.1,

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

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

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

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 [32]. 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.

5 Implementation

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 3.2. We made use of several library routines from the C++ Standard Library.

Several measures were taken to ensure efficient memory access. First, we generate a sample of linear nonzero indices (a list of integers l where l refers to the l^{th} nonzero in row-major order) into our matrix, using `std::mt19937`. We also use `std::seed_seq` with both a global seed and the trial number as sources of entropy to initialize random number generators in different trials differently. Next, we sort these linear indices (using `std::sort`) before converting them to (i, j) coordinate pairs (using `std::lower_bound`) as a subroutine and incrementing the row (i) coordinate as the conversion progresses. Thus, the sample coordinates are constructed and processed in row-major sorted order.

Random number generators on each thread are seeded using a global seed, the trial number, and the processor number as sources of entropy in a call to `std::seed_seq`.

5.1 Parallelization

Because PHIL computes samples independently, we can create a multicore parallel implementation of PHIL, which we will call PPHIL. We implement PPHIL using OpenMP. Line 6 contains the majority of the work in Algorithm 3.1, computing

$$\mathcal{Y}_{\mathbf{b}} = \sum_{j=1}^{S} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$$

for all $\mathbf{b} \leq B$. We can parallelize this computation by dividing the summation and computation of x among processors. Assume we have N processors and each processor n is responsible for samples $\mathbf{i}_{J_{n-1}+1}, \mathbf{i}_{J_{n-1}+2}, \ldots, \mathbf{i}_{J_n}$ such that $J_0 = 0$ and $J_N = S$. Then,

$$\mathcal{Y}_{\mathbf{b}} = \sum_{n=1}^{N} \sum_{j=J_{n-1}+1}^{J_n} x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}_j)$$

In a parallel execution, each processor uses a local copy of $\mathcal{Y}$ to accumulate a local sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$. After all iterations are complete, we can sum the local copies to obtain the true $\mathcal{Y}$, which we scale as in Line 9 of Algorithm 3.1 to produce the fill estimate.

We can also parallelize the construction of the sorted sample with replacement using a slight modification of the method described in [33]. We start by assigning each processor an equally sized, contiguous range of all possible linear nonzero indices. This keeps the processors operating on distinct sets of nonzeros which are more likely to be close to each other. Assume processor n is responsible for nonzeros $l_{n-1} + 1$ to l_n where $l_0 = 1$ and $l_N = k(\mathcal{A})$. If the nonzeros are sampled with replacement, the number of samples which processor n is responsible for $(J_n - J_{n-1})$ is binomially distributed. Letting $\beta(t, p)$ represent a binomially distributed random variable with t trials each with probability p, we can compute $\mathbf{J}$ once at the beginning of the algorithm using the following relationship.

$$J_n = J_{n-1} + \beta\left(J_N - J_{n-1}, \frac{l_n - l_{n-1}}{l_N - l_{n-1}}\right)$$

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

Thus, $J_n - J_{n-1}$ is the number of samples that would fall into the range $l_{n-1} + 1$ to l_n in a sample of S nonzero indices from 1 to N with replacement.

All that must be communicated to each processor n is the range of linear nonzero indices (l_{n-1} to l_n) and the number of samples that are to be taken in that range ($J_n - J_{n-1}$), and the rest of computation can then be performed independently. Finally, in parallel, each processor can populate a local buffer with the required number of samples (with replacement) within each range, then sort these linear indices and create the coordinate pairs similarly to the serial version. Since the processors are responsible for equally, binomially, distributed amounts of work, it is very likely that the work will be close to balanced. This can be shown using the same concentration bound as before [31].

6 Results

We compare PHIL to the competing algorithm described in [13], which we will refer to as OSKI. We use a test suite inspired by matrices from [13] 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 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) [24] and the performance model given in [15] (described in Section 2), the estimates produced by PHIL yield BCSR matrix-vector multiply performance comparable to the performance obtained using estimates from OSKI.

6.1 System

We ran all of our experiments on two different nodes. The first node had two sockets, each with a 12-core Intel® Xeon™ Processor E5-2695 v2 "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.

The second node had two sockets, each with a 16-core Intel® Xeon™ Processor E5-2698 v3 "Haswell" at 2.3 GHz. Each core has 64 KB of L1 cache and 256 KB of L2 cache. Each socket has 40 MB of shared L3 cache.

We found that our parallel algorithms (PPHIL and the kernels generated by TACO) ran fastest on one socket, with 12 threads for the "Ivy Bridge" architecture and 16 threads for the "Haswell" architecture.

We compiled our fill estimation kernels using `g++` using the flags `-std=c99 -fopenmp -O3 -march=native -ffast-math` TACO generates parallel BCSR kernels for each block size, using constant loop bounds for the inner loops. It compiled these kernels with `gcc` using the flags `-std=c99 -fopenmp -O3 -march=native -ffast-math -funroll-loops` Our implementations of PHIL and OSKI ran serially, and the implementation of PPHIL ran in parallel.

6.2 Test Cases

In Figure 3, we test our implementations on a suite of 42 matrices inspired by the test set in [13]. All but two are from the University of Florida Sparse Matrix Collection [3]. 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 [13]. 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.

6.3 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 [15] and summarized in Section 2 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 6.1. 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.

6.4 Experiments

Figure 3 compares the the estimation algorithms in terms of mean estimation time, mean maximum relative error (Definition 6.1), 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. All means are the average of 100 trials. Times are normalized to the mean time taken to perform one parallel sparse matrix-vector multiply (SpMV) on the unblocked CSR matrix. The block sizes are chosen using the performance model in [15]. 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. All blocked and non-blocked matrix-vector multiplies are performed using TACO [24].

The data shows that PHIL was both accurate and efficient.

Over all matrices, architectures, and maximum block sizes, the maximum error observed by PPHIL and PHIL was 0.0480, while in a few cases OSKI produced results with a mean maximum relative error which was worse or much worse than 1, a complete loss of accuracy.

On average, PPHIL was able to produce a fill estimate in 1.3810 times the time it took to compute one sparse matrix vector multiply (2.0440 times when $B = 12$, 0.7181 times when $B = 4$). This was an average of 61.176 times faster than OSKI. To compare the two serial algorithms, PHIL was an average of 7.8160 times faster than OSKI. Since PHIL and PPHIL use the same 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, PHIL was more efficient on larger matrices (when autotuning is most important). When $B = 12$ and there were more than $10,000,000$ nonzeros, PPHIL was able to produce a fill estimate in an average of 0.7268 times the time it took to compute one sparse matrix vector multiply.

Our parallelization strategy was effective. On our "Ivy Bridge" architecture (with 12 cores), when $B = 12$, PPHIL was 6.6154 times faster than PHIL. On our "Haswell" architecture (with 16 cores), when $B = 12$, PPHIL was 9.6680 times faster than PHIL.

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 [20], 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 on four interesting test matrices, PHIL provides better estimates of the fill than OSKI for any amount of time invested. We ran the implementations for longer on the pathological cases in order to see them produce good estimates. Out of all four matrices, PHIL and OSKI have the most similar performance on `pathological_PHIL`. On `pathological_OSKI`, OSKI fails to estimate the fill in any reasonable time.

7 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

Figure 3

(a) We compare the fill estimation capabilities of PHIL, PPHIL (a parallel version of PHIL), and OSKI on our **"Ivy Bridge"** system with a maximum considered block size of $\mathbf{B} = \mathbf{4}$. The parameters to PHIL and PPHIL are $\epsilon = 0.25$ and $\delta = 0.01$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting). Highlighted cells show the better result between PHIL, PPHIL, and OSKI. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since an autotuner may choose to use CSR if no speedup is observed with the new block size, these results are listed as 1.0.

Matrix Information			Normalized Time to Estimate Fill			Mean Maximum Relative Error			Normalized SpMV Time ([15] Model)		
Name	k (#nonzeros)	Size (#rows + #cols)	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI
Domain: 2D/3D Problem											
nd24k	28,715,634	144,000	3.227	0.430	10.43	0.006	0.007	0.002	0.769	0.769	0.769
BenElechi1	13,150,496	491,748	2.710	0.542	10.17	0.003	0.003	0.004	0.639	0.639	0.639
kim2	11,330,020	913,952	2.956	0.602	14.27	0.010	0.011	0.002	1.0*	1.0*	1.0*
nd6k	6,897,316	36,000	7.576	1.378	10.15	0.007	0.007	0.004	0.752	0.752	0.752
nd3k	3,279,690	18,000	15.12	3.017	9.898	0.007	0.006	0.006	0.578	0.578	0.578
Domain: Computational Fluid Dynamics											
atmosmodl	10,319,760	2,979,504	1.998	0.453	20.49	0.008	0.007	0.001	1.000	1.000	1.000
3dtube	3,213,618	90,660	8.644	1.721	8.526	0.009	0.008	0.014	0.498	0.498	0.498
Domain: Computer Vision											
bundle_adj	20,208,051	1,026,702	0.862	0.143	3.925	0.005	0.006	0.023	0.708	0.708	0.708
Domain: Electromagnetics											
fem_hifreq_circuit	20,239,237	982,200	1.552	0.324	9.531	0.006	0.006	0.004	0.737	0.737	0.737
Domain: Graph											
hugetric-00010	19,771,708	13,185,530	0.365	0.073	14.59	0.004	0.004	0.001	1.0*	1.0*	1.0*
kron_g500-logn17	10,228,360	262,144	1.538	0.308	4.716	0.001	0.001	0.011	1.000	1.000	1.000
flickr	9,837,214	1,641,756	0.356	0.063	1.846	0.002	0.002	0.014	1.000	1.000	1.000
pdb1HYS	4,344,765	72,834	7.737	1.593	8.496	0.006	0.006	0.012	0.441	0.441	0.441
fl2010	2,346,294	968,962	4.528	0.924	13.12	0.002	0.002	0.003	1.000	1.000	1.000
in2010	1,281,716	534,142	7.612	1.809	15.35	0.003	0.003	0.004	1.0*	1.0*	1.0*
ok2010	1,274,148	538,236	7.632	1.469	12.36	0.002	0.002	0.004	1.0*	1.0*	1.0*
Domain: Linear Programming											
spal	46,168,124	331,899	1.481	0.220	6.737	0.005	0.005	0.008	0.981	0.981	0.981
rail4284	11,284,032	1,101,178	2.901	0.554	5.821	0.007	0.007	0.125	1.0*	1.0*	1.0*
degme	8,127,528	844,916	3.587	0.807	10.63	0.005	0.006	0.060	1.0*	1.0*	1.0*
gupta1	2,164,210	63,604	8.085	1.808	5.175	0.008	0.007	0.226	1.000	1.000	1.0*
pds-100	1,096,002	670,820	13.62	2.594	15.87	0.002	0.001	0.009	1.000	1.000	1.000
Domain: Mathematical Optimization											
largebasis	5,560,100	880,040	4.645	1.098	18.08	0.008	0.008	0.004	1.0*	1.0*	1.0*
exdata_1	2,269,501	12,002	7.431	1.607	3.818	0.004	0.005	0.018	0.468	0.468	0.467
Domain: Model Reduction Problem											
boneS10	55,468,422	1,829,796	1.043	0.175	12.33	0.007	0.007	0.002	0.754	0.754	0.754
Domain: Optimization Problem											
mip1	10,352,819	132,926	2.773	0.416	5.517	0.007	0.006	0.061	0.644	0.644	0.668
Domain: Power Network											
TSOPF_RS_b2383	16,171,169	76,240	3.012	0.432	6.579	0.005	0.005	0.008	0.707	0.707	0.707
kkt_power	14,612,663	4,126,988	1.078	0.220	13.58	0.004	0.004	0.003	1.000	1.000	1.000
Domain: Structural											
af_shell10	52,672,325	3,016,130	0.909	0.157	14.70	0.007	0.007	0.002	1.0*	1.0*	1.0*
ldoor	46,522,475	1,904,406	1.041	0.194	12.86	0.007	0.007	0.005	0.959	0.959	0.959
Emilia_923	41,005,206	1,846,272	1.258	0.248	14.15	0.006	0.006	0.003	0.870	0.870	0.870
inline_1	36,816,342	1,007,424	1.271	0.232	10.03	0.007	0.006	0.004	0.746	0.746	0.746
F1	26,837,113	687,582	1.170	0.237	6.856	0.006	0.007	0.006	0.658	0.658	0.658
af_shell9	17,588,875	1,009,710	2.147	0.447	12.30	0.007	0.007	0.004	0.892	0.892	0.889
halfb	12,387,821	449,234	2.880	0.571	10.17	0.007	0.007	0.009	0.771	0.771	0.771
troll	11,985,111	426,906	3.096	0.629	10.42	0.006	0.007	0.009	0.634	0.634	0.634
pwtk	11,634,424	435,836	2.987	0.598	10.25	0.007	0.007	0.006	0.802	0.802	0.802
fcondp2	11,294,316	403,644	2.933	0.625	9.663	0.005	0.006	0.007	0.654	0.654	0.654
crankseg_1	10,614,210	105,608	4.628	0.851	9.728	0.008	0.008	0.019	1.0*	1.0*	1.0*
m_t1	9,753,570	195,156	4.220	0.876	10.07	0.006	0.006	0.012	0.673	0.673	0.673
gearbox	9,080,404	307,492	4.508	0.923	11.78	0.007	0.007	0.010	0.770	0.770	0.770
bmw7st_1	7,339,667	282,694	3.989	0.830	9.847	0.008	0.008	0.014	0.778	0.778	0.787
ship_001	4,644,230	69,840	7.011	1.559	8.438	0.008	0.008	0.022	0.787	0.786	0.784
s3dkt3m2	3,753,461	180,898	9.433	2.032	12.64	0.007	0.007	0.009	0.789	0.789	0.789
ct20stif	2,698,463	104,658	9.911	2.187	8.895	0.008	0.008	0.022	0.734	0.734	0.734
nasasrb	2,677,324	109,740	13.40	2.612	12.28	0.005	0.005	0.019	0.549	0.549	0.549
Domain: Synthetic											
pathological_PHIL	14,499,856	240,000	3.003	0.565	8.575	0.005	0.006	0.005	0.661	0.661	0.661
pathological_OSKI	6,999,994	2,000,000	1.066	0.241	4.619	0.005	0.005	1.800	0.765	0.765	1.0*

Figure 3

(b) We compare the fill estimation capabilities of PHIL, PPHIL (a parallel version of PHIL), and OSKI on our **"Ivy Bridge"** system with a maximum considered block size of $\mathbf{B = 12}$. The parameters to PHIL and PPHIL are $\epsilon = 3$ and $\delta = 0.01$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting). Highlighted cells show the better result between PHIL, PPHIL, and OSKI. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since an autotuner may choose to use CSR if no speedup is observed with the new block size, these results are listed as 1.0.

Matrix Information			Normalized Time to Estimate Fill			Mean Maximum Relative Error			Normalized SpMV Time ([15] Model)		
Name	k (#nonzeros)	Size (#rows + #cols)	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI
Domain: 2D/3D Problem											
nd24k	28,715,634	144,000	7.772	1.028	87.77	0.031	0.031	0.016	0.796	0.796	0.796
BenElechi1	13,150,496	491,748	9.054	1.398	81.26	0.022	0.023	0.011	0.679	0.679	0.679
kim2	11,330,020	913,952	9.385	1.438	91.55	0.032	0.034	0.006	1.0*	1.0*	1.0*
nd6k	6,897,316	36,000	19.34	2.903	69.70	0.030	0.030	0.028	0.688	0.688	0.688
nd3k	3,279,690	18,000	49.43	6.774	86.04	0.031	0.030	0.038	0.617	0.617	0.617
Domain: Computational Fluid Dynamics											
atmosmodl	10,319,760	2,979,504	6.370	1.026	90.93	0.023	0.023	0.008	1.000	1.000	1.000
3dtube	3,213,618	90,660	42.28	6.467	97.21	0.024	0.024	0.070	0.554	0.554	0.554
Domain: Computer Vision											
bundle_adj	20,208,051	1,026,702	2.497	0.391	28.05	0.024	0.026	0.089	0.745	0.745	0.745
Domain: Electromagnetics											
fem_hifreq_circuit	20,239,237	982,200	4.467	0.676	70.50	0.016	0.015	0.013	0.740	0.740	0.740
Domain: Graph											
hugetric-00010	19,771,708	13,185,530	1.399	0.249	63.27	0.009	0.010	0.005	1.0*	1.0*	1.0*
kron_g500-logn17	10,228,360	262,144	4.064	0.732	35.14	0.004	0.004	0.045	1.0*	1.0*	1.0*
flickr	9,837,214	1,641,756	1.208	0.168	12.00	0.006	0.007	0.041	1.0*	1.0*	1.0*
pdb1HYS	4,344,765	72,834	32.87	4.602	84.24	0.022	0.024	0.039	0.543	0.543	0.543
fl2010	2,346,294	968,962	18.33	2.990	59.29	0.006	0.006	0.009	1.0*	1.0*	1.0*
in2010	1,281,716	534,142	33.67	5.712	61.44	0.008	0.008	0.015	1.000	1.000	1.000
ok2010	1,274,148	538,236	25.62	4.257	44.43	0.007	0.006	0.012	1.0*	1.0*	1.0*
Domain: Linear Programming											
spal	46,168,124	331,899	3.462	0.454	62.84	0.015	0.015	0.026	0.963	0.963	0.963
rail4284	11,284,032	1,101,178	5.552	0.802	37.88	0.017	0.017	0.388	0.775	0.775	0.817
degme	8,127,528	844,916	11.19	1.694	76.79	0.017	0.017	0.077	1.0*	1.0*	1.0*
gupta1	2,164,210	63,604	36.10	5.550	54.42	0.023	0.024	0.514	1.0*	1.0*	1.0*
pds-100	1,096,002	670,820	33.12	5.059	44.42	0.004	0.004	0.026	1.000	1.000	1.000
Domain: Mathematical Optimization											
largebasis	5,560,100	880,040	13.61	2.082	77.22	0.025	0.025	0.016	0.880	0.880	0.880
exdata_1	2,269,501	12,002	17.98	2.611	23.74	0.032	0.033	4.029	0.351	0.351	0.350
Domain: Model Reduction Problem											
boneS10	55,468,422	1,829,796	2.787	0.437	95.33	0.027	0.026	0.009	0.769	0.769	0.769
Domain: Optimization Problem											
mip1	10,352,819	132,926	8.063	1.153	45.83	0.030	0.032	0.334	0.626	0.626	0.650
Domain: Power Network											
TSOPF_RS_b2383	16,171,169	76,240	9.009	1.290	68.73	0.038	0.039	0.073	0.797	0.797	0.799
kkt_power	14,612,663	4,126,988	2.944	0.472	57.13	0.009	0.009	0.014	1.0*	1.0*	1.0*
Domain: Structural											
af_shell10	52,672,325	3,016,130	2.527	0.331	95.59	0.024	0.024	0.004	0.791	0.791	0.791
ldoor	46,522,475	1,904,406	2.792	0.498	87.46	0.025	0.024	0.012	0.764	0.764	0.764
Emilia_923	41,005,206	1,846,272	3.455	0.507	97.22	0.021	0.021	0.010	0.809	0.809	0.809
inline_1	36,816,342	1,007,424	3.505	0.506	76.41	0.019	0.020	0.014	0.840	0.840	0.840
F1	26,837,113	687,582	3.301	0.492	52.89	0.019	0.019	0.019	0.665	0.665	0.665
af_shell9	17,588,875	1,009,710	6.804	1.142	87.64	0.025	0.026	0.007	0.766	0.766	0.766
halfb	12,387,821	449,234	9.328	1.455	79.13	0.027	0.027	0.027	0.850	0.850	0.851
troll	11,985,111	426,906	11.66	1.931	92.59	0.024	0.023	0.030	0.766	0.766	0.766
pwtk	11,634,424	435,836	9.512	1.494	75.14	0.034	0.035	0.018	0.779	0.779	0.779
fcondp2	11,294,316	403,644	9.396	1.451	72.23	0.024	0.022	0.030	0.609	0.609	0.609
crankseg_1	10,614,210	105,608	13.46	2.021	81.54	0.025	0.023	0.049	0.952	0.952	0.982
m_t1	9,753,570	195,156	12.66	1.854	78.65	0.020	0.020	0.037	0.667	0.667	0.667
gearbox	9,080,404	307,492	14.17	2.165	86.71	0.022	0.022	0.035	0.727	0.727	0.727
bmw7st_1	7,339,667	282,694	16.65	2.658	87.87	0.026	0.026	0.038	1.0*	1.0*	1.0*
ship_001	4,644,230	69,840	26.79	3.490	81.01	0.030	0.028	0.060	0.863	0.863	0.867
s3dkt3m2	3,753,461	180,898	22.77	3.080	65.90	0.033	0.033	0.021	0.597	0.597	0.598
ct20stif	2,698,463	104,658	34.57	5.596	65.73	0.026	0.026	0.067	0.532	0.532	0.555
nasasrb	2,677,324	109,740	35.83	5.236	76.95	0.020	0.019	0.043	0.378	0.378	0.378
Domain: Synthetic											
pathological_PHIL	14,499,856	240,000	8.006	1.157	70.89	0.042	0.045	0.042	0.659	0.659	0.663
pathological_OSKI	6,999,994	2,000,000	2.532	0.374	18.15	0.012	0.014	3.666	0.774	0.774	0.782

Figure 3

(c) We compare the fill estimation capabilities of PHIL, PPHIL (a parallel version of PHIL), and OSKI on our **"Haswell"** system with a maximum considered block size of $\mathbf{B} = \mathbf{4}$. The parameters to PHIL and PPHIL are $\epsilon = 0.25$ and $\delta = 0.01$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting). Highlighted cells show the better result between PHIL, PPHIL, and OSKI. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since an autotuner may choose to use CSR if no speedup is observed with the new block size, these results are listed as 1.0.

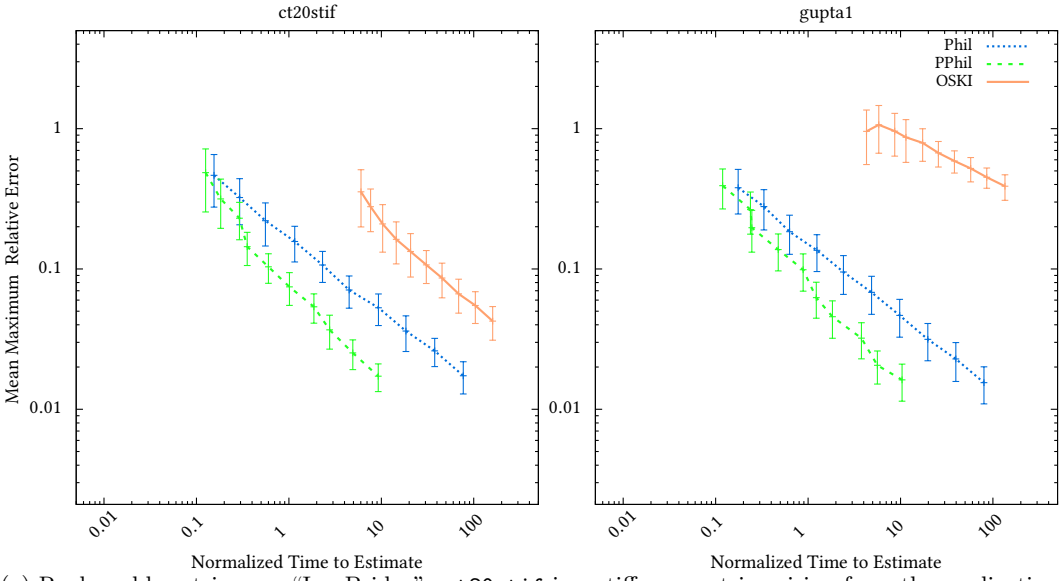
Matrix Information			Normalized Time to Estimate Fill			Mean Maximum Relative Error			Normalized SpMV Time ([15] Model)		
Name	k (#nonzeros)	Size (#rows + #cols)	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI
Domain: 2D/3D Problem											
nd24k	28,715,634	144,000	1.905	0.259	6.889	0.006	0.007	0.002	0.482	0.482	0.482
BenElechi1	13,150,496	491,748	1.787	0.174	7.001	0.003	0.003	0.003	0.492	0.492	0.492
kim2	11,330,020	913,952	1.548	0.210	6.794	0.011	0.010	0.002	0.645	0.645	0.645
nd6k	6,897,316	36,000	3.905	0.459	5.847	0.006	0.006	0.005	0.417	0.417	0.417
nd3k	3,279,690	18,000	25.50	3.004	19.20	0.007	0.006	0.007	0.445	0.445	0.445
Domain: Computational Fluid Dynamics											
atmosmodl	10,319,760	2,979,504	1.275	0.171	9.999	0.007	0.008	0.001	1.000	1.000	1.000
3dtube	3,213,618	90,660	20.55	2.069	21.00	0.008	0.008	0.015	0.587	0.587	0.587
Domain: Computer Vision											
bundle_adj	20,208,051	1,026,702	0.741	0.072	3.300	0.005	0.005	0.023	0.630	0.630	0.630
Domain: Electromagnetics											
fem_hifreq_circuit	20,239,237	982,200	1.319	0.192	7.792	0.005	0.005	0.004	0.759	0.759	0.759
Domain: Graph											
hugetric-00010	19,771,708	13,185,530	0.512	0.050	18.12	0.004	0.004	0.001	1.000	1.000	1.000
kron_g500-logn17	10,228,360	262,144	1.389	0.119	4.601	0.001	0.001	0.012	1.0*	1.0*	1.0*
flickr	9,837,214	1,641,756	0.409	0.036	1.843	0.002	0.003	0.013	1.000	1.000	1.000
pdb1HYS	4,344,765	72,834	8.541	0.998	9.658	0.006	0.006	0.010	0.502	0.502	0.502
fl2010	2,346,294	968,962	6.620	0.640	13.31	0.002	0.002	0.003	1.0*	1.0*	1.0*
in2010	1,281,716	534,142	13.03	1.152	15.21	0.002	0.003	0.004	1.0*	1.0*	1.0*
ok2010	1,274,148	538,236	14.28	1.259	16.78	0.002	0.002	0.003	1.0*	1.0*	1.0*
Domain: Linear Programming											
spal	46,168,124	331,899	1.156	0.119	5.961	0.005	0.005	0.007	0.634	0.634	0.634
rail4284	11,284,032	1,101,178	2.113	0.217	5.291	0.007	0.008	0.130	1.000	1.000	0.998
degme	8,127,528	844,916	2.080	0.202	5.723	0.005	0.005	0.060	1.0*	1.0*	1.0*
gupta1	2,164,210	63,604	14.37	1.567	9.737	0.008	0.008	0.232	1.0*	1.0*	0.997
pds-100	1,096,002	670,820	17.43	1.750	14.71	0.001	0.002	0.008	1.0*	1.0*	1.0*
Domain: Mathematical Optimization											
largebasis	5,560,100	880,040	2.805	0.288	8.462	0.007	0.007	0.005	0.783	0.783	0.783
exdata_1	2,269,501	12,002	12.83	1.220	7.453	0.004	0.004	0.020	0.482	0.482	0.482
Domain: Model Reduction Problem											
boneS10	55,468,422	1,829,796	0.725	0.088	8.173	0.007	0.007	0.002	0.671	0.671	0.671
Domain: Optimization Problem											
mip1	10,352,819	132,926	2.069	0.182	5.229	0.006	0.007	0.054	0.577	0.577	0.577
Domain: Power Network											
TSOPF_RS_b2383	16,171,169	76,240	2.386	0.198	5.826	0.004	0.005	0.007	0.434	0.434	0.434
kkt_power	14,612,663	4,126,988	0.768	0.077	7.536	0.004	0.004	0.003	1.000	1.000	1.000
Domain: Structural											
af_shell10	52,672,325	3,016,130	0.658	0.077	9.323	0.006	0.007	0.002	0.824	0.824	0.824
ldoor	46,522,475	1,904,406	1.119	0.110	12.68	0.008	0.008	0.005	1.0*	1.0*	1.0*
Emilia_923	41,005,206	1,846,272	0.704	0.094	7.590	0.006	0.006	0.003	0.564	0.564	0.564
inline_1	36,816,342	1,007,424	1.450	0.137	11.24	0.007	0.006	0.005	0.825	0.825	0.825
F1	26,837,113	687,582	0.851	0.112	5.029	0.007	0.007	0.006	0.462	0.462	0.462
af_shell9	17,588,875	1,009,710	1.375	0.130	7.791	0.007	0.007	0.004	0.600	0.600	0.600
halfb	12,387,821	449,234	1.724	0.154	6.272	0.007	0.007	0.010	0.504	0.504	0.504
troll	11,985,111	426,906	2.030	0.252	6.788	0.006	0.006	0.009	0.463	0.463	0.463
pwtk	11,634,424	435,836	1.808	0.162	6.266	0.007	0.007	0.006	0.494	0.494	0.494
fcondp2	11,294,316	403,644	1.814	0.164	6.095	0.005	0.006	0.008	0.433	0.433	0.433
crankseg_1	10,614,210	105,608	2.343	0.220	5.866	0.008	0.008	0.019	0.803	0.803	0.803
m_t1	9,753,570	195,156	2.184	0.212	5.650	0.006	0.006	0.012	0.304	0.304	0.304
gearbox	9,080,404	307,492	2.802	0.397	7.425	0.007	0.007	0.010	0.421	0.421	0.421
bmw7st_1	7,339,667	282,694	2.682	0.264	6.364	0.008	0.007	0.014	0.477	0.477	0.477
ship_001	4,644,230	69,840	5.949	0.531	7.651	0.008	0.008	0.023	0.406	0.406	0.406
s3dkt3m2	3,753,461	180,898	11.67	1.078	15.14	0.007	0.007	0.010	0.526	0.526	0.526
ct20stif	2,698,463	104,658	26.75	2.664	23.79	0.008	0.008	0.024	0.620	0.620	0.620
nasasrb	2,677,324	109,740	21.77	2.182	18.93	0.005	0.005	0.019	0.403	0.403	0.403
Domain: Synthetic											
pathological_PHIL	14,499,856	240,000	1.860	0.217	5.562	0.006	0.006	0.006	0.372	0.372	0.372
pathological_OSKI	6,999,994	2,000,000	0.982	0.134	2.827	0.005	0.005	1.800	0.776	0.776	0.776

Figure 3

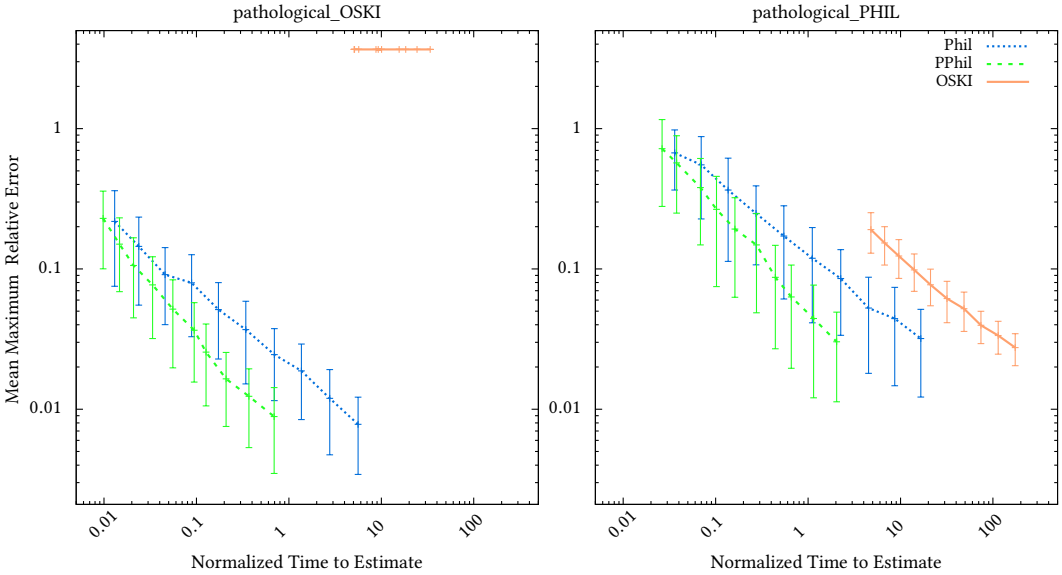
(d) We compare the fill estimation capabilities of PHIL, PPHIL (a parallel version of PHIL), and OSKI on our **"Haswell"** system with a maximum considered block size of $\mathbf{B = 12}$. The parameters to PHIL and PPHIL are $\epsilon = 3$ and $\delta = 0.01$. The parameters to OSKI are $\sigma = 0.02$ (the recommended setting). Highlighted cells show the better result between PHIL, PPHIL, and OSKI. * Results with an asterisk are cases where a slowdown was observed when the performance model was used with the given estimates. Since an autotuner may choose to use CSR if no speedup is observed with the new block size, these results are listed as 1.0.

Matrix Information			Normalized Time to Estimate Fill			Mean Maximum Relative Error			Normalized SpMV Time ([15] Model)		
Name	k (#nonzeros)	Size (#rows + #cols)	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI	PHIL	PPHIL	OSKI
Domain: 2D/3D Problem											
nd24k	28,715,634	144,000	8.433	0.938	112.4	0.031	0.031	0.016	1.0*	1.0*	1.0*
BenElechi1	13,150,496	491,748	5.001	0.495	48.62	0.022	0.023	0.011	0.570	0.570	0.570
kim2	11,330,020	913,952	5.166	0.499	54.42	0.034	0.034	0.006	0.627	0.627	0.627
nd6k	6,897,316	36,000	12.51	1.388	55.28	0.032	0.031	0.027	0.590	0.590	0.590
nd3k	3,279,690	18,000	80.60	9.004	172.0	0.031	0.030	0.039	1.0*	1.0*	1.0*
Domain: Computational Fluid Dynamics											
atmosmodl	10,319,760	2,979,504	4.242	0.409	63.43	0.023	0.023	0.007	1.0*	1.0*	1.0*
3dtube	3,213,618	90,660	64.44	6.809	168.0	0.024	0.022	0.073	0.485	0.485	0.485
Domain: Computer Vision											
bundle_adj	20,208,051	1,026,702	1.943	0.232	23.80	0.025	0.025	0.087	0.779	0.779	0.779
Domain: Electromagnetics											
fem_hifreq_circuit	20,239,237	982,200	3.402	0.384	55.35	0.015	0.015	0.014	0.605	0.605	0.605
Domain: Graph											
hugetric-00010	19,771,708	13,185,530	1.282	0.127	64.33	0.009	0.009	0.005	1.000	1.000	1.000
kron_g500-logn17	10,228,360	262,144	3.973	0.459	39.98	0.004	0.004	0.044	1.000	1.000	1.000
flickr	9,837,214	1,641,756	1.002	0.097	11.23	0.006	0.006	0.040	1.000	1.000	1.000
pdb1HYS	4,344,765	72,834	32.18	2.968	95.77	0.024	0.023	0.041	0.601	0.601	0.601
fl2010	2,346,294	968,962	28.64	2.627	84.26	0.006	0.006	0.009	1.0*	1.0*	1.0*
in2010	1,281,716	534,142	60.05	5.496	100.2	0.007	0.007	0.015	1.000	1.000	1.000
ok2010	1,274,148	538,236	59.06	5.399	96.17	0.006	0.006	0.012	1.000	1.000	1.000
Domain: Linear Programming											
spal	46,168,124	331,899	2.991	0.324	60.44	0.015	0.015	0.025	0.717	0.717	0.717
rail4284	11,284,032	1,101,178	5.666	0.583	49.33	0.017	0.017	0.359	0.930	0.930	0.934
degme	8,127,528	844,916	7.401	0.810	54.22	0.016	0.017	0.076	1.0*	1.0*	0.998
gupta1	2,164,210	63,604	49.63	5.254	86.09	0.023	0.024	0.510	1.0*	1.0*	1.0*
pds-100	1,096,002	670,820	68.77	6.304	90.79	0.004	0.003	0.027	1.000	1.000	1.000
Domain: Mathematical Optimization											
largebasis	5,560,100	880,040	10.12	0.923	54.76	0.027	0.024	0.015	0.612	0.612	0.612
exdata_1	2,269,501	12,002	58.49	6.470	88.21	0.033	0.031	5.051	0.633	0.633	0.610
Domain: Model Reduction Problem											
boneS10	55,468,422	1,829,796	1.720	0.180	65.37	0.027	0.027	0.009	0.635	0.635	0.635
Domain: Optimization Problem											
mip1	10,352,819	132,926	6.075	0.683	40.67	0.032	0.032	0.315	0.556	0.556	0.551
Domain: Power Network											
TSOPF_RS_b2383	16,171,169	76,240	5.577	0.614	47.80	0.040	0.036	0.075	0.453	0.453	0.453
kkt_power	14,612,663	4,126,988	2.452	0.233	47.12	0.008	0.008	0.013	1.000	1.000	1.000
Domain: Structural											
af_shell10	52,672,325	3,016,130	1.478	0.170	60.91	0.025	0.025	0.004	0.614	0.614	0.614
ldoor	46,522,475	1,904,406	1.950	0.215	64.42	0.024	0.024	0.012	0.551	0.551	0.551
Emilia_923	41,005,206	1,846,272	1.977	0.194	60.87	0.021	0.021	0.010	0.652	0.652	0.652
inline_1	36,816,342	1,007,424	2.368	0.235	58.50	0.018	0.019	0.014	0.807	0.807	0.807
F1	26,837,113	687,582	2.712	0.341	48.82	0.019	0.019	0.019	0.708	0.708	0.708
af_shell9	17,588,875	1,009,710	3.764	0.431	51.25	0.025	0.025	0.007	0.445	0.445	0.445
halfb	12,387,821	449,234	11.66	1.343	103.9	0.027	0.026	0.030	0.831	0.831	0.831
troll	11,985,111	426,906	6.259	0.648	52.39	0.024	0.024	0.032	0.532	0.532	0.532
pwtk	11,634,424	435,836	6.516	0.734	57.14	0.034	0.034	0.018	0.680	0.677	0.695
fcondp2	11,294,316	403,644	6.187	0.571	49.22	0.022	0.023	0.029	0.488	0.488	0.488
crankseg_1	10,614,210	105,608	15.45	1.513	117.8	0.024	0.024	0.050	1.0*	1.0*	1.0*
m_t1	9,753,570	195,156	7.774	0.723	54.87	0.020	0.019	0.039	0.849	0.849	0.849
gearbox	9,080,404	307,492	8.115	0.807	51.59	0.022	0.022	0.037	0.540	0.540	0.540
bmw7st_1	7,339,667	282,694	10.53	1.143	58.06	0.026	0.026	0.039	0.554	0.554	0.554
ship_001	4,644,230	69,840	31.22	2.913	111.4	0.029	0.029	0.064	0.689	0.689	0.689
s3dkt3m2	3,753,461	180,898	34.77	4.044	109.0	0.032	0.034	0.021	0.459	0.459	0.459
ct20stif	2,698,463	104,658	64.89	5.899	130.3	0.025	0.026	0.065	0.416	0.416	0.416
nasasrb	2,677,324	109,740	102.3	9.649	211.1	0.019	0.019	0.044	0.609	0.609	0.609
Domain: Synthetic											
pathological_PHIL	14,499,856	240,000	5.697	0.685	57.57	0.043	0.048	0.042	0.440	0.440	0.440
pathological_OSKI	6,999,994	2,000,000	2.388	0.270	18.17	0.012	0.012	3.666	0.704	0.704	0.714

Figure 4



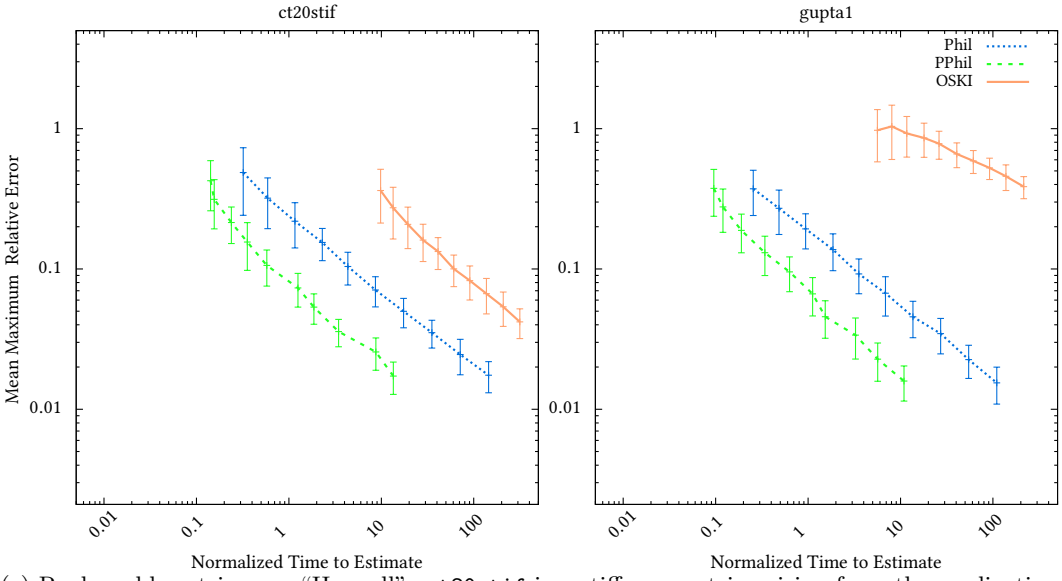
(a) Real world matrices on "Ivy Bridge". `ct20stif` is a stiffness matrix arising from the application of finite element analysis to an engine block. It has a mix of mostly 3×3 and 6×6 dense blocks. `gupta1` is the matrix representation of a linear programming problem, and has no obvious block structure.



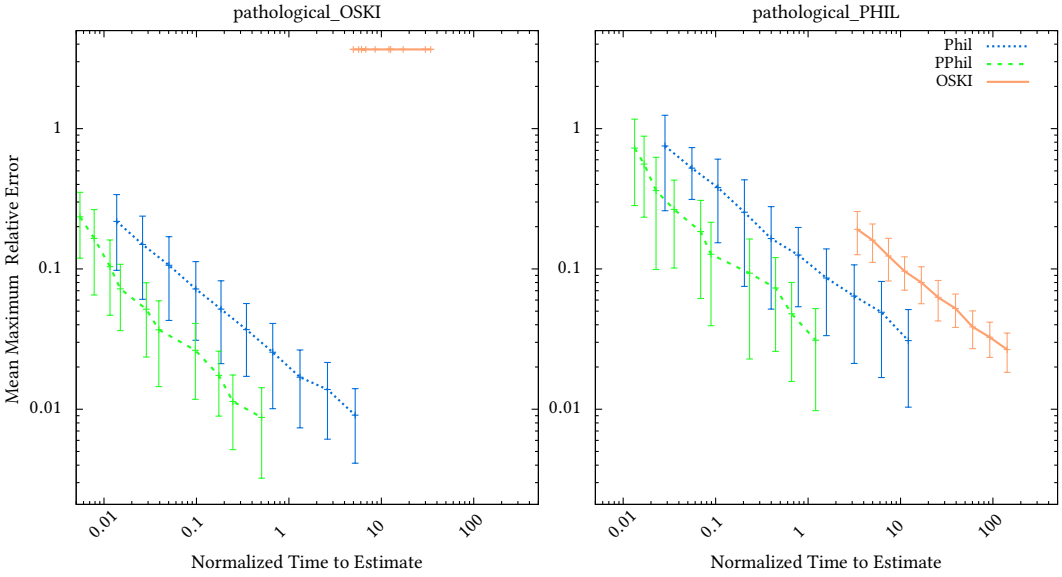
(b) Pathological cases on "Ivy Bridge". These matrices are described in more detail in Section 6.2.

Figure 4: Mean maximum relative error (Definition 6.1) 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 [24]) for four matrices on **"Ivy Bridge"** with a maximum block size of $\mathbf{B = 12}$. 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. Each point is a separate setting for the parameters where ϵ varies exponentially from 50 to 2 and σ ranges exponentially from 0.001 to 0.05. Note that errors above 1 represent a complete loss of accuracy.

Figure 4



(c) Real world matrices on "Haswell". `ct20stif` is a stiffness matrix arising from the application of finite element analysis to an engine block. It has a mix of mostly 3×3 and 6×6 dense blocks. `gupta1` is the matrix representation of a linear programming problem, and has no obvious block structure.



(d) Pathological cases on "Haswell". These matrices are described in more detail in Section 6.2.

Figure 4: Mean maximum relative error (Definition 6.1) 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 [24]) for four matrices on **"Haswell"** with a maximum block size of $\mathbf{B = 12}$. 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. Each point is a separate setting for the parameters where ϵ varies exponentially from 50 to 2 and σ ranges exponentially from 0.001 to 0.05. Note that errors above 1 represent a complete loss of accuracy.

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 [34]. 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.

7.1 Future Work

Future work includes extensions to handle sparse tensors of arbitrary order in multiple storage formats. Software complexity is the limiting factor in the development of such an implementation.

7.2 Extensions

Some formats store their blocks in a sparse format [9, 12]. These blocks are usually much larger than the blocks mentioned in this paper, but we can extend an algorithm for Problem 2.1 to estimate the fill of larger block sizes by limiting our attention to multiples of some base block size.

Problem 7.1 (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 2.1 on $\mathcal{A}'$ is a solution to Problem 7.1 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 2.1 can solve Problem 7.1 with an addition of $O(k(\mathcal{A}))$ to their asymptotic running time.

References

[1] W. Ahrens, H. Xu, and N. Schiefer, "A Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 546–556.

[2] H. Xu, "Fill Estimation for Blocked Sparse Matrices and Tensors," Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Jun. 2018.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] E.-J. Im and K. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Computational Science — ICCS 2001*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, May 2001, pp. 127–136.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] D. M. Kane, J. Nelson, and D. P. Woodruff, "An Optimal Algorithm for the Distinct Elements Problem," in *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '10. New York, NY, USA: ACM, 2010, pp. 41–52.

[30] A. Metwally, D. Agrawal, and A. E. Abbadi, "Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic." ACM Press, 2008, p. 618.

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

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

[33] P. Sanders, S. Lamm, L. Hübschle-Schneider, E. Schrade, and C. Dachsbacher, "Efficient Parallel Random Sampling–Vectorized, Cache-Efficient, and Online," *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 29:1–29:14, Jan. 2018.

[34] 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.