Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2017-011 June 9, 2017

An Efficient Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats

Peter Ahrens, Nicholas Schiefer, and Helen Xu

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

An Efficient Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats

Peter Ahrens
pahrens@mit.edu

Nicholas Schiefer
schiefer@mit.edu

Helen Xu
hjxu@mit.edu

Abstract

Tensors, which are the linear-algebraic extensions of matrices in arbitrary dimensions, have numerous applications to data processing tasks in computer science and computational science. Many tensors used in diverse application domains are sparse, typically containing more than 90% zero entries. Efficient computation with sparse tensors hinges on algorithms that can leverage the sparsity to do less work, but the irregular locations of the nonzero entries pose significant challenges to performance engineers. Many tensor operations such as tensor-vector multiplications can be sped up substantially by breaking the tensor into equally sized blocks (only storing blocks which contain nonzeros) and performing operations in each block using carefully tuned code. However, selecting the best block size from among many possibilities is computationally challenging.

Previously, Vuduc et al. defined the *fill* of a sparse tensor to be the number of stored entries in the blocked format divided by the number of nonzero entries, and showed how the fill can be used as part of an effective, efficient heuristic for evaluating the quality of a particular blocking scheme [1, 2]. In particular, they showed that if the fill could be computed exactly, then the measured performance of their sparse matrix-vector multiply was within 5% of the optimal setting. However, they gave no theoretical accuracy bounds for their method for estimating the fill, and it is vulnerable to several classes of adversarial examples.

In this paper, we present a sampling-based method for finding a $(1+\varepsilon)$-approximation to the fill of an order N tensor for all block sizes less than B, with probability at least $1-\delta$, using $O(NB^N \log(B/\delta)/\varepsilon^2)$ samples for each block size. We introduce an efficient routine to sample for all B^N block sizes at once in $O(NB^N)$ time. We extend our concentration bounds to a more efficient bound based on sampling without replacement, using the recent Hoeffding-Serfling inequality [3]. We then implement[1] our algorithm and evaluate it on sparse matrices from the University of Florida collection and compare our scheme to that of Vuduc, as implemented in the Optimized Sparse Kernel Interface (OSKI) library, and a brute-force method for obtaining the ground truth. We find that our algorithm provides faster estimates of the fill at all accuracy levels, providing evidence that this is both a theoretical and practical improvement.

1 Introduction

Tensors are multi-dimensional generalizations of matrices. They have important applications in the physical and computational sciences, ranging from machine learning to computational chemistry. Many tensors used in diverse application domains are sparse, typically containing more than 90% zero entries. Most fundamental linear algebraic operations on tensors, such as tensor-vector multiplications, run in time proportional to the number of elements in the tensors. Sparse tensors provide an opportunity to write algorithms and data structures with complexity proportional to the number of nonzero entries, with substantial increases in performance. However, the increased complexity of data structures that can describe the irregular locations of nonzeros in these tensors poses a significant challenge to algorithm designers and performance engineers.

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

These challenges will only grow as architectures become increasingly specialized. In order to write the most efficient sparse tensor code, the programmer must take into account both the target architecture and the relevant structural properties of the nonzeros of the sparse tensor. Writing custom code for each processor requires extensive engineering effort. Additionally, the structure of nonzeros in a sparse tensor is usually known only at runtime. As a result, *autotuning* (automatically generating customized code) has become a necessary part of writing efficient code for operations on sparse tensors.

Sparse Tensor Representations. Previous efforts in autotuning for sparse tensors focus on sparse matrices, which see the broadest application. The diverse space of operations and nonzero patterns of sparse matrices have led to the development of a wide variety of sparse matrix formats that allow programmers to more efficiently operate on the matrices. Perhaps the most popular such format is the *Compressed Sparse Row (CSR)* [4]. Like most sparse matrix formats, CSR stores only the locations and values of the nonzero entries of the matrix; the specific details of the format are not relevant to the present exposition and are omitted. Note that the results in this paper apply to any sparse matrix format which can be generalized to include a block structure.

To decrease the complexity of storing the locations of individual nonzeros, performance engineers have developed a variant of CSR called *Blocked Compressed Sparse Row (BCSR)* [5]. In BCSR, an $m \times n$ matrix is partitioned into $m/r \times n/c$ submatrices, where each submatrix is of size $r \times c$. The submatrices are called blocks, and are stored in a dense format, with zeros represented explicitly. Only blocks which contain nonzeros are stored, and the locations of the stored blocks are recorded using CSR format.

The key advantage of the BCSR format (and blocked formats in general) is the reduced complexity of operations on the "dense" part of the matrices. For example, consider the common operation of a matrix-vector multiplication. If a matrix has a natural block structure, then the blocked matrix can be multiplied by a blocked vector using standard CSR methods, but the individual blocks can be multiplied using a small, fixed-size, dense matrix-vector multiply. Performance engineers have experience in writing efficient code for small dense linear algebra kernels. The programmer and compiler can utilize standard techniques like loop unrolling, register and cache blocking, and instruction-level parallelism.

Matrices with a natural block structure appear in numerous applications, such as matrices produced by finite element methods. The BCSR format can be extended naturally to support higher-dimensional tensors as well [6, 7, 8].

Preliminaries. Throughout this paper, we will discuss N-dimensional tensors in a particular orthogonal basis. That is, tensors are N-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 an order N tensor $\mathcal{A}$ indexed by $(i_1, i_2, \ldots, i_N)$ is denoted $\mathcal{A}[i_1, i_2, ..., i_N]$. For compactness of notation, we sometimes specify an index into a tensor as a N-component vector $\mathbf{i} = (i_1, i_2, \ldots, i_N)$. If we wish to represent the integer range $i, i+1, ..., i'$, we use the syntax $i \rightarrow i'$. If we wish to represent the range of indices between two vectors, we use the syntax $\mathbf{i} \rightarrow \mathbf{i}'$, meaning $i_1 \rightarrow i'_1, ..., i_N \rightarrow i'_N$. Subtensors are formed when we fix a subset of indices. We use a colon 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 $\mathcal{A}[:, n/4 \rightarrow 3n/4]$.

The number of nonzero entries in a $\mathcal{A}$ is denoted $k(\mathcal{A})$. When we compare a vector to a scalar, we mean to compare each entry of the vector pointwise. For notational convenience, we occasionally redefine the starting index of a tensor. Thus, $\mathcal{A} \in \mathbb{F}^{\mathbf{I} \rightarrow \mathbf{I}'}$ is an $(\mathbf{I}'_1 - \mathbf{I}_1 + 1) \times ... \times (\mathbf{I}'_N - \mathbf{I}_N + 1)$ tensor whose smallest index is $\mathbf{I}$ and largest index is $\mathbf{I}'$.

1.1 The problem of selecting a block size

Since the performance of blocked sparse tensor operations depends on the block size, we must find some block size that gives good—and ideally the best—performance on the matrix. If we

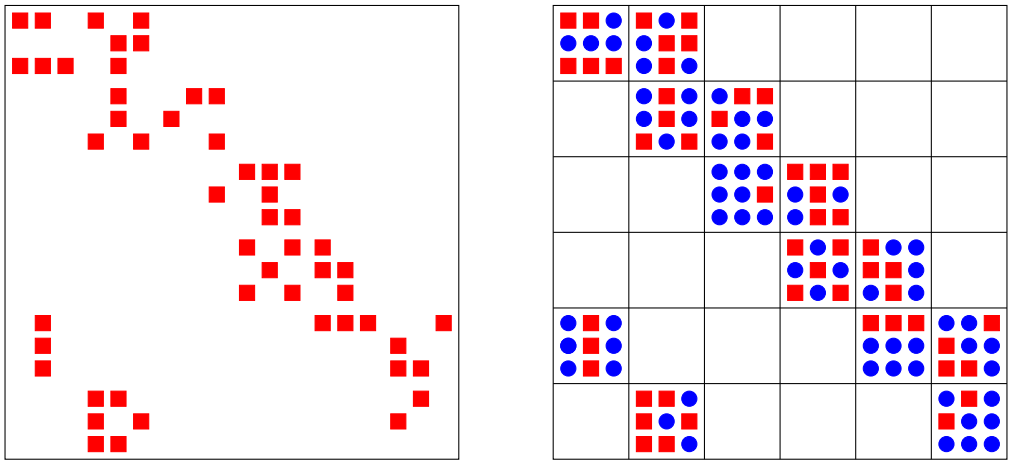

Figure 1.1 – A sparse matrix before blocking (left) and an example of a blocked sparse matrix (right). The squares denote nonzero elements and circles are explicit zeros that are introduced due to the blocking scheme. In this example, the blocking scheme is defined by $\mathbf{b} = (3, 3)$ resulting in $k_{\mathbf{b}} = 13$. The number of nonzero elements $k(\mathcal{A}) = 52$, so we can compute the *fill* as follows: $f_{\mathbf{b}} = (3 \times 3 \times 13)/52 = 2.25$.

set the block size too small, then we must store the locations of more blocks, and we waste time processing the block locations. If we set the block size too large, then the blocks will be filled with too many zeros, and we waste time computing unnecessary dense matrix operations.

Definition 1.1. A *blocking scheme* $\mathbf{b}$ of a tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \cdots \times I_n}$ is parameterized by a vector $\mathbf{b} = (b_1, b_2, \ldots, b_N)$ of block sizes. The blocking scheme induced by $\mathbf{b}$ is a partition of $\mathcal{A}$ into N-dimensional subtensors with b_i entries along the i^{th} dimension. Thus, a nonzero at location $\mathbf{i}$ would be stored at the block index

$$\left(\left\lceil \frac{i_1}{b_1} \right\rceil, \left\lceil \frac{i_2}{b_2} \right\rceil, ..., \left\lceil \frac{i_N}{b_N} \right\rceil \right).$$

We present an example of a blocking scheme in a sparse matrix in Figure 1.1. Blocked formats fill in the empty slots of nonempty blocks with explicit zeroes and do not fill in empty blocks.

When manipulating a sparse tensor $\mathcal{A}$, we want to find a blocking scheme that includes all of the nonzero entries of $\mathcal{A}$ in few blocks. We are therefore 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.

Intuitively, a blocking scheme is good if it packs all of the nonzeros into a small number of non-empty blocks. We define the *fill*, which captures this notion of blocking scheme quality:

Definition 1.2 (Vuduc et al. [2])**.** The *fill* of a tensor $\mathcal{A}$ with respect to a particular blocking scheme $\mathbf{b}$ is the ratio

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

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 what tensor we are referring to, we often write the fill as $f_{\mathbf{b}}$.

In their seminal work, Vuduc et al. demonstrate that the fill can be used as an effective heuristic for predicting the performance of a particular block size on a sparse matrix $\mathcal{A}$. They showed that when the fill was known exactly, performance of the resulting blocking scheme was optimal or near-optimal (within 5%) on all of the platforms that they tested [2]. Once per machine, we compute

a profile of how the machine performs for a particular block size. Let $P_{\mathbf{b}}$ be the performance of the machine (in MFLOP/s) on a dense matrix stored with blocking scheme $\mathbf{b}$. We can think of $P_{\mathbf{b}}$ as 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 $P_{\mathbf{b}}/f_{\mathbf{b}}(\mathcal{A})$.

Unfortunately, the fill of a tensor with respect to a blocking scheme can vary substantially depending on the tensor's structure. The blocking scheme that minimizes the fill of a tensor can be found by searching over all possible blocking schemes and computing the number of nonempty blocks for each. However, it is computationally intractable in practice to calculate the fill exactly, since we would spend far more time estimating the fill to find the optimal blocking scheme than we would performing the tensor operations directly [9, 2].

In practice, we care only about block sizes that are small enough to fit in most L1 caches, which is typically at most 12 entries along both dimensions for matrices [9]. Furthermore, the cost of calculating the fill rivals the performance benefits obtained from blocking. Thus, our problem is to quickly compute an approximation to the fill with reasonable accuracy:

Problem 1.1 (Fill approximation problem)**.** Given a tensor $\mathcal{A}$ and a maximum block size B, compute a (randomized) approximation $F_{\mathbf{b}}(\mathcal{A})$ such that

$$(1-\varepsilon)f_{\mathbf{b}}(\mathcal{A}) \le F_{\mathbf{b}}(\mathcal{A}) \le (1+\varepsilon)f_{\mathbf{b}}(\mathcal{A})$$

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

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

Previously, Vuduc et al. gave sampling methods for estimating the fill of a sparse matrix, but did not give any theoretical analysis of the accuracy of their method. Furthermore, their method takes as long as 1 to 10 times the time it takes to perform a sparse matrix-vector multiplication on the same matrix [9].

1.2 Our contributions

We describe the first algorithm which solves the fill approximation problem with provable guarantees, and demonstrate that it is faster and more accurate on a suite of sparse matrices than the state-of-the-art algorithm due to Vuduc [2]. At a high level, our algorithm repeatedly samples a nonzero entry in the tensor, then computes the number of nonzero elements in the block of that entry, for all possible blocking schemes. For each blocking scheme, it then averages the reciprocal of this count over all sampled indices. We provide a more detailed description of this sampling algorithm in Section 3. We show that this is an unbiased estimator for the fill of the tensor, and give tight concentration bounds.

More formally, suppose that our algorithm is given a maximum block size B and a sparse tensor $\mathcal{A}$. Our algorithm repeatedly samples a location $\mathbf{i}$ from a tensor $\mathcal{A}$. For each blocking scheme $\mathbf{b} \le B$, it computes the number $z_{\mathbf{b}}(\mathbf{i})$ of nonzero entries in *the block that* $\mathbf{i}$ *appears in* under the blocking scheme $\mathbf{b}$. After drawing a total of S samples $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$, it computes the averages

$$F_{\mathbf{b}} := \frac{b_1 b_2 \cdots b_N}{S} \sum_{j=1}^{S} \frac{1}{z_{\mathbf{b}}(\mathbf{i}_j)}$$

for all $\mathbf{b} \le B$.

Our algorithm extends the concept of fill from sparse matrices to general sparse tensors in a natural way. We also contribute a fast, tightly optimized method for computing $z_{\mathbf{b}}(\mathbf{i})$ for all blocking schemes $\mathbf{b}$ at the same time using multi-dimensional prefix sums (cumulative sums).

1.2.1 Theoretical contributions

We provide the first rigorous analysis of an algorithm for approximating the fill of an N-dimensional tensor under a particular blocking scheme. We provide a full analysis of our sampling method in Section 4.

First, we show that our sampling-based algorithm is indeed an unbiased estimator for the fill $f_{\mathbf{b}}$ (with expectation equal to the fill):

Theorem 1.1. *For any blocking scheme* $\mathbf{b}$*, the random variable* $F_{\mathbf{b}}$ *is an unbiased estimator for the fill: that is,* $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}(\mathcal{A})$.

An unbiased estimator is useful only if it has reasonable concentration about its mean. Prior work on fill estimation gave *no* theoretical analysis of the concentration of the estimator, in part because the heuristic is not conducive to theoretical analysis. In fact, prior methods suffer from vastly worse performance on certain adversarial examples, as we show experimentally in Section 5.

In contrast, we give two concentration bounds for $F_{\mathbf{b}}$, showing that our algorithm solves the fill approximation problem so long as we use enough samples. If we sample the nonzeros with replacement, an analysis based on Hoeffding's inequality gives that:

Theorem 1.2. *If we sample at least* $\frac{NB^N}{2\varepsilon^2}\log\left(\frac{2B}{\delta}\right)$ *samples with replacement, then*

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

Notice that for constant δ, this bound is independent of the number of nonzeros $k(\mathcal{A})$, whereas the algorithm introduced by Vuduc et al. depends linearly upon the number of nonzeros [2]. Because it runs in constant time with respect to the number of nonzeros, the bound on S could (and does, for realistic settings of ε and δ) exceed the number of nonzeros. This is fundamental to bounds based on Hoeffding's inequality, and methods based on sampling-with-replacement in general.

To avoid this issue, we consider the case where we sample *without replacement*. This analysis is more involved, since the sampled locations are no longer independent and so we cannot use Hoeffding's inequality. Instead, we use the recent Hoeffding-Serfling inequality, due to Bardenet and Maillard, obtaining a strictly tighter bound, which is also at most the number of nonzeros [3].

Theorem 1.3. *Let* $T = \frac{NB^N}{2\varepsilon^2}\log\left(\frac{2B}{\delta}\right)$*. If*

$$S \geq \frac{T - T/k(\mathcal{A}) + \sqrt{(T - T/k(\mathcal{A}))^2 + 4T(1 + T/k(\mathcal{A}))}}{2 + 2T/k(\mathcal{A})}\,,$$

then

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

1.2.2 Experimental contributions

We implemented[2] the sampling algorithm described in Section 3 for sparse CSR matrices in C and compared it against the existing algorithm for fill estimation proposed by Vuduc et al. [9] using the same test matrices. We also examine the performance on pathological inputs for each algorithm. We show that our algorithm approximates the fill more accurately and quickly than the existing method and present our findings in 5.

Finally, we note that estimating the fill can be an important part of any sparse data structure which uses blocking, not just BCSR. In fact, any sparse data structure can be adapted to a blocked regime by grouping a tensor into blocks and simply treating nonzero blocks as nonzeros of some sparse tensor.

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

2 Related Work

Fill estimation is an important intermediate step in autotuning blocked sparse matrix computations [10, 1, 11, 12, 13]. Previous work also includes autotuning for matrix computations on GPUs [14] and performance tuning for sparse matrix kernels [15, 16].

To our knowledge, there are no theoretical guarantees on the accuracy of existing algorithms for fill estimation in matrices. [9] provides an empirical study.

2.1 OSKI Heuristic for fill estimation

We first describe the existing heuristic for fill estimation. Since the algorithm is implemented in the Optimized Sparse Kernel Interface (OSKI) library, we will refer to it as `OSKI` [2]. `OSKI` samples from the nonzero structure using some user-chosen parameter $\sigma \in [0, 1]$ that adjusts the runtime and accuracy of the algorithm.

Most of the work in `OSKI` is accomplished in the EvaluateRows subroutine. Let B be the maximum number of rows or columns in a block — that is, the maximum block size is $B \times B$ (recall that for matrices, a typical setting of B is 12). EvaluateRows computes an estimate of the fill for a fixed r and all $1 \leq c \leq B$. Let the input matrix $\mathcal{A}$ have dimensions $m \times n$. We define the i-th *block row* to be the rows ir through $(i+1)r-1$ ($\mathcal{A}[ir \to (i+1)r-1, :]$). EvaluateRows exactly computes the number of nonzero blocks in an expected fraction σ of the block rows. `OSKI` works by calling EvaluateRows once for each row size $1 \leq r \leq B$.

EvaluateRows evaluates an expected fraction σ of block rows by evaluating each one with probability σ. For each block row that EvaluateRows chooses to evaluate, it uses B arrays of length n (this construct is referred to as `blocks_visited`) to store the number of nonzeros seen so far in each block as it iterates over the nonzeros of $\mathcal{A}[ir \to (i+1)r-1, :]$ in row-major order. Each time a nonzero is seen in a previously unvisited block of size $r \times c$, the estimate of the number of blocks of size $r \times c$ (stored in an array as $\texttt{nnz\_visited}[c]$) is incremented.

For each blocking scheme (r, c), the fill estimate is defined by

$$F_{r,c}(A, \sigma) = \frac{rc(\texttt{nnz\_visited}_r[c])}{\sigma k(\mathcal{A})}$$

We provide pseudocode for EvaluateRows in Algorithm 2.1. The function takes as input a tensor $\mathcal{A}$, maximum block dimension B, evaluation probability σ, and a fixed row dimension r. The algorithm estimates the fill for all blocking schemes with row dimension r. To compute fill estimates for all blocking schemes, we call the function once for each possible row dimension r.

Algorithm 2.1.

```
1:  function EvaluateRows(A, B, σ, r)
2:
3:      blocks_visited_c ∈ ℕ^n ∀1 ≤ c ≤ B
4:      nnz_visited ∈ ℕ^B
5:      blocks_visited_c ← 0 ∀1 ≤ c ≤ B
6:      nnz_visited ← 0
7:      for i ∈ 1 → m/r do
8:          Flip a coin with heads probability σ.
9:          if heads then
10:             for nonzero column indices j in A[s,t] ∈ A[ir → (i+1)r−1, :] do
11:                 for c ∈ 1 → B do
12:                     nnz_visited_r ← nnz_visited_r + 1
13:                     blocks_visited_c[⌊j/c⌋] ← blocks_visited_c[⌊j/c⌋] + 1
14:                     if blocks_visited_c[⌊j/c⌋] ← 1 then
15:                         nnz_visited[c] ← nnz_visited[c] + 1
16:             Loop through the nonzeros again and zero out blocks_visited.
```

Theorem 2.1. *OSKI takes time $\Omega(\sigma B^2 k(\mathcal{A}))$ in expectation.*

Proof. First, notice that the algorithm EvaluateRows takes $\Omega(\sigma B k(\mathcal{A}))$ time in expectation.

Since each block row is evaluated with probability σ, each nonzero in the matrix is evaluated with probability σ and since the algorithm performs at least B operations for each nonzero evaluated, the algorithm takes $\Omega(\sigma B k(\mathcal{A}))$ in expectation.

Since OSKI must call EvaluateRows B times (once for each row size $1 \leq r \leq B$), the proof is complete. □

3 The Algorithm

For notational convenience, we introduce a few important definitions for working with blocking schemes on tensors:

Definition 3.1. The *head* of a block is the unique element in the block with the lowest index along all dimensions. For any index $\mathbf{i}$, let $h_{\mathbf{b}}(\mathbf{i})$ denote the index of the head of $\mathbf{i}$'s block under the blocking scheme $\mathbf{b}$. Similarly, the *tail* of a block is the unique element in the block with the highest index along all dimensions. For any index $\mathbf{i}$, let $t_{\mathbf{b}}(\mathbf{i})$ denote the index of the tail of $\mathbf{i}$'s block under $\mathbf{b}$.

Our algorithm works by repeatedly sampling a nonzero entry of the tensor, computing a value associated with that entry for all blocking schemes $\mathbf{b}$, and then averaging over the samples for each blocking scheme. The function that we compute is $x_{\mathbf{b}}$, defined on each index $\mathbf{i}$ of a nonzero of $\mathcal{A}$ and given by:

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

where $z_{\mathbf{b}}(\mathbf{i})$ 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.

More formally, we begin by drawing a total of S samples $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$ from the set of nonzero indices in $\mathcal{A}$. We then compute the estimates

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

for all $\mathbf{b} \leq B$.

As it turns out, the average of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all i is closely related to the fill of the matrix, up to factors that are trivial to compute. The estimate $F_{\mathbf{b}}$ that our algorithm computes is an unbiased estimator for the fill:

Theorem 3.1 (Restatement of Theorem 1.1)**.** *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, so long as the block contains at least one entry. Thus, the sum of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ over all nonzeros $\mathbf{i}$ in $\mathcal{A}$ is equal to the number of blocks that contain nonzeros. Thus, we may multiply our average by $b_1 b_2 ... b_n$ to obtain an estimator of $f_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$, by definition. □

Consider the population $\chi_{\mathbf{b}}(\mathcal{A}) = (x_{\mathbf{b}}(\mathcal{A}, \mathbf{i}) | \mathcal{A}[\mathbf{i}] \neq 0)$. We have just shown that the average value of elements in $\chi_{\mathbf{b}}(\mathcal{A})$ is $k_{\mathbf{b}}(\mathcal{A})/k(\mathcal{A})$.

Thus, our task is to randomly sample elements from $\chi_{\mathbf{b}}$ to compute an estimate of its average. We can compute a sample of $\chi_{\mathbf{b}}$ by selecting a nonzero uniformly at random, looking up how many nonzeros are in the block corresponding to this nonzero, and returning the reciprocal. This is a lot of work to do for one sample, especially if the block is very full. However, the computations of $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for the same $\mathbf{i}$ use many of the same computations and data. Once we have the locations of all the nonzeros within a B radius of our nonzero at index $\mathbf{i}$, we can compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all

$\mathbf{b} \leq B$ at the same time using prefix sums (cumulative sums), saving an enormous amount of work. We provide pseudocode for this algorithm, called Compute$\mathcal{X}$, in Algorithm 3.3.

We define the NumSamples function in Algorithm 3.2 and provide analysis of the number of samples necessary in Section 4.

Putting these pieces together, our algorithm is as follows:

Algorithm 3.1. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \ldots \times I_N}$, $\mathbf{i}$, 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:
$0 \leq \delta \leq 1$
$\varepsilon > 0$
$B \geq 1$

function EstimateFill($\mathcal{A}$, B, ε, δ)
  $\mathcal{Y} \in \mathbb{R}^{B \times \ldots \times B}$
  $\mathcal{F} \in \mathbb{R}^{B \times \ldots \times B}$
  $S \leftarrow$ NumSamples$(\mathcal{A}, B, \varepsilon, \delta)$
  $\mathcal{Y} \leftarrow 0$
  for $\mathbf{i} \in$ sample of size S without replacement from the nonzero indices of $\mathcal{A}$ **do**
    $\mathcal{Y} \leftarrow \mathcal{Y} +$ Compute$\mathcal{X}(\mathcal{A}, B, \mathbf{i})$
  for $\mathbf{b} \in 0 \rightarrow B$ **do**
    $\mathcal{F}[\mathbf{b}] \leftarrow \frac{b_1 b_2 \ldots b_n \mathcal{Y}[\mathbf{b}]}{s}$
  return $\mathcal{F}$

Ensure:
$(1-\varepsilon) f_{\mathbf{b}}(\mathcal{A}) \leq \mathcal{F}[\mathbf{b}] \leq (1+\varepsilon) f_{\mathbf{b}}(\mathcal{A})$ with probability at least $(1-\delta)$.

Since we know that Algorithm 3.1 will work if its subroutines work, we have only to explain the functions NumSamples and Compute$\mathcal{X}$.

3.1 NumSamples

We state the NumSamples algorithm here, and leave the analysis for Section 4. The number of samples used here corresponds to the bound according to sampling without replacement.

Algorithm 3.2. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times \ldots \times I_N}$, ε, and δ, compute an estimate of the number of samples necessary to return an (ε, δ) approximation.

Require:
$0 \leq \delta \leq 1$
$\varepsilon > 0$
$B \geq 1$

function EstimateFill($\mathcal{A}$, B, ε, δ)
  $T \leftarrow \frac{NB^N}{2\varepsilon^2} \log\left(\frac{2B}{\delta}\right)$.
  $S \leftarrow \frac{T - T/k(\mathcal{A}) + \sqrt{(T - T/k(\mathcal{A}))^2 + 4T(1 + T/k(\mathcal{A}))}}{2 + 2T/k(\mathcal{A})}$
  return S

Ensure:
S is such that Algorithm 3.1 will return an approximation $\mathcal{F}$ which satisfies $(1-\varepsilon) f_{\mathbf{b}}(\mathcal{A}) \leq \mathcal{F}[\mathbf{b}] \leq (1+\varepsilon) f_{\mathbf{b}}(\mathcal{A})$ with probability at least $(1-\delta)$.

3.2 Compute$\mathcal{X}$

The main idea of Compute$\mathcal{X}$ is to create a tensor $\mathcal{Z}_0$ corresponding to the number of nonzeros of $\mathcal{A}$ in certain subtensors surrounding $\mathbf{i}$. More formally, $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B \rightarrow \mathbf{i}+B-1}$ will be constructed so that $\mathcal{Z}_0[\mathbf{j}]$ is equal to the number of nonzeros in the subtensor $\mathcal{A}[\mathbf{i} - B \rightarrow \mathbf{j}]$. 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 $Z_0[t_{\mathbf{b}}(\mathbf{i})] - Z_0[t_{b_1}(i_1), h_{b_2}(i_2) - 1] - Z_0[h_{b_1}(i_1) - 1, t_{b_2}(i_2)] + Z_0[h_{\mathbf{b}}(\mathbf{i}) - 1]$. Higher dimensions become more complicated, but we will show how to reuse computations to keep things manageable.

First, notice that we can compute $\mathcal{Z}_0$ using prefix sums! 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 taking the first prefix sum, $\mathcal{Z}_0[\mathbf{j}]$ represents the number of nonzeros in $\mathcal{A}[i_1 - B \to j_1, j_2, ..., j_N]$. After taking the n^{th} prefix sum, $\mathcal{Z}[\mathbf{j}]$ represents the number of nonzeros in $\mathcal{A}[i_1 - B \to j_1, ..., i_n - B \to j_n, j_{n+1}, ..., j_N]$. After the N^{th} prefix sum, we have computed $\mathcal{Z}_0$.

Computing the actual values of z once we have $\mathcal{Z}_0$ is slightly trickier. For each value of b_1, we compute $\mathcal{Z}_1[j_2, ..., j_N]$ to be the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) \to t_{b_1}(i_1), i_2 - B \to j_2, ..., i_N - B \to j_N]$ as $\mathcal{Z}_0[t_{b_1}(i_1), j_2, ..., j_N] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, j_2, ..., j_N]$. Once we have Z_1 for a particular value of b_1, then for each value of b_2 we can take differences between elements of Z_1 to compute Z_2, where $Z_2[j_3, ..., j_N]$ is the number of nonzeros in the subtensor $\mathcal{A}[h_{b_1}(i_1) \to t_{b_1}(i_1), h_{b_2}(i_2) \to t_{b_2}(i_2), i_3 - B \to j_3, ..., i_N - B \to j_N]$. Continuing in this way, $\mathcal{Z}_N$ is just the scalar $z_{\mathbf{b}}(\mathbf{j})$.

We provide an example of the prefix sum routine in a sparse matrix in Figure 3.1 and show how to use subtractions to count the number of nonzero entries in Figure 3.2.

To reflect the fact that $\mathcal{A}$ may be stored in an arbitrary sparse format, we abstract the process of finding the indices of nonzeros within a certain range into an algorithm called NONZEROSINRANGE. NONZEROSINRANGE$(\mathcal{A}, \mathbf{j}, \mathbf{j}')$ returns a list of all $\mathbf{i} \in \mathbf{j} \to \mathbf{j}'$ such that $\mathcal{A}[\mathbf{i}] \neq 0$. Efficient implementations of NONZEROSINRANGE will be discussed for various sparse formats in Section 3.2.1

We can now state COMPUTE$\mathcal{X}$.

Algorithm 3.3. Given a sparse tensor $\mathcal{A} \in \mathbb{F}^{I_1 \times I_2 \times ... \times I_N}$, $\mathbf{i}$, and B, compute $x_{\mathbf{b}}(\mathcal{A}, \mathbf{i})$ for all N-dimensional grids $\mathbf{b} \leq B$. Note that $\mathcal{A}$ may be stored in a sparse format, whereas all other tensors are stored in a dense format.

Require:
$\mathcal{A}[\mathbf{i}] \neq 0$
$B \geq 1$

function COMPUTE$\mathcal{X}$($\mathcal{A}$, $\mathbf{i}$, B)
  $\mathcal{Z}_0 \in \mathbb{N}^{\mathbf{i}-B \to \mathbf{i}+B-1}$
  $\mathcal{Z}_0 \leftarrow 0$
  for $\mathbf{j} \in$ NONZEROSINRANGE$(\mathcal{A}, \mathbf{i} - B, \mathbf{i} + B - 1)$ **do**
    $\mathcal{Z}_0[\mathbf{j}] \leftarrow 1$
  for $n \in 1 \to N$ **do**
    for $j \in i_n - B + 1 \to i_n + B - 1$ **do** ▷ Perform prefix sum
      $\mathcal{Z}_0[\underbrace{:, ..., :, j}_{n}, :, ..., :] \leftarrow \mathcal{Z}_0[\underbrace{:, ..., :, j}_{n}, :, ..., :] + \mathcal{Z}_0[\underbrace{:, ..., : j-1}_{n}, :, ..., :]$
  for $b_1 \in 1 \to B$ **do**
    $\mathcal{Z}_1 \leftarrow \mathcal{Z}_0[t_{b_1}(i_1)], \underbrace{:, ..., :}_{n-1}] - \mathcal{Z}_0[h_{b_1}(i_1) - 1, \underbrace{:, ..., :}_{n-1}]$
    for $b_2 \in 1 \to B$ **do**
      $\mathcal{Z}_2 \leftarrow \mathcal{Z}_1[t_{b_2}(i_2), \underbrace{:, ..., :}_{n-2}] - \mathcal{Z}_1[h_{b_2}(i_2) - 1, \underbrace{:, ..., :}_{n-2}]$

⋮

      for $b_N \in 1 \to B$ **do**
        $\mathcal{Z}_N \leftarrow \mathcal{Z}_{N-1}[t_{b_N}(i_N)] - \mathcal{Z}_{N-1}[h_{b_N}(i_N) - 1]$
        $\mathcal{X}[\mathbf{b}] \leftarrow \frac{1}{\mathcal{Z}_N}$

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

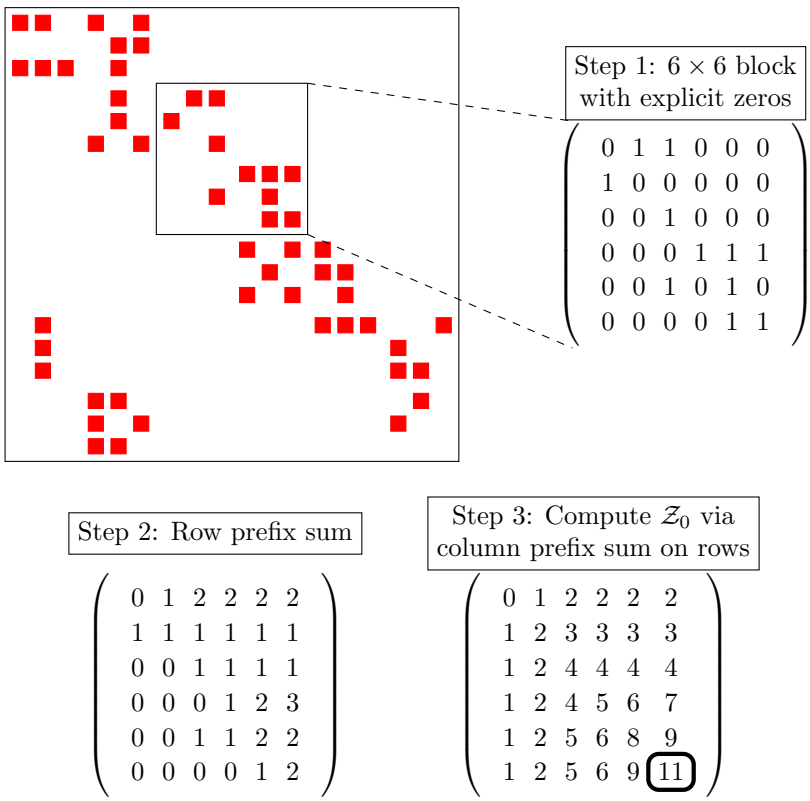

Figure 3.1 – A view of the prefix sum routine in a sparse matrix. The filled rectangles represent nonzero elements. In this example, we want to compute the number of nonzero elements in the 6×6 block. To do so, we fill in a block with ones where our matrix has ones and zeros where it has zeros in the block. We then perform a prefix sum on the rows, then the columns. The highlighted element in step 3 is the number of nonzeros in the block.

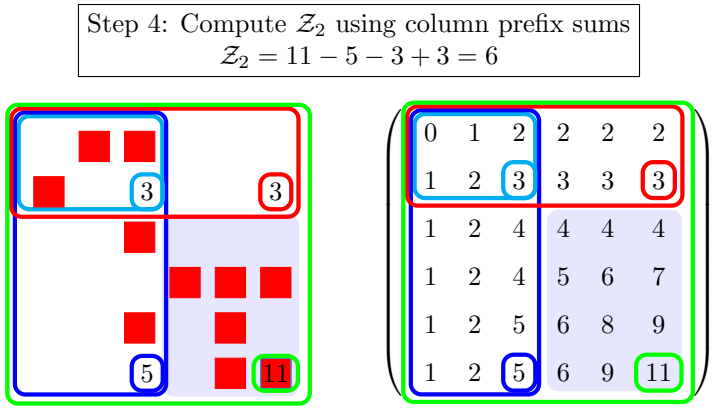

Figure 3.2 – An example of subtractions in COMPUTE$\mathcal{X}$ on a matrix. First, we compute the prefix sums. In this example, we want to find the number of nonzeros in the shaded area. We compute the number of nonzeros in the sub-block by subtracting the prefix sum results from the complement of the requested sub-block in the overall block.

The time complexity of ComputeX is as follows:

Theorem 3.2. *The algorithm* ComputeX *uses at most* $(N+1)(2B)^N$ *floating point operations (flops) to compute* $\mathcal{X}$.

Proof. Each prefix sum takes at most $(2B)^N$ additions to compute, and we compute N prefix sums. In the final loop, $\mathcal{Z}_n$ is of size $(2B)^{N-n}$. We must compute $\mathcal{Z}_n$ at most B^n times. Therefore, the block difference computation incurs at most $\sum_{n=1}^{N} 2^{-n}(2B)^N$ subtractions. We must also do one division for all B^N block sizes. Therefore, the algorithm uses at most $(N+1)(2B)^N$ flops to compute $\mathcal{X}$, plus the time spent in NonzerosInRange. □

3.2.1 NonzerosInRange

The implementation of NonzerosInRange depends on the initial format of the sparse matrix $\mathcal{A}$. We discuss two possible implementations to give the reader an idea of how one might implement this routine and to explain why this routine should not be costly in theory or practice.

If $\mathcal{A}$ is a matrix in CSR format (where nonzeros in each row are stored in sorted order of their column index), then using a binary search within each row provides an $O(B\log_2(N) + B^2)$ implementation, where the B^2 term reflects the maximum number of indices that may need to be returned.

If $\mathcal{A}$ is a tensor stored as an unsorted list of nonzero indices, we can perform the following procedure. Before we run EstimateFill, we block the entire matrix $\mathcal{A}$ into blocks of size $B \times ... \times B$, 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. Then, our implementation of NonzerosInRange, which is only ever called with ranges of size $2B \times ... \times 2B$, needs only to look up the 3^N 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 cost of $O(k(\mathcal{A}))$ and an individual query cost of $O(3^N B^N)$.

4 Analysis

We use a *sampling procedure* to repeatedly sample nonzero entries of the tensor and evaluate $x_{\mathbf{b}}$ on each selected entry, for all parameter settings $\mathbf{b}$ simultaneously. At the end of the algorithm, we averages these values for each blocking scheme $\mathbf{b}$, to obtain an estimate of $f_{\mathbf{b}}(\mathcal{A})$ for all $\mathbf{b}$. Suppose that our procedures samples a total of S nonzeros of $\mathcal{A}$, which are located at positions $\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_S$. We want to select S as small as possible for efficiency while still having provable guarantees on the concentration of our unbiased estimator $\sum_j x_{\mathbf{b}}(\mathbf{i}_j)/S$.

We give two concentration bounds for our estimator: one that assumes that the samples $\mathbf{i}_j$ are sampled *with replacement*, using Hoeffding's inequality [17], and an improved version where the $\mathbf{i}_j$ are sampled *without replacement*, using the recent Hoeffding-Serfling inequality due to Bardenet and Maillard [3]. Although the two concentration bounds are identical as the number of nonzeros $k(\mathcal{A})$ grows, the two bounds differ when the number of nonzeros is small.

4.1 A concentration bound when sampling with replacement

We make use of Hoeffding's inequality, which we state here for completeness:

Theorem 4.1 ([17])**.** *Let* $X_1, X_2, \ldots, X_M$ *be* M *independent random variables bounded such that* $0 \le X_j \le 1$. *Let* $\bar{X} = \frac{1}{M}\sum_{j=1}^{M} X_j$ *be their mean. Then for any* $t \ge 0$,

$$\Pr[|X - \mathbb{E}[X]| \ge t] \le 2\exp(-2Mt^2)\,.$$

Observe that for any blocking scheme $\mathbf{b}$ and any tensor element $\mathbf{i}$, the value $x_{\mathbf{b}}(\mathbf{i})$ 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 $x_{\mathbf{b}}(\mathbf{i}_1), x_{\mathbf{b}}(\mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathbf{i}_S)$ are independent. We can therefore apply Theorem 4.1 to obtain our first concentration bound:

Theorem 4.2 (Restatement of Theorem 1.2)**.** *If we sample at least* $S \geq \frac{NB^N}{2\varepsilon^2} \log\left(\frac{2B}{\delta}\right)$ *samples with replacement, then*

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

Proof. By definition, $F_{\mathbf{b}} = \frac{b_1 b_2 \cdots b_N}{S} \sum_{j=1}^{S} x_{\mathbf{b}}(\mathbf{i}_j)$. By Theorem 1.1, $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}$. $x_{\mathbf{b}}(\mathbf{i}_1), x_{\mathbf{b}}(\mathbf{i}_2), \ldots, x_{\mathbf{b}}(\mathbf{i}_S)$ are independent and bounded between 0 and 1. By Theorem 4.1, we have

$$\Pr[|F_{\mathbf{b}} - f_{\mathbf{b}}| \geq \varepsilon f_{\mathbf{b}}] \leq 2\exp(-2S\varepsilon^2 f_{\mathbf{b}}^2) \leq 2\exp(-2S\varepsilon^2/B^N)\,,$$

since $F_{\mathbf{b}}$ is $b_1 b_2 \cdots b_N$ times an average of S values, each of which is at least $1/B^N$, since each block has at least one nonzero entry and is of size at most B^N. By the union bound over the B^N possible blocking schemes $\mathbf{b}$,

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

Therefore, if $S \geq \frac{NB^N}{2\varepsilon^2} \log\left(\frac{2B}{\delta}\right)$,

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

Note that this bound is not directly dependent on $k(\mathcal{A})$. Clearly, 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. This bound is quite reasonable when $S \ll k(\mathcal{A})$, the number of nonzero entries in the tensor. Because it is constant with respect to the number of nonzeros, the bound on S could (and does, for realistic settings of ε and δ) exceed the number of nonzeros. This is fundamental to bounds based on Hoeffding's inequality, and methods based on sampling-with-replacement in general. In the next section, we obtain a bound on the number of samples needed that scales "smoothly" with the number of nonzeros, and critically never exceeds it.

4.2 A concentration bound when sampling without replacement

Recall that our algorithm can be viewed as sampling a set of random locations in the tensor, and then evaluating the *deterministic* function $x_{\mathbf{b}}$ for various blocking schemes $\mathbf{b}$ at that location. In the previous section, we imagined sampling the locations with replacement, so that the selected nonzeros are independent. By a stochastic domination argument, we could easily extend this bound to the case where the locations are sampled without replacement, but the bound remains weak in the regime where the minimal S and $k(\mathcal{A})$ are similar in size. Here, we use the following recent concentration bounds for sampling *without replacement* due to Bardenet and Maillard [3]:

Theorem 4.3 (Hoeffding-Serfling inequality)**.** *Let* $\chi = \{x_1, x_2, \ldots x_M\}$ *be a finite population of* $M > 1$ *real points with* $a = \min_j x_j$ *and* $b = \max_j x_j$*. Let* $(X_1, X_2, \ldots X_m)$ *be a list of size* $m < M$ *sampled without replacement from* χ*. Then for all* $\varepsilon > 0$*, we have*

$$\Pr\left[\frac{1}{m}\sum_{j=1}^{m} X_j - \frac{1}{M}\sum_{j=1}^{M} x_j \geq \varepsilon\right] \leq \exp\left(-\frac{2m\varepsilon^2}{(1 - m/M)(1 + 1/m)(b-a)^2}\right).$$

Here, our finite populations are $\chi_{\mathbf{b}}$, the images of the nonzeros under the functions $x_{\mathbf{b}}$. Observe that our algorithm as stated in Section 3 samples points independently, but without replacement, and so we can readily apply Theorem 4.3.

Theorem 4.4 (Restatement of Theorem 1.3)**.** *Let* $T = \frac{NB^N}{2\varepsilon^2} \log\left(\frac{2B}{\delta}\right)$. *If*

$$S \geq S_0 = \frac{T - T/k(\mathcal{A}) + \sqrt{(T - T/k(\mathcal{A}))^2 + 4T(1 + T/k(\mathcal{A}))}}{2 + 2T/k(\mathcal{A})},$$

then

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

Proof. By Theorem 1.1, $\mathbb{E}[F_{\mathbf{b}}] = f_{\mathbf{b}}$, and so by Theorem 4.3,

$$\Pr[F_{\mathbf{b}} \notin (1 \pm \varepsilon) f_{\mathbf{b}}] \leq 2 \exp\left(-\frac{2S\varepsilon^2}{(1 - S/k(\mathcal{A}))(1 + 1/S)}\right).$$

Taking the union bound over the B^N such parameters, we obtain that the probability that we have more than ε relative error for *any* blocking scheme is

$$\begin{aligned}\Pr\left[\max_{\mathbf{b} \leq B} \frac{|f_{\mathbf{b}} - F_{\mathbf{b}}|}{f_{\mathbf{b}}} \geq \varepsilon\right] &\leq 2B^N \exp\left(-\frac{2S\varepsilon^2}{(1 - S/k(\mathcal{A}))(1 + 1/S)}\right) \\ &\leq 2B^N \exp\left(-\frac{2S_0\varepsilon^2}{(1 - S_0/k(\mathcal{A}))(1 + 1/S_0)}\right).\end{aligned}$$

Substituting our expression for S_0 and rearranging, we obtain that

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

5 Results

We implemented[3] our algorithm, which we will refer to as `ASX`, for sparse matrices in CSR format in C.

We chose C to provide a fair comparison to the competing algorithm described in [2], which we will refer to as `OSKI`. To remove differences in speed from using different library functions, we modified `OSKI` to use the default GNU Scientific Library (GSL) random number generator, which is an implementation of the `mt19937` Mersenne twister, a pseudorandom number generator which is considered suitable for use in Monte Carlo simulations [18, 19]. To avoid differences in speed due to different integer types, we modified both `ASX` and `OSKI` to store sparse matrix indices using the unsigned type `size_t`, a macro which expands to a 64-bit unsigned quantity on our system.

We also chose C because C can efficiently execute the dense integer and floating point operations in Algorithm 3.3. An important factor in the design of this algorithm was that most of the computational work is a good target for instruction-level parallelism and cache optimizations. We have not yet fully optimized the inner kernel, but we leave this remark here to hint at future work.

5.1 Test Cases

We run our code on 5 matrices, 3 of which are matrices taken from the SuiteSparse Collection and used by Vuduc et al. [2] to measure `OSKI`, and 2 of which are pathological cases we invented to trip up both `OSKI` and `ASX` respectively.

The three matrices taken from SuiteSparse attempt to show the performance of the fill algorithm over a variety of fill patterns which appear in practice.

`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).

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

`ct20stif` arises from the application of finite element analysis to a different problem, that of structural mechanics. This matrix consists of a mix of different block sizes, mostly 3×3 and 6×6.

`gupta1` is the matrix representation of a linear programming problem, and has no obvious block structure.

`pathological_ASX` is a matrix designed to bring out the worst in our `ASX` algorithm. Because the indices of nonzeros are sampled with equal probability, blocks with many nonzeros become much more likely to be sampled than blocks without nonzeros. We maximize the probability of sampling full blocks by filling them completely. We minimize the probability of sampling sparser blocks by leaving only one nonzero in each one. We create a 1000×1000 matrix with 500 full 12×12 blocks and 500 sparse 12×12 blocks. This is a matrix which `ASX` should perform poorly on.

`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 10000×10000, and the first 6 rows are dense, while all other rows have only a 1 in the first column.

5.2 Performance Results

Vuduc et al. describe how the fill heuristic is multiplied by a performance constant to create a performance heuristic. After computing the approximate fill for each blocking, the blocking with the maximum such heuristic value is chosen [2]. We can guarantee that the relative error between the estimated best performance heuristic value and the true best performance heuristic value is at most the maximum relative error in all of the estimates:

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

Therefore, we measure the mean over several trials of the maximum relative error over all estimates. Keep in mind that if the mean maximum relative error is greater than 1, this represents a complete loss of accuracy, as a bogus algorithm which returns 0 for all estimates would achieve a better mean maximum relative error.

Each data point on the following plots represents the mean (for both maximum relative error and time) of 100 runs. All trials were performed on a Mid 2015 15-inch Retina MacBook Pro boasting an Intel®Core™i7-4770HQ CPU @ 2.20GHz Processor with 32KB of L1 cache, 26KB of L2 cache, 6.3MB of L3 cache, and 64B cache lines.

In practice, we found that the runtime and accuracy of the `OSKI` implementation varied substantially across matrices, even for the same recommended parameter setting of $\sigma = 0.02$. Ideally, sampling algorithms should be able to provide users with a consistent level of accuracy so that they can use the estimates provided with some confidence. Figure 5.1 shows the runtime and accuracy of `ASX` and `OSKI` on all of the above matrices using default settings. We see that on practical test cases, `ASX` provides results which are twice as accurate as `OSKI` in half the time. We also see that `ASX` performs about as well on pathological matrices as it does on practical ones, whereas `OSKI` does not provide useful estimates on these matrices.

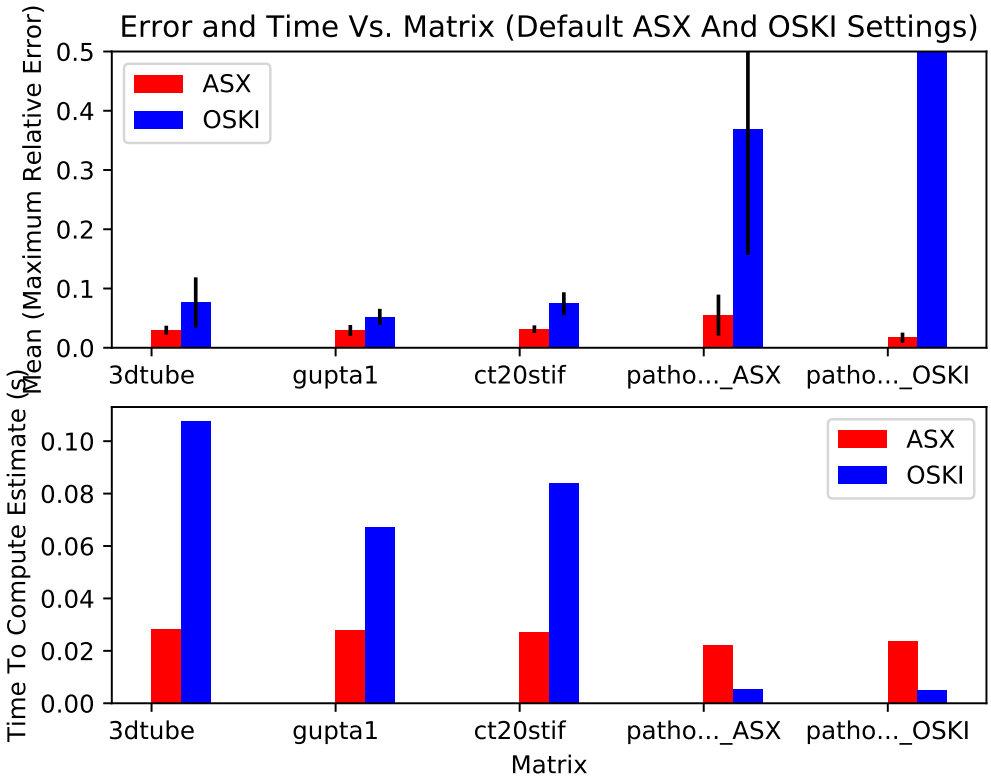

Figure 5.1 – Accuracy and Time for both `ASX` and `OSKI` on several matrices (average of 100 trials, error bars reflect one standard deviation above and below the mean). `ASX` parameters are $\varepsilon = 0.5$ and $\delta = 0.01$. `OSKI` parameters are $\sigma = 0.02$. In all cases, the average error due to `OSKI` is greater than that of `ASX`. `ASX` is faster on real-world matrices and slower on pathological cases, but unlike `OSKI`, `ASX` provides useful results for the pathological cases.

The variance in `OSKI`'s runtime made it difficult to create useful plots. Despite this difficulty, almost all sampling algorithms, including `OSKI`, provide some tradeoff between accuracy and runtime. Even though the relationship between `OSKI`'s parameters and its runtime and accuracy is unpredictable, the relationship between `OSKI`'s runtime and its accuracy is familiar. Therefore, we show, for both `ASX` and `OSKI`, the error in the estimated fill after running these methods for differing lengths of time.

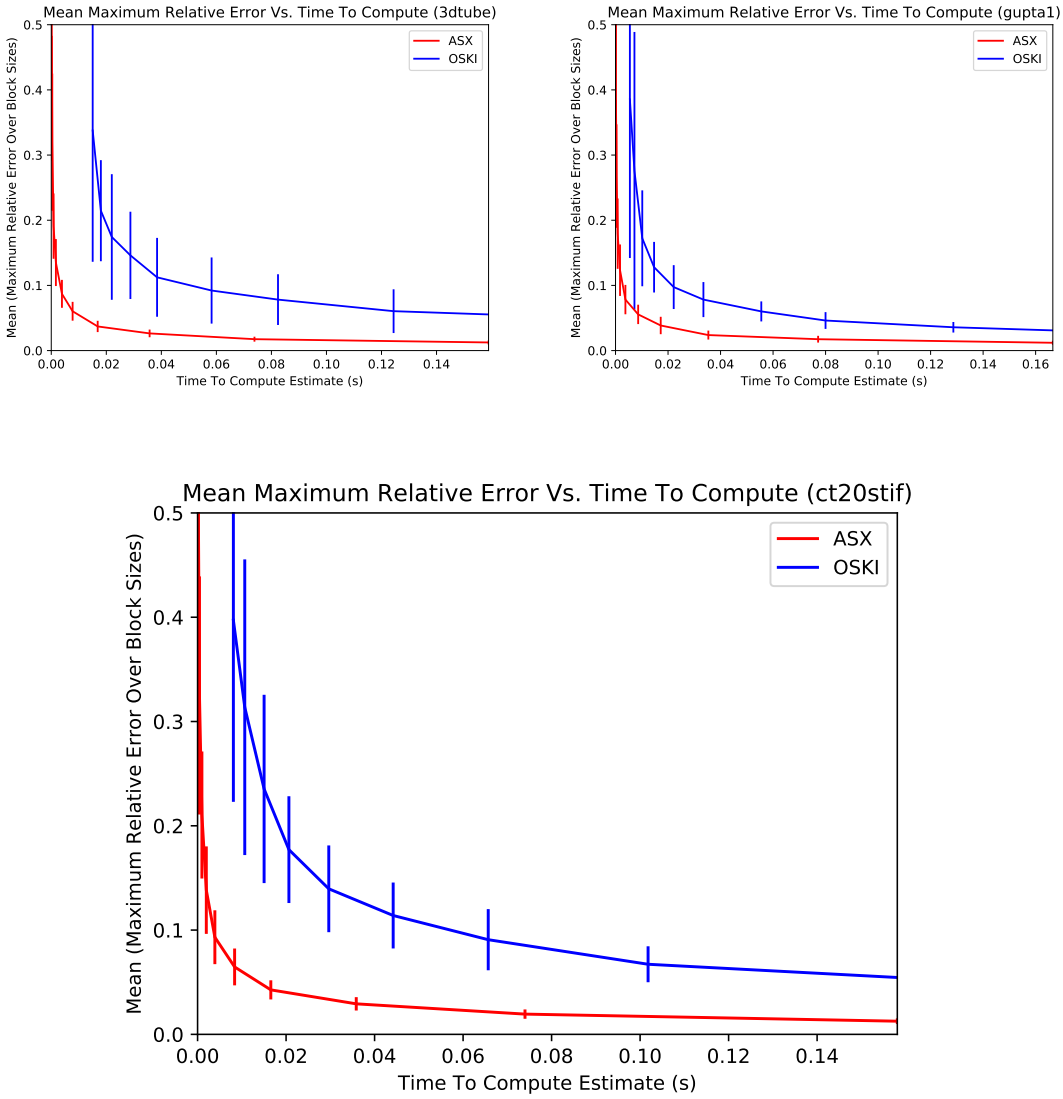

Figure 5.2 – Accuracy vs. Time Tradeoffs for matrices which arise in practice (average of 100 trials, error bars reflect one standard deviation above and below the mean). Given the same runtime, `ASX` estimates the fill more accurately than `OSKI` does in all cases.

Figure 5.2 shows the performance of the two algorithms on the matrices which arise in practice. Here we see that `ASX` reliably provides results within 0.1 relative error much faster than `OSKI` can.

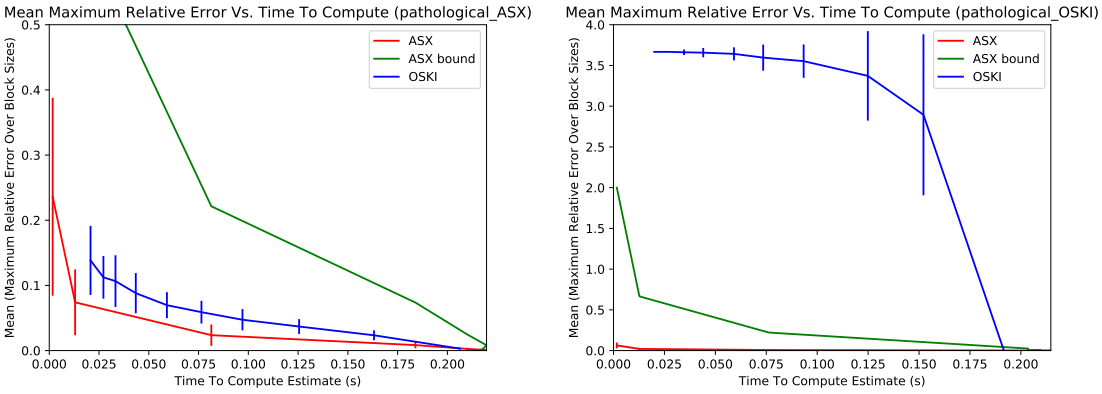

Figure 5.3 – Accuracy Vs. Time Tradeoffs for pathological cases (average of 100 trials, error bars reflect one standard deviation above and below the mean). Here we show the theoretical bound ε for `ASX` corresponding to a setting of $\delta = 0.01$.

Figure 5.3 shows the performance of the two algorithms on pathological cases. We ran both `ASX` and `OSKI` to completion, meaning that both algorithms were run until they computed an exact estimate. In both the `pathological_ASX` and `pathological_OSKI` cases, we find that `ASX` estimates the fill more accurately in less time than `OSKI`. In the `pathological_ASX` case, we see that `ASX` performs better than `OSKI`, but the difference is smaller than in the practical cases.

For the `pathological_OSKI` case, we see that `OSKI` fails to estimate the fill in any reasonable runtime. The theoretical error bound of `ASX` approaches zero faster than `OSKI`'s empirical estimates in the `pathological_OSKI` case. Furthermore, in both pathological cases, the empirical error of `ASX` is lower than the theoretical bound and is significantly lower than the empirical error rate of `OSKI` in the `pathological_OSKI` case.

6 Conclusion

We have shown our algorithm to be more predictable and accurate than existing approaches. Our algorithm can efficiently compute an approximation of the fill in a wide variety of circumstances. Specifically, `ASX` computes the fill more accurately and faster than existing approaches on real-world inputs and provides useful estimates of the fill in pathological test cases.

Sampling techniques are useful in autotuning since we can often sacrifice some accuracy in the heuristics for a faster autotuner runtime. As libraries for numerical computation evolve and autotuning moves from compile-time implementations to run-time implementations, developers will need heuristics whose execution time is small in comparison to that of the routine which is being tuned [20].

We have shown an example of how sampling can be applied to autotuning to efficiently produce good approximations with provable guarantees and hope that this work shows the broader potential for sampling techniques when designing 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.

6.1 Future Work

The main motivation behind the design of our algorithm was to express the problem as a dense set of operations that can be computed efficiently. We have shown that our approach is faster than existing approaches in all test cases. Future work includes a parallel implementation of both

`ASX` and `OSKI`. Although both algorithms easily parallelize to a multicore setting, we hope to gain even more performance through instruction level parallelism in dense operations such as the prefix sum and the differences operation.

Recall that a blocking scheme $\mathbf{b}$ is defined by the dimensions of each block $(b_1, b_2, \ldots, b_N)$. Another variant on the fill estimation problem introduces *offsets*. An offset is defined by a vector $\mathbf{o} = (o_1, o_2, \ldots o_N)$ such that the block indices of a nonzero element $\mathbf{i}$ are defined as follows:

$$\left(\left\lceil \frac{i_1 + o_1}{b_1} \right\rceil, \left\lceil \frac{i_2 + o_2}{b_2} \right\rceil, ..., \left\lceil \frac{i_N + o_N}{b_N} \right\rceil \right) .$$

Both the `ASX` and `OSKI` algorithm currently estimate the fill using only block sizes, but some matrices may have smaller fills in an offset blocked sparse matrix. Because most of the information needed to compute $x_{\mathbf{b},\mathbf{o}}$ is already computed when we compute $x_{\mathbf{b}}$, we can compute the fill over all possible combinations of block sizes and offsets by modifying only the differences operation.

Another extension to the problem is to limit the *volume* of the blocking scheme. That is, for any blocking scheme $\mathbf{b}$, we require $b_1 \times b_2 \times \ldots \times b_N \leq V$ for some maximum volume V. For volume to be a nontrivial quantity, we would set it to less than B^N where B is the maximum size of a block dimension. If the blocks are too large, the performance of tensor operations declines as we are required to fill in more explicit zeros. Thus, we are unlikely to choose these block sizes after calculating the fill. If the volume is limited to V, then the expected value of $x_b(\mathcal{A}, \mathbf{i})$ on a randomly chosen $\mathbf{i}$ is at least $1/V$. Thus, limiting the volume increases the lower bound on the expected value of the fill, meaning that the theoretical accuracy guarantee will get closer to the empirically measured accuracy.

References

[1] Rich Vuduc, James W Demmel, Katherine A Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 26–26. IEEE, 2002.

[2] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[3] Rémi Bardenet, Odalric-Ambrym Maillard, et al. Concentration inequalities for sampling without replacement. *Bernoulli*, 21(3):1361–1385, 2015.

[4] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[5] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[6] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 5. ACM, 2015.

[7] Brett W Bader and Tamara G Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[8] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. 2017. MIT-CSAIL-TR-2017-003 `http://hdl.handle.net/1721.1/107013`.

[9] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[10] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[11] Buse Yilmaz, Bariş Aktemur, MaríA J. Garzarán, Sam Kamin, and Furkan Kiraç. Autotuning runtime specialization for sparse matrix-vector multiplication. *ACM Trans. Archit. Code Optim.*, 13(1):5:1–5:26, March 2016.

[12] Richard Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. *High Performance Computing and Communications*, pages 807–816, 2005.

[13] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *The International Journal of High Performance Computing Applications*, 21(4):467–484, 2007.

[14] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM sigplan notices*, volume 45, pages 115–126. ACM, 2010.

[15] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. extscSparsity: Optimization framework for sparse matrix kernels. *Int'l. J. High Performance Computing Applications (IJHPCA)*, 18(1):135–158, February 2004.

[16] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in sparsity. *Computational Science?ICCS 2001*, pages 127–136, 2001.

[17] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[18] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[19] Makoto Matsumoto. Mersenne Twister with improved initialization. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`, 2002. Online; accessed 14 May 2017.

[20] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–131, 2003.