Combinatorial Tiling for Sparse Neural Networks
Filip Pawlowski, Rob H Bisseling, Bora Uçar, Albert-Jan N Yzelman

To cite this version:
Filip Pawlowski, Rob H Bisseling, Bora Uçar, Albert-Jan N Yzelman. Combinatorial Tiling for Sparse Neural Networks. 2020 IEEE High Performance Extreme Computing (virtual conference), Sep 2020, Waltham, MA, United States. hal-02910997v3

HAL Id: hal-02910997
https://inria.hal.science/hal-02910997v3
Submitted on 3 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Combinatorial Tiling for Sparse Neural Networks

Filip Pawłowski†, Rob H. Bisseling‡, Bora Uçar¶, and A. N. Yzelman§
† Huawei Paris Research Center
Boulogne-Billancourt, France
‡ ENS Lyon, filip.pawlowski@ens-lyon.fr
§ Huawei Zürich Research Center
Zürich, Switzerland
{filip.pawlowski1, albertjan.yzelman}@huawei.com

Abstract—Sparse deep neural networks (DNNs) emerged as the result of search for networks with less storage and lower computational complexity. The sparse DNN inference is the task of using such trained DNN networks to classify a batch of input data. We propose an efficient, hybrid model- and data-parallel DNN inference using hypergraph models and partitioners. We exploit tiling and weak synchronization to increase cache reuse, hide load imbalance, and hide synchronization costs. Finally, a blocking approach allows application of this new hybrid inference procedure for deep neural networks. We initially experiment using the hybrid tiled inference approach only, using the first five layers of networks from the IEEE HPEC 2019 Graph Challenge, and attain up to 2x speedup versus a data-parallel baseline.

I. INTRODUCTION

The Graph Challenge encourages new developments in graph analytics, which, in its 2019 edition [1] focused on fast inference for sparse deep neural networks. It described the problem using the GraphBLAS C API [2], which advocates the use of matrix-based algorithms. Sparsely-connected neural networks exhibit lower computational complexity and lower memory requirements compared to their dense counterparts. They may originate by pruning a dense network as in the Banded Sparse Neural Networks [3], or result from training a fixed sparse topology in the RadiX-Net [4]. The input data matrix may also be sparse, due to feature extraction techniques generating sparse representations (from, e.g., image, video, or signal data), or because input may be naturally sparse (e.g., graph inputs). One dataset with sparse representations is the MNIST database of handwritten digits [5].

We summarise the sparse DNN challenge in terms of input and neural networks (NNs) used. A sparse neural network consists of \( \mathbb{N} \) sparse weight matrices \( W^{(k)} \in \mathbb{R}^{n_k \times n_{k+1}} \) called layers, where \( k, n_k, n_{k+1} \in \mathbb{N} \) for all \( 0 \leq k < d \). A feature matrix \( X^{(0)} \in \mathbb{R}^{n \times n_0} \) consists of \( n \in \mathbb{N} \) sparse feature vectors, one for each data instance to be classified. Sparse inference refers to the computation of the final classification matrix \( X^{(d)} \in \mathbb{R}^{n \times c} \) from the input feature matrix \( X^{(0)} \):

\[
X^{(d)} = f(\cdots f(f(X^{(0)} W^{(0)} + e_n b^{(0)T}) W^{(1)}) + e_n b^{(1)T}) \cdots W^{(d-1)} + e_n b^{(d-1)T}),
\]

where \( c = n_d \) is the number of classes, \( f : \mathbb{R} \to \mathbb{R} \) is an activation function to be applied element-wise, \( b^{(k)} \in \mathbb{R}^{n_{k+1} \times 1} \) is a vector of bias at layer \( k \), and \( e_n = (1, \ldots, 1)^T \).

The challenge’s input feature matrix consists of 60 000 images from the MNIST dataset with each image flattened to a single vector and thresholded such that the values are either 0 or 1. Images are interpolated to the number of neurons in the neural networks: 1024, 4096, 16384, and 65536. Several deep sparse neural networks are generated using RadiX-Net [4], with the number of neurons, layers, and bytes in Table I. This size, in bytes, assumes Compressed Row Storage (CRS) using four-byte values and indices. Inference employs the rectified linear unit (ReLU): for \( Y = f(X) \) and \( X, Y \in \mathbb{R}^{m \times n} \),

\[
Y_{ij} = f(X_{ij}) = \begin{cases} 0 & \text{if } X_{ij} < 0 \\ X_{ij}, & \text{otherwise} \end{cases}, \forall 0 \leq i < m, \ 0 \leq j < n.
\]

The bias \( b \) in Equation (1) is \(-0.3, -0.35, -0.4, -0.45\) for each of the 1k, 4k, 16k, and 64k neuron NNs, respectively. This correction can be integrated with the application of \( f \). Without loss of generality we hence omit \( b \) from the remainder text.

Sparse inference may be viewed as a repeated sparse matrix–sparse matrix multiplication (SpGEMM) \( C = AB \) with sparse matrices \( A, B, \) and \( C \) where nonzero output elements are filtered using the function \( f \). Thus, our goal is to compute \( X^{(k+1)} = f(X^{(k)} W^{(k)}) \) for each layer. Parallelization strategies typically work by partitioning the input feature matrix (data-parallel), partitioning the neural network (model-parallel), and/or by pipelining [6]. The previous sparse DNN challenge submissions exploit data-parallelism and propose methods to maintain load balance since the number of nonzero elements in the rows of the input matrix differ arbitrarily. There were six submissions [7]–[12] providing performance data for analysis [13]. Most employ high-performance frameworks, such as SuiteSparse:GraphBLAS, GraphBLAST, or Kokkos, to take advantage of highly-optimized SpGEMM kernels and achieve shorter development time. Since such frameworks are designed for (trans)portability, their use typically disallows optimizations across the individual steps of the inference.

Our approach assumes a shared-memory architecture consisting of \( p_s \) connected processors. Each supports \( p_t \) threads, for a total of \( p = p_s p_t \) threads. We pin threads to specific cores, so to prevent it moving from one core to another during the computation. To analyse the shared-memory algorithms, we distinguish between two types of data movement, intra-socket and inter-socket. These are quantified via the intra-socket cost
The main contribution of this paper is twofold. First, we introduce a hypergraph model of the sparse neural network inference process. Second, we propose an efficient shared-memory hybrid algorithm which combines the data- and model-parallel methods to achieve tiling through multiple layers of a neural network. The latter is enabled by the use of existing hypergraph partitioners, yielding an effective partitioning that leverages the density of weight matrices. We evaluate our techniques on the data and neural networks defined by the sparse DNN challenge.

### A. The challenge dataset

We extract some statistics of the 1k-neuron, 120-layer network to demonstrate important properties of the weight matrices. Let the density of an $m \times n$ matrix $A$ be $\frac{\text{nz}(A)}{mn}$, and the compressed density be $\frac{\text{nz}(A)}{m' n'}$, where $\text{nz}(\cdot)$ returns the number of nonzeros in a matrix and $m'$ is the number of nonempty rows. Table II shows the density and the compressed density of feature matrices $X(k)$ for a few of the first fifteen layers of the network, as well as the size of the memory occupied by the three CRS matrices in $X^{(k+1)} = X^{(k)}W^{(k)}$. The last column reports the total memory (in MB) occupied by the three CRS matrices in $X^{(k+1)} = X^{(k)}W^{(k)}$. We assume $j$ has been applied after all layers, except for $X^{(0)}$.

<table>
<thead>
<tr>
<th>Layer $k$</th>
<th>Compressed density</th>
<th>Density</th>
<th>Memory (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.38</td>
<td>10.38</td>
<td>136.70</td>
</tr>
<tr>
<td>1</td>
<td>31.08</td>
<td>28.33</td>
<td>149.15</td>
</tr>
<tr>
<td>2</td>
<td>20.65</td>
<td>13.94</td>
<td>80.99</td>
</tr>
<tr>
<td>3</td>
<td>34.28</td>
<td>6.86</td>
<td>45.98</td>
</tr>
<tr>
<td>4</td>
<td>62.56</td>
<td>4.26</td>
<td>29.07</td>
</tr>
<tr>
<td>5</td>
<td>97.23</td>
<td>3.45</td>
<td>23.63</td>
</tr>
<tr>
<td>6</td>
<td>99.93</td>
<td>3.12</td>
<td>21.93</td>
</tr>
<tr>
<td>7</td>
<td>100.00</td>
<td>3.02</td>
<td>21.46</td>
</tr>
</tbody>
</table>

### B. Sequential SpGEMM

There are a number of efficient SpGEMM algorithms that differ in the way the matrices are stored, the nonzeros are visited, and the way scalar multiples are accumulated [14]. Our own sequential codes, alike Davis et al.’s [8], use a variant of Gustavson’s algorithm with modifications to fuse the filtering step (ReLU) and thresholding of the nonzeros. We replace the index list typically used in a sparse accumulator SPA with a dense array SPAC, which flags columns belonging to the active row $i$. Contrary to an index list, this array need not be reset between iterations and is more efficient when rows of the output matrix hold relatively many nonzeros—which, as argued in Section I-A, holds for NN inference. Finally, since our inference algorithms require two consecutive SpGEMMs with a common input matrix, we implement a fused variant to prevent accessing that matrix twice.

The work complexity of $X^{(k+1)} = X^{(k)}W^{(k)}$ using Gustavson’s approach is $\Theta(\sum_{i=0}^{n-1} \text{nz}(X^{(i)}) \text{nz}(W^{(i)}))$. The CRS matrices of $X^{(k)}$ and $X^{(k+1)}$ are streamed to the CPU only once, which incurs a data movement of $\Theta(\text{nz}(X^{(k)}) + \text{nz}(X^{(k+1)}) + n)$. The CRS of $W^{(k)}$ as well as the sparse accumulator structures (SPA and SPAC of size $\Theta(\max_k nk_k)$) are accessed $\Theta(\text{nz}(X^{(k)}))$ times; the whole $W^{(k)}$ is streamed at most once per row of $X^{(k)}$, and only once at minimum.

### C. Sequential inference

The work complexity of the sparse inference (1) derived by consecutively applying the preceding SpGEMM analysis is $\Theta \left( \sum_{k=0}^{d-1} \left[ \sum_{i=0}^{n_k-1} \text{nz}(X^{(k)}_{i,\cdot}) \text{nz}(W^{(k)}_{i,\cdot}) \right] \right)$. Recall that while input and output feature matrices are touched only once, the weight matrix $W^{(k)}$ is touched multiple times. If this matrix can be cached, the data movement cost is

$$\Theta \left( \sum_{k=0}^{d-1} \left[ \text{nz}(X^{(k)}) + n \right] + \sum_{k=0}^{d-1} \left[ \text{nz}(W^{(k)}) + n_k \right] \right).$$

If $W^{(k)}$ cannot be cached this increases with $\Theta(\sum_{k=0}^{d-1} n_k \text{nz}(X^{(k)}_{\cdot,\cdot}) \text{nz}(W^{(k)}_{\cdot,\cdot}))$. We use two buffers to store input and output feature matrices throughout inference, as each SpGEMM may reuse the preceding SpGEMM’s input buffer to store its output while re-using the preceding output buffer as input. The total storage requirement thus becomes

$$\Theta \left( \max_k \text{nz}(X^{(k)}) + \sum_{k=0}^{d-1} \text{nz}(W^{(k)}) + n + \sum_{k=0}^{d} n_k \right).$$

### II. APPROACH

We present the data-parallel inference, which is the de-facto standard in deep learning in Section II-B. We then introduce the model-parallel approach in II-C which we extend to include tiling in Section II-D. Finally, we combine the tiling model-parallel inference with the data-parallel inference and describe how to apply the resulting method for deep neural networks in Section II-E. Our model-parallel, tiling, and hybrid tiling inference methods all rely on hypergraph modeling of the inference process. We introduce this first.
The internal rows or columns to the external set $N_S$ so that such conflicts will not arise in the subsequent presentation.

**B. Data-parallel inference**

Data-parallel inference partitions the feature matrix $X^{(0)} \in \mathbb{R}^{n \times n_c}$ into $p$ parts using a row-wise partitioning. We map row $i$ to thread $[i/n/p]$ to achieve this, so that each thread performs sequential inference on different sets of (approximately) $n/p$ input data elements in an embarrassingly parallel fashion.

Each thread $q$ allocates two buffers to store the local feature matrices $X_{\{s_q \}}^{(k)}$ and $X_{\{s_q \}}^{(k+1)}$ when processing $W^{(k)}$, where $s_q$ is the set of data indices assigned to thread $q$, while the weight matrices are allocated once and shared. This approach has no work overhead compared to a sequential method.

There is, however, a minimum data movement overhead of $\sum_{k=0}^{d-1} (nz(W^{(k)}) + n_q (p - p_q))$ as all threads need to refer to the shared weights; replication of the weights on each socket would reduce this to $\sum_{k=0}^{d-1} (nz(W^{(k)}) + n_q p_q)$. This bound increases for each weight matrix whose accesses to, depending on the density pattern of $X^{(k)}$, do not fit in cache. The storage overhead is $\Theta(p)$ since every thread maintains a CRS matrix.

**C. Model-parallel inference**

Model-parallel inference partitions the neural network, which in our case corresponds to a partitioning of the staircase matrix in Figure 1 mapping each individual weight to a unique thread. Let $W_q^{(k)}$ be the matrix consisting of the set of weights of $W^{(k)}$ distributed to thread $q$, and $R_q = \{r_{q0}, \ldots, r_{q(p-1)}\}$ be a partitioning of row indices of $W^{(k)}$ such that all weights on rows in $R_q^{(k)}$ are in $W_q^{(k)}$ (internal rows) while $R_q^{(k+1)}$ are rows shared by multiple parts of $W^{(k)}$ (external rows). Let $Z_q = \{z_{q0}, \ldots, z_{q(p-1)}\}$ be a similar partitioning of columns, and let $\lambda_q^{(k)}$ and $\lambda_q^{(k+1)}$ refer to the connectivities of row and column $q$, respectively. When the layer number $k$ can be inferred from context we shall omit the superscripts from the $r_{qj}$, $z_{qj}$, $\lambda_j^{(k)}$.

The matrix $W_q^{(k)}$ is stored locally by thread $q$, as are the corresponding input and output feature matrices $X_{\{r_q, z_q\}}^{(k)}$ and $X_{\{s_{qz}, z_{qz}\}}^{(k+1)}$. Since the column sets $r_{qj}$ and $z_{qj}$ are disjoint, each thread $q$ stores a disjoint set of features. The parts corresponding to separator columns $r_{qj}$, $z_{qj}$, however, are replicated across all threads. Since a thread $q$ may refer to remote data such as the separator features $X_{\{s_{qz}, z_{qz}\}}$, for $0 \leq r < p$, $r \neq q$, we retain the subscript $r$ to emphasize which data is allocated to what thread.

The nonzeros of $W_q^{(k)}$ can be logically subdivided into four parts from $W^{(k)}_{\{r_{qj}, z_{qj}\}}$, each with different data dependencies: (i) processing $W_q^{(k)}_{\{r_{qj}\}}$ only depends on the locally computed data $X_{\{r_{qj}\}}^{(k)}$; (ii) processing $W_q^{(k)}_{\{z_{qj}\}}$ requires collaboratively computed data $X_{\{s_{qz}, z_{qz}\}}^{(k)}$ for all $r$; (iii) processing $W_q^{(k)}_{\{r_{qj}, z_{qj}\}}$ requires collaborative computation of the output $\sum_{r=0}^{p-1} X_{\{s_{qz}, z_{qz}\}}^{(k+1)}$; and (iv) processing $W_q^{(k)}_{\{r_{qj}, z_{qj}\}}$...
requires both parallel read access on $X_{\{s_q,r_S\}}^{(k)}$ for all $r$ and collaborative computation of $\sum_{r=0}^{p-1} X_{\{s_q,r_S\}}^{(k+1)}$. We assume the number of rows $n$ is larger than or equal to $p$ and employ two-phase reduction to collaboratively compute $\sum_{r=0}^{p-1} X_{\{s_q,r_S\}}^{(k+1)}$. This subdivides the total number of rows $n$ into $p$ parts and makes each thread responsible for reducing approximately $n/p$ rows only; thread $q$ hence computes $X_{\{s_q,r_S\}}^{(k+1)} = \sum_{r=0}^{p-1} X_{\{s_q,r_S\}}^{(k+1)}$ and stores that result locally in $X_{\{s_q,r_S\}}^{(k+1)}$. Algorithm 1 summarizes the resulting approach.

Algorithm 1: The model-parallel layer-$k$ inference at thread $q$.

**Input:** Local input $X_{\{r_S\}}^{(k)}$, separator part $X_{\{s_q,r_S\}}^{(k)}$, and weight matrix $W_{\{s_q,r_S\}}^{(k)}$.

**Output:** Local output $X_{\{s_q,r_S\}}^{(k+1)}$ and separator $X_{\{s_q,r_S\}}^{(k+1)}$.

1: for $r \leftarrow 0$ to $p-1$
2: $X_{\{s_q,r_S\}}^{(k+1)} \leftarrow f(X_{\{r_S\}}^{(k)}, W_{\{r_S\}}^{(k)}) + X_{\{s_q,r_S\}}^{(k+1)}$
3: $X_{\{s_q,r_S\}}^{(k+1)} \leftarrow X_{\{s_q,r_S\}}^{(k+1)}$
4: $X_{\{s_q,r_S\}}^{(k+1)} \leftarrow X_{\{s_q,r_S\}}^{(k+1)}$
5: $X_{\{s_q,r_S\}}^{(k+1)} \leftarrow X_{\{s_q,r_S\}}^{(k+1)}$ Without ReLU
6: exec barrier
7: $X_{\{s_q,r_S\}}^{(k+1)} \leftarrow f(\sum_{r=0}^{p-1} X_{\{s_q,r_S\}}^{(k+1)})$
8: exec barrier

There is no work overhead in flops as the summations at two-phase reduction correspond to summations of a sequential SpGEMM, now delayed due to having to reduce them over multiple threads. Ignoring the symbolic phase and assuming the $W_{\{r_S\}}^{(k)}$ can be cached, memory movement overhead is solely due to reading the input separator parts $X_{\{s_q,r_S\}}^{(k+1)}$ at Lines 3 and 5 as well as both writing and reading of partial results at Lines 5 and 7, respectively. This yields a data movement overhead of

$$\Theta \left( \sum_{k=0}^{d-1} \left[ \text{nz}(X_{\{s_q,r_S\}}^{(k+1)}) + \max_{q} \sum_{r=0}^{p-1} \text{nz}(X_{\{s_q,r_S\}}^{(k+1)}) + \sum_{r=0}^{p-1} \text{nz}(X_{\{s_q,r_S\}}^{(k+1)}) \right] + nph \right).$$

Comparing to the movement overhead of the data-parallel method, $\Theta(p \sum_{k=0}^{d-1} \text{nz}(W_{\{r_S\}}^{(k)}))$, the model-parallel variant is preferable whenever the combined size of all $X_{\{s_q,r_S\}}^{(k+1)}$ and $X_{\{s_q,r_S\}}^{(k+1)}$ is smaller than that of all $W_{\{r_S\}}^{(k)}$. The former combined size is proportional to the number of external columns, which is exactly the objective function (the cutnet metric) of the partitioning process. If the $W_{\{r_S\}}^{(k)}$ do not fit in cache, an additional $\Theta(\sum_{k=0}^{d-1} \max_{q} \text{nz}(W_{\{r_S\}}^{(k)} + |r_S| \cup |r_S|))$ data movement occurs, resulting in an additional overhead of $\Theta\left( (p \sum_{k=0}^{d-1} \max_{q} \text{nz}(W_{\{r_S\}}^{(k)} + \sum_{r=0}^{p-1} \text{nz}(W_{\{r_S\}}^{(k)})) \right)$, where $\varepsilon$ is the attained load imbalance of the partitioning.

The parallel storage overhead is bounded by the maximum number of extra nonzeros stored in the partial results $X_{\{s_q,r_S\}}^{(k+1)}$ plus the extra CRS indexing arrays for $pd$ local weight matrices, $p$ buffer matrices, and $p^2$ two-phase reduction buffers:

$$O \left( \max_{0 \leq k < d} \sum_{q=0}^{p-1} \left[ \text{nz}(X_{\{s_q,r_S\}}^{(k+1)}) + p(n + \sum_{k=1}^{d} |r_S|) \right] \right).$$

The synchronization overhead is $O(dpL)$.

D. Tiling model-parallel inference

Tiling refers to a technique which increases data reuse by prematurely pausing a loop’s iteration such that its output may be used by the subsequent iteration immediately and while they are still cached. Tiling techniques may incur overheads such as recomputation of factors needed to continue a paused iteration. We achieve a tiling in the model-parallel algorithm just presented by processing a batch of data elements through all layers of the network, before moving on to another batch. This allows the intermediate results $X^{(k)}$, $0 < k < d$, to remain in cache, which is especially effective when also the local parts of the neural network fit in cache.

To compute the tiling (model-parallel) inference, each thread $q$ invokes Algorithm 2 to run a model-parallel inference for each of the $t = \lfloor n/b_{size} \rfloor$ batches of $b_{size}$ rows of $X^{(k)}$. To reduce the barrier overhead of $O(dpL)$, each thread $q$ overlaps the computation of local results $X_{\{s_q,r_S\}}^{(k+1)}$ with the barrier, implemented as weak point-to-point synchronizations, before processing the contributions of remote threads $X_{\{s_q,r_S\}}^{(k+1)}$.

The costs of tiling inference are similar to that of model-parallel, with overheads multiplied by $t$ while substituting $b_{size}$ for $n$. This retains overheads proportional to $n$, and magnifies overheads proportional to $|r_S|^{(k)}$ and $|z_S|^{(k)}$ by $t$. Hence, there exists a trade-off between choosing higher block sizes that tile for smaller caches versus how well underlying neural networks can be partitioned. Since there is no explicit barrier, synchronization overheads lie far below $O(tdpL)$ in practice.

E. Hybrid tiling and deep inference

We may combine the data-parallel and tiling inference methods. Assume $p = p_0p_1$ threads, then hybrid inference splits $n$ into $p_0$ parts using a block distribution and processes each part using the tiling inference with $p_1$ threads. Thus, $p_0$ thread groups use the same $p_1$-way partitioned weights. We select the minimum $p_1$ for which weights fit in the combined local caches. Higher $p_1$ would only be sensible if the separator size would decrease, which normally is not the case. Choosing $p_0 \geq \lceil p/p_1 \rceil$ then enables use of (almost) all available cores.

Deep neural networks where weights do not fit local cache sizes can still benefit from tiling. First, we cut the network’s layers into successive blocks, and then apply our proposed tiling algorithm for each block. For example, if $d = 120$ and we create blocks of 5 layers, inference should perform 24 calls to the tiling algorithm. This requires streaming from main memory not only $X^{(0)}$, but all of $X^{(5k)}$, $0 \leq k < 24$. An automated method for selecting an appropriate cut could proceed greedily, ‘growing’ blocks of layers for which local parts remain cacheable and separator sizes remain below a
Algorithm 2 Tiling model-parallel inference at thread q.

1: for \( m \leftarrow 0 \) to \( \lfloor n/b_{\text{size}} \rfloor - 1 \) do
2:  \( b_m = \{mb_{\text{size}}, \ldots, \min((m+1)b_{\text{size}}, n) - 1\} \),
3:  \( \{s_0, \ldots, s_{p-1}\} \) be a block partitioning of \( b_m \) into \( p \) parts.
4:  Let \( C_q \) be a \( b_m \times n_{k+1} \) in-cache matrix
5:  Let \( D_q \) be \( (r_{(0)} \mod b_m) \) in-cache matrix
6:  \( D_q \leftarrow X_{\lfloor b_m \rfloor} \) \( \text{Read into cache} \)
7:  resetAvail\( (C_{l,\{r\}_{S}}, \text{resetAvail} (D_{s_{q},r_{S}})q \)
8:  for \( k = 0 \) to \( d \) do
9:     if \( k \neq 0 \) then \( \text{Inter-thread data movement} \)
10:        while done \( < \) \( p \) do
11:             if checkAvail\( (D_{s_{q},r_{S}})q \) then
12:                 \( D_{s_{q},r_{S}})q \leftarrow D_{s_{q},r_{S}})q + D_{s_{q},r_{S}})r \)
13:             done \( \leftarrow \) done \( + \) \( 1 \)
14:          \( r \leftarrow (q + 1) \mod p \)
15:          if \( r \) equals \( q \) then
16:             \( r \leftarrow (q + 1) \mod p \)
17:     \( D_{s_{q},r_{S}})q \leftarrow f(D_{s_{q},r_{S}})q \) \( \text{Delayed ReLU} \)
18:     resetAvail\( (C_{s_{q},r_{S}})q, \text{signalAvail} (D_{s_{q},r_{S}})q \)
19:  if \( k \neq 0 \) then \( \text{Intra- and inter-thread data movement} \)
20:         \( r = q \), done \( = \) \( 0 \)
21:  while done \( < \) \( p \) do
22:     if checkAvail\( (D_{s_{q},r_{S}})q \) then
23:          \( C_{s_{q},r_{S}})q \leftarrow f(D_{s_{q},r_{S}})q W_{s_{q},r_{S}})q + D_{s_{q},r_{S}})q \)
24:          \( D_{s_{q},r_{S}})q \leftarrow D_{s_{q},r_{S}})q + D_{s_{q},r_{S}})q \)
25:          \( C_{s_{q},r_{S}})q \leftarrow D_{s_{q},r_{S}})q W_{s_{q},r_{S}})q + D_{s_{q},r_{S}})q \)
26:          \( D_{s_{q},r_{S}})q \leftarrow D_{s_{q},r_{S}})q \) \( \text{Without ReLU} \)
27:          done \( \leftarrow \) done \( + \) \( 1 \)
28:          \( r \leftarrow (r + 1) \mod p \)
29:     resetAvail\( (D_{s_{q},r_{S}})q, \text{signalAvail} (C_{s_{q},r_{S}})q \)
30:        \( \text{Swap} C \) and \( D \)
31:     \( X_{\{d\}_{s_{q},r_{S}})q} \leftarrow D_{s_{q},r_{S}})q \) \( \text{Write back to memory} \)
32:  \( X_{\{d\}_{s_{q},r_{S}})q} \leftarrow D_{s_{q},r_{S}})q \)
33:

We use single-precision floats for values and integers for indices, following earlier challenge submissions [8]. Our implementation builds on an internal C++ GraphBLAS code base used in sequential mode ensuring that threads allocate data using a local allocation policy enforced by the libnuma library. OpenMP is used for parallelization together with a custom ANSI C module that implements the (almost) synchronization-free mechanism of our tiling method. Code compiles using GCC 9.2.0. Both machines run Linux with kernel version 3.10.0 on Ivy Bridge and 5.4.0 on Cascade Lake.

The performance of the tiling inference exceeds that of the data-parallel algorithm only if the feature matrix \( X^{(0)} \) and weights \( W^{(k)} \) do not fit cache; otherwise, the same beneficial cache effects of tiling naturally take place for the data-parallel algorithm. For the tiling methods, we take the maximum tile size for which all thread-local tiles of \( X^{(k)} \) remain in L3 cache, or both the thread-local tiles of \( X^{(k)} \) and \( W^{(k)} \) for all \( k \) remain in L3 cache if the combined network does not fit L3 cache.

The density of \( C \) and \( D \) are unknown and change as inference proceeds. We assume they are dense and correct the resulting storage requirements using the density of the input matrix \( X^{(0)} \) for the density parameter \( \alpha \leq 1 \). This accounts for the density of \( X^{(k)} \) assuming it does not grow with \( k \). We ensure that \( b_{\text{size}} > 0 \) is divisible by \( p_1 \).

We experiment on the first five layers of the 120-layer HPEC networks: 1) to confirm the beneficial effects of caching the intermediate results, and 2) to confirm our understanding of the performance characteristics of the proposed tiling and hybrid tiling inference methods. As the \( X^{(k)} \) fit in cache, we expect no benefit from the tiling method versus our data-parallel baseline implementation on the Ivy Bridge machine for both the 1k-neuron and 4k-neuron HPEC neural networks (where additionally the \( X^{(k)} \) have decreasing densities as \( k \) increases, see Table I). Indeed, the results in Table III confirm slowdowns of the tiling method versus the data-parallel baseline, which grow with \( p \) due to the parallel overhead increasing with the separator size. We omit the results for the 4k-neuron network as the same effect is observed.

For the 16k- and 64k-neuron networks, the \( X^{(0)} \) do not fit in cache. Due to decreasing density, however, it may be that later \( X^{(k)} \) could fit; this would be beneficial to data-parallel inference. Regardless, we certainly expect better performance from our tiling variant for the first five layers. For increasing \( p \), since the parallel overhead grows with the number of external rows and columns, we expect this benefit to decrease. Table III confirms both expected behaviors.

Having done pure tiling for various \( p \), we are able to select suitable \( p_1 \) equal to the best performing \( p \). We use the remaining threads for the hybrid inference. Table IV shows the results for the 5-layer 16k- and 64k-neuron network. These confirm that the hybrid tiling inference scales, maintaining similar speedups versus the data-parallel baseline as for the pure tiling method with \( p = p_1 \) threads. From Table IV, we additionally conclude that using hyperthreads benefits both the data-parallel and tiling methods, in the latter case presumably made possible by the almost synchronization-free method.

certain threshold, using the hybrid method to deal with case where this method ends up with relatively low \( p_1 \).

More elaborate schemes may be envisioned; whatever these secondary details may be, a blocked tiling approach as presented here should lie at the core of a competitive sparse neural network inference method. Assuming an appropriate partitioning, a blocked tiling inference identifies two parameters: a block size for which the combined consecutive layers fit cache, and a tile size for which the intermediate results can be cached as well. We proceed with a series of experiments demonstrating this general approach works for the Graph Challenge dataset.

III. EXPERIMENTS

We use two machines for experiments: an Ivy Bridge node consisting of two sockets each equipped with ten cores, and a Cascade Lake node with two sockets each with 22 cores. The L2 cache size on the Ivy Bridge node is 256 kB while on the Cascade Lake it is 1 MB per core. Their L3 cache sizes are 2.5 MB and 1.25 MB per core, respectively. Both have 32 KB of L1 data cache size per core.
employed between (hyper)threads.

The fastest on the first five HPEC layers using the full Ivy Bridge machine are obtained by our hybrid tiling method, which are 41% faster versus the data-parallel method for the 16k-neuron network, and 50% faster on the 64k one.

Finally, we repeat both experiments for the Cascade Lake machine in Tables V and VI. For the hybrid tiling experiments, we used less than the available cores per socket due to divisibility. The results in both tables confirm the same expected behavior, and thus show that the method applies to all architectures. On Cascade Lake, the fastest inference on the first five HPEC layers using 101% of the 16k- and 64k-neuron networks.

<table>
<thead>
<tr>
<th>Network</th>
<th>p</th>
<th>p₀</th>
<th>p₁</th>
<th>Time (in s)</th>
<th>Tiling (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>3.66</td>
<td>2.20 (1.66)</td>
</tr>
<tr>
<td>16k</td>
<td>40</td>
<td>4</td>
<td>2</td>
<td>2.56</td>
<td>1.81 (1.41)</td>
</tr>
<tr>
<td>64k</td>
<td>20</td>
<td>4</td>
<td>5</td>
<td>12.58</td>
<td>7.62 (1.65)</td>
</tr>
<tr>
<td>64k</td>
<td>40</td>
<td>8</td>
<td>5</td>
<td>9.72</td>
<td>6.49 (1.50)</td>
</tr>
</tbody>
</table>

Table IV

The hybrid tiling inference results over the first 5 layers of 16k-, and 64k-neuron networks on Ivy Bridge.

<table>
<thead>
<tr>
<th>Network</th>
<th>p</th>
<th>p₀</th>
<th>p₁</th>
<th>Time (in s)</th>
<th>Tiling (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>4.65</td>
<td>2.61 (1.78)</td>
</tr>
<tr>
<td>16k</td>
<td>22</td>
<td>12</td>
<td>2</td>
<td>4.49</td>
<td>2.23 (2.01)</td>
</tr>
<tr>
<td>16k</td>
<td>24</td>
<td>6</td>
<td>4</td>
<td>3.96</td>
<td>2.19 (1.81)</td>
</tr>
<tr>
<td>16k</td>
<td>44</td>
<td>22</td>
<td>2</td>
<td>2.36</td>
<td>1.18 (2.00)</td>
</tr>
<tr>
<td>16k</td>
<td>44</td>
<td>11</td>
<td>2</td>
<td>2.36</td>
<td>1.63 (1.45)</td>
</tr>
<tr>
<td>64k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>16.99</td>
<td>9.06 (1.88)</td>
</tr>
<tr>
<td>64k</td>
<td>22</td>
<td>11</td>
<td>2</td>
<td>15.85</td>
<td>8.49 (1.87)</td>
</tr>
<tr>
<td>64k</td>
<td>40</td>
<td>8</td>
<td>5</td>
<td>8.91</td>
<td>4.75 (1.88)</td>
</tr>
<tr>
<td>64k</td>
<td>44</td>
<td>22</td>
<td>2</td>
<td>8.68</td>
<td>4.47 (1.94)</td>
</tr>
<tr>
<td>64k</td>
<td>44</td>
<td>4</td>
<td>11</td>
<td>8.68</td>
<td>5.10 (1.70)</td>
</tr>
</tbody>
</table>

Table V

The hybrid tiling inference results over the first 5 layers of 16k-, and 64k-neuron networks on Cascade Lake.

The fastest on the first five HPEC layers using the full Ivy Bridge machine are obtained by our hybrid tiling method, which are 41% faster versus the data-parallel method for the 16k-neuron network, and 50% faster on the 64k one.

Finally, we repeat both experiments for the Cascade Lake machine in Tables V and VI. For the hybrid tiling experiments, we used less than the available cores per socket due to divisibility. The results in both tables confirm the same expected behavior, and thus show that the method applies to all architectures. On Cascade Lake, the fastest inference on the first five HPEC layers using 101% of the 16k- and 64k-neuron networks.

<table>
<thead>
<tr>
<th>Network</th>
<th>p</th>
<th>p₀</th>
<th>p₁</th>
<th>Time (in s)</th>
<th>Tiling (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>3.66</td>
<td>2.20 (1.66)</td>
</tr>
<tr>
<td>16k</td>
<td>40</td>
<td>4</td>
<td>2</td>
<td>2.56</td>
<td>1.81 (1.41)</td>
</tr>
<tr>
<td>64k</td>
<td>20</td>
<td>4</td>
<td>5</td>
<td>12.58</td>
<td>7.62 (1.65)</td>
</tr>
<tr>
<td>64k</td>
<td>40</td>
<td>8</td>
<td>5</td>
<td>9.72</td>
<td>6.49 (1.50)</td>
</tr>
</tbody>
</table>

Table IV

The hybrid tiling inference results over the first 5 layers of 16k-, and 64k-neuron networks on Ivy Bridge.

<table>
<thead>
<tr>
<th>Network</th>
<th>p</th>
<th>p₀</th>
<th>p₁</th>
<th>Time (in s)</th>
<th>Tiling (speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>4.65</td>
<td>2.61 (1.78)</td>
</tr>
<tr>
<td>16k</td>
<td>22</td>
<td>12</td>
<td>2</td>
<td>4.49</td>
<td>2.23 (2.01)</td>
</tr>
<tr>
<td>16k</td>
<td>24</td>
<td>6</td>
<td>4</td>
<td>3.96</td>
<td>2.19 (1.81)</td>
</tr>
<tr>
<td>16k</td>
<td>44</td>
<td>22</td>
<td>2</td>
<td>2.36</td>
<td>1.18 (2.00)</td>
</tr>
<tr>
<td>16k</td>
<td>44</td>
<td>11</td>
<td>2</td>
<td>2.36</td>
<td>1.63 (1.45)</td>
</tr>
<tr>
<td>64k</td>
<td>20</td>
<td>5</td>
<td>4</td>
<td>16.99</td>
<td>9.06 (1.88)</td>
</tr>
<tr>
<td>64k</td>
<td>22</td>
<td>11</td>
<td>2</td>
<td>15.85</td>
<td>8.49 (1.87)</td>
</tr>
<tr>
<td>64k</td>
<td>40</td>
<td>8</td>
<td>5</td>
<td>8.91</td>
<td>4.75 (1.88)</td>
</tr>
<tr>
<td>64k</td>
<td>44</td>
<td>22</td>
<td>2</td>
<td>8.68</td>
<td>4.47 (1.94)</td>
</tr>
<tr>
<td>64k</td>
<td>44</td>
<td>4</td>
<td>11</td>
<td>8.68</td>
<td>5.10 (1.70)</td>
</tr>
</tbody>
</table>

Table V

The hybrid tiling inference results over the first 5 layers of 16k-, and 64k-neuron networks on Cascade Lake.

Optimizations such as the use of algorithms for sparse matrix–dense matrix multiplication may be used when the compressed density reaches 100% during inference. Another potential improvement relates to the use of the cutnet metric: it should be possible to achieve overheads proportional to the $\lambda - 1$-metric instead, since similar bounds were achieved in earlier work on sparse matrix–vector multiplication [18]. Allowing permutations between layers, which can also be modeled via hypergraphs [20], may allow for further decreased separator size. Work is in progress to efficiently apply the proposed hybrid tiling method, potentially with the mentioned optimizations, on batches of input layers successively. This will confirm the applicability and benefits of the proposed approach on a broader set of neural network inference tasks.
REFERENCES


