CS8656 Project Proposal // An Efficient Fill Estimation Algorithm for Sparse Tensors in Blocked Formats

Peter Ahrens, Nicholas Schiefer, Helen Xu

We present an algorithm to efficiently compute an important heuristic for autotuning sparse tensor operations. A tensor is a multidimensional array. A sparse tensor is a tensor whose entries are mostly zeros, which do not have to be stored or operated on in most linear algebraic operations. Since sparse tensors typically contain more than 90% zero entries, taking advantage of sparsity can provide substantial increases in performance. However, the increased complexity of datastructures that can describe the irregular locations of nonzeros in these tensors poses a significant challenge to performance engineers.

These challenges are magnified in an era of increasing heterogeneity of processors. 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 and the structure of nonzeros is usually known only at runtime. Therefore, autotuning (automatically generating customized code) has become a necessary part of writing efficient sparse code.

Previous efforts in autotuning for sparse tensors focus on sparse matrices, which are more broadly applicable in fields ranging from scientific computing to machine learning. 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. We limit our description to perhaps the most popular such format, Compressed Sparse Row (CSR) and a variant we will call Blocked Compressed Sparse Row (BCSR). In CSR, only the nonzeros and their locations are stored in each row of the matrix. In Blocked Compressed Sparse Row, an $m \times n$ matrix is divided into $m/r \times n/c$ submatrices, where each submatrix is of size $r \times c$. Only blocks which contain nonzeros are stored, and these nonzero blocks are stored in CSR format. Storing the locations of nonzero blocks requires less memory and less computational logic than storing the locations of individual nonzeros. For matrices that naturally have a block structure of nonzeros, such as those produced by finite element methods, this can lead to an increase in the performance of sparse matrix operations. However, because we store the entire block (filling in zero entries with explicit zeros) setting the block size to be too large can decrease the performance.

Given the definition of BCSR, it is natural to wonder how one might choose the correct block size for a given matrix. Vuduc et. al. describes an effective heuristic for predicting the performance P (in $Mflop/s$) of a particular block size on a sparse matrix A. We refer to the number of nonzeros in A as $k(A)$. We refer to the number of blocks of size $r \times c$ which contain nonzeros in A as $k_{r,c}(A)$. We can then define the *fill* of the matrix to be $f(A) = rck_{r,c}(A)/k(A)$. Once per machine, we compute a profile of how the machine performs for a particular block size. Let $P_{rc}(dense)$ be the performance of the machine (in $Mflop/s$) on a dense matrix stored with block size $r \times c$. Then we can estimate $P_{rc}(A)$ as

$$\tilde{P}_{rc}(A) = \frac{P_{rc}(dense)}{f_{rc}(A)}$$

Thus, our task is to compute f_{rc} for all r and c to within some tolerable relative accuracy, and to do so efficiently. Statistical sampling methods given by Vuduc et. al. provide no theoretical guarantee of accuracy, and take as long as 1 to 10 times the time it takes to perform a sparse matrix vector multiplication on the same matrix. We describe an algorithm which provides estimates to within ϵ relative error with probability $1 - \delta$ in time $O(\log(\delta)/\epsilon^2)$, and show that our algorithm runs efficiently and accurately on real-world cases. Note that our algorithm depends only on the desired accuracy, whereas the algorithm in [**?**] depends linearly upon the number of nonzeros.

Our algorithm estimates a more general notion of fill for tensors, where we divide the tensor into smaller subarrays (our blocks) and again only the nonzero blocks and their locations are stored in each row of the tensor. We further generalize this by allowing the user to offset the grid of blocks by some fixed amount, so that the block structure of the tensor does not have to align with the block size.

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

1 Notation

A *tensor* is a multidimensional array. A tensor of *order* N is an element of the tensor (direct) product of N vector spaces. We assume all of our vector spaces are over an arbitrary field $\mathbb{F}$. Vectors are order 1 tensors and will be denoted by boldface lowercase letters, like this: $\mathbf{a}$. Matrices are order 2 tensors and will be denoted by boldface capital letters, like this: $\mathbf{A}$. Tensors will be denoted by boldface capital Euler script letters, like this: $\boldsymbol{\mathcal{A}}$.

We refer to populations using capital Euler script letters, like this: $\mathcal{X}$. We refer to random variables and index bounds using capital letters, like this: X. We refer to functions, indices and elements of populations using lowercase letters, like this x.

The n^{th} element in a sequence is denoted by $\boldsymbol{\mathcal{A}}^{(n)}$. Element $(i_1, i_2, ..., i_N)$ of an order-N tensor $\boldsymbol{\mathcal{A}} \in \mathbb{F}^{I_1 \times I_2 \times ... \times I_N}$ is denoted $\boldsymbol{\mathcal{A}}_{i_1,i_2,...,i_N}$ or $\boldsymbol{\mathcal{A}}[i_1, i_2, ..., i_N]$.

If we wish to represent the integer range $i, i+1, ..., j$, we use the syntax $i \to j$. When this syntax appears as a subscript, it is a shorthand for the range of subscripted variables. For example, we use the syntax $i_{1\to N}$ as a shorthand for $i_1, i_2, ..., i_N$.

Subarrays are formed when we fix a subset of indices. We use a colon to indicate all elements of a mode. Thus, the middle $n/2$ columns of a matrix $\mathbf{A} \in \mathbb{F}^{n\times n}$ would be written $\mathbf{A}_{:,n/4\to 3n/4}$.

For convenience, we say that $\mathcal{A} = 0$ if and only if every element of $\mathcal{A}$ is 0.

2 Formulation of the problem

We are given a tensor $\mathcal{A} \in \mathbb{F}^{I_1\times I_2\times ...\times I_N}$ and positive integers B, o where $0 \leq o < B$.

We will group the nonzero natural numbers into contiguous **blocks** of size B, and shift these blocks by o. The function l looks up the block index j of a number i, so that i is in the j^{th} block.

$$l_{B,o}(i) = \left\lceil \frac{i+o}{B} \right\rceil$$

We also define a sort of inverse function of l, r, which returns the range of numbers corresponding to the j^{th} block.

$$r_{B,o}(j) = (o + j * (B-1) + 1) \to (o + j * B)$$

We can extend this blocking concept to multiple dimensions. An N-dimensional **blocking scheme** $b = (B_1, B_2, ..., B_N, o_1, o_2, ..., o_N)$ is characterized by block dimensions $B_1, B_2, ..., B_N$ and block offsets $o_1, o_2, ..., o_N$ where $0 \leq o_n \leq B_n$ for all $1 \leq n \leq N$. We say that $b \leq B$ if $B_1, B_2, ..., B_N \leq B$. We extend our definitions of l and r to an N-dimensional blocking scheme b as follows:

$$r_b(j_{1\to N}) = r_{B_1,o_1}(j_1) \times r_{B_2,o_2}(j_2) \times ... \times r_{B_N,o_N}(j_N)$$

$$l_b(i_{1\to N}) = (l_{B_1,o_1}(i_1), l_{B_2,o_2}(i_2), ..., l_{B_N,o_N}(i_N))$$

Let $k(\mathcal{A})$ be the number of nonzero elements of the tensor $\mathcal{A}$. The definition of k can be extended to an N-dimensional blocking scheme b so that $k_b(A)$ is the number of nonzero blocks in the tensor $\mathcal{A}$.

$$k_b(\mathcal{A}) = \sum_{(j_{1\to N}) | \mathcal{A}[r_b(j_{1\to N})] \neq 0} 1$$

Thus, if we broke up our range of tensor indices into blocks of size $B_1, B_2, ..., B_N$ and offset these blocks by $o_1, o_2, ..., o_N$, $k_{(B_{1\to N}, o_{1\to N})}(\mathcal{A})$ tells us how many of these blocks would be needed to cover the nonzeros of $\mathcal{A}$. Note that $k_{(1,1,...,1,0,0,...,0)}(\mathcal{A}) = k(\mathcal{A})$.

Now we can formally define the **fill** f_b.

$$f_b(\mathcal{A}) = \frac{k_b(\mathcal{A})}{k(\mathcal{A})}$$

The problem is to compute an approximation $\tilde{f}_b(\mathcal{A})$ such that $f_b(\mathcal{A})(1-\epsilon) \leq \tilde{f}_b(\mathcal{A}) \leq f_b(\mathcal{A})(1+\epsilon)$ for all N-dimensional blocking schemes $b \leq B$ with probability at least $1-\delta$.

3 Previous Work

finish plz. Mainly this is Vuduc

4 The Algorithm

We define the function x_b on each nonzero element $(i_{1\to N})$ of $\mathcal{A}$ as follows.

$$x_b(\mathcal{A}, i_{1\to N}) = \frac{1}{k(\mathcal{A}[r_b(l_b(i_{1\to N}))])}$$

$x_b(\mathcal{A}, i_{1\to N})$ is therefore equal to the reciprocal of the number of nonzeros in its block. Consider the sum of x_b over all of the nonzeros of A. We have that

$$\begin{aligned}
&\sum_{(i_{1\to N})|\mathcal{A}[i_{1\to N}]\neq 0} x_b(\mathcal{A}, i_{1\to N})\\
&= \sum_{(j_{1\to N})|\mathcal{A}[r_b(j_{1\to N})]\neq 0} \left(\sum_{(i_{1\to N})\in r_b(j_{1\to N})|\mathcal{A}[i_{1\to N}]\neq 0} x_b(A, i_{1\to N}) \right)\\
&= \sum_{(j_{1\to N})|\mathcal{A}[r_b(j_{1\to N})]\neq 0} \left(\sum_{(i_{1\to N})\in r_b(j_{1\to N})|A(i_{1\to N})\neq 0} \frac{1}{k(\mathcal{A}[r_b(l_b(i_{1\to N}))])} \right)\\
&= \sum_{(j_{1\to N})|\mathcal{A}[r_b(j_{1\to N})]\neq 0} \left(\sum_{(i_{1\to N})\in r_b(j_{1\to N})|A(i_{1\to N})\neq 0} \frac{1}{k(\mathcal{A}[r_b(j_{1\to N})])} \right)\\
&= \sum_{(j_{1\to N})|\mathcal{A}[r_b(j_{1\to N})]\neq 0} 1\\
&= k_b(\mathcal{A})
\end{aligned}$$

Consider the population $\mathcal{X}_b(\mathcal{A}) = (x_b(\mathcal{A}, i_{1\to N})|\mathcal{A}(i_{1\to N}) \neq 0)$. We have just shown that the average value of elements in $\mathcal{X}_b(\mathcal{A})$ is

$$\frac{\sum_{(i_{1\to N})|\mathcal{A}(i_{1\to N})\neq 0} x_b(\mathcal{A}, i_{1\to N})}{\|\{(i_{1\to N})|\mathcal{A}(i_{1\to N}) \neq 0\}\|} = \frac{k_b(\mathcal{A})}{k(\mathcal{A})} = f_b(\mathcal{A})$$

Thus, our task is to randomly sample elements from $\mathcal{X}_b$ to compute an estimate of its average. We can compute a sample of $\mathcal{X}_b$ 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, once we have the locations of all the nonzeros within a B radius of our nonzero at index $i_{1 \to N}$, we can compute $x_b(i_{1 \to N})$ for all $b \leq B$ at the same time, saving an enormous amount of work. We call this algorithm SAMPLE.

5 Analysis of Algorithm

Here, we will beat the analysis of this algorithm to death so that we can get a bound on the number of operations required to compute this estimate (we are talking about constants after all)

5.1 Error Analysis

Here, we will bound (very tightly) the number of samples needed

5.2 Runtime Analysis

Here, we will bound (very tightly) the number of operations per sample needed

6 Results

Here we explore the relationship between runtime and accuracy of the fill prediction on several matrices from Vuduc et. al. and also from the suitesparse collection from florida