Combinatorial Tiling for Sparse Neural Networks

Filip Pawlowski, Rob H Bisseling, Bora Uçar, Albert-Jan N Yzelman

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
§ Department of Mathematics
Utrecht University, the Netherlands
R.H.Bisseling@uu.nl
¶ CNRS and LIP (UMR5668),
Lyon, France
bora.ucar@ens-lyon.fr

***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 $2\times$ 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 as 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 $d \in \mathbb{N}$ 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 $X^{(0)}$:

$$\begin{aligned} X^{(d)} = f(\cdots f(f(X^{(0)}W^{(0)} + \mathbf{e}_n b^{(0)^T})W^{(1)} \\ + \mathbf{e}_n b^{(1)^T}) \cdots W^{(d-1)} + \mathbf{e}_n b^{(d-1)^T}) \,, \end{aligned} \tag{1}$$

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 $\mathbf{e}_n = (1, \ldots, 1)^T$.

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 $X, Y \in \mathbb{R}^{m \times n}$,

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

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 nonzeros 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

Number of neurons	Input matrix	Number of layers d 120	480	1920
1k	48.86	30.47	121.88	487.51
4k	191.11	121.88	487.50	1950.01
16k	754.46	487.50	1950.00	7800.01
64k	2992.42	1950.00	7800.00	31200.01

Table I

THE SIZES (IN MB) OF THE INPUT FEATURE MATRIX $X^{(0)}$ AND THE HPEC NEURAL NETWORKS WITH DIFFERENT NUMBERS OF LAYERS AND NEURONS.

Layer k	Compressed density	Density	Memory (in MB)
0	10.38	10.38	136.70
1	31.08	28.33	149.15
2	20.65	13.94	80.99
4	34.28	6.86	45.98
6	62.56	4.26	29.07
8	97.23	3.45	23.63
10	99.93	3.12	21.93
15	100.00	3.02	21.46

Table II

THE COMPRESSED DENSITY AND THE DENSITY OF FEATURE MATRICES $X^{(k)}$. 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 f HAS BEEN APPLIED AFTER ALL LAYERS, EXCEPT FOR $X^{(0)}$.

g and the inter-socket cost h, both in seconds per byte. We write L (in seconds) for the time to complete a barrier.

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 $\frac{\mathrm{nz}(A)}{mn}$, and the *compressed density* be $\frac{\mathrm{nz}(A)}{m'n}$, where $\mathrm{nz}(\cdot)$ 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 the feature matrices $X^{(k)}$ for a few of the first fifteen layers of the network, as well as the size of the memory touched by the SpGEMM operation processing that layer. The density of feature matrices continuously decreases from the second layer onwards and remains constant at 3.02% after the 14th layer. On the other hand, the compressed density increases and reaches 100%, and remains so after layer 14. We believe that compressed densities tend to hundred percent in most neural networks beyond RadiX-Net since classification tends to result in nonzero probabilities for all classes.

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 $\Theta(\sum_{i=0}^{n-1} \mathrm{nz}(X^{(k)}_{\{:,i\}}) \,\mathrm{nz}(W^{(k)}_{\{i,:\}}))$. The CRS matrices of $X^{(k)}$ and $X^{(k+1)}$ are streamed to the CPU only once, which incurs a data movement of $\Theta(\mathrm{nz}\,(X^{(k)}) + \mathrm{nz}\,(X^{(k+1)}) + n)$. The CRS of $W^{(k)}$ as well as the sparse accumulator structures (SPA and SPAC of size $\Theta(\max_k n_{k+1})$) are accessed $\Theta(\mathrm{nz}(X^{(k)}))$ 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 $\Theta\left(\sum_{k=0}^{d-1}\left[\sum_{i=0}^{n_k-1} \mathrm{nz}(X^{(k)}_{\{:,i\}}) \,\mathrm{nz}(W^{(k)}_{\{i,:\}})\right]\right)$. 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

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

If $W^{(k)}$ cannot be cached this increases with $\mathcal{O}(\sum_{k=0}^{d-1}\sum_{i=0}^{n_k-1} \mathrm{nz}(X^{(k)}_{\{:,i\}}) \,\mathrm{nz}(W^{(k)}_{\{i,:\}})g)$. 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

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

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.

$$\begin{pmatrix} {W^{(0)}}^T & W^{(1)} & 0 & \cdots & 0 \\ 0 & {W^{(2)}}^T & W^{(3)} & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & {W^{(d-4)}}^T & W^{(d-3)} & 0 \\ 0 & \cdots & 0 & {W^{(d-2)}}^T & W^{(d-1)} \end{pmatrix}$$

Figure 1. The staircase matrix of a neural network consisting of d layers. (Here, d is even.)

A. Hypergraph models for inference

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of hyperedges $\mathcal{N}$. A hyperedge (a net) $h \in \mathcal{N}$ is a subset of vertices, $h \subseteq \mathcal{V}$. The hypergraph partitioning problem [15] is the task of dividing the vertices of a hypergraph into K parts of roughly equal sizes, while minimizing an objective (cost) function defined in terms of nets; the most common variants of this problem are NP-complete. That is one seeks a partition $\Pi_{\mathcal{V}} = \{\mathcal{V}_0, \ldots, \mathcal{V}_{K-1}\}$ of the vertex set of $\mathcal{H}$. This simultaneously induces a $(K+1)$-way partition on the nets, $\Pi_{\mathcal{N}} = \{\mathcal{N}_0, \ldots, \mathcal{N}_{K-1}; \mathcal{N}_S\}$, where nets in $\mathcal{N}_i$ for $0 \leq i < K$ have vertices only in $\mathcal{V}_i$ and are *internal*, while nets in $\mathcal{N}_S$ have vertices in more than one part and are *external*. The connectivity λ_h of a net h is equal to the number of parts it connects. A partition $\Pi_{\mathcal{V}}$ is said to be balanced if for each part $\mathcal{V}_k$, $w(\mathcal{V}_k) \leq w(\mathcal{V}/K)(1+\varepsilon)$, where ε is a constraint to the partitioner, and w is the amount of work involved (usually measured as nonzeros in a corresponding matrix). Çatalyürek and Aykanat [16] proposed a fine-grain hypergraph model for sparse matrices A, under which each nonzero of A becomes a unique vertex while each column and row of A corresponds to a net consisting of the nonzeros it contains.

We contribute a hypergraph model of the whole neural network inference process which can be interpreted as applying the fine-grain model on the staircase matrix defined in Figure 1. This matrix connects the consecutive layers $W^{(k)}$ into a staircase, thus preserving the relationship between the individual layers. A partitioning of this hypergraph results in a distribution of weight matrices, thus defining which thread processes which part of the neural network. We do this under two constraints: 1) a load balance criterion on weights in each individual layer, which ensures threads have a roughly equal amount of work when collaboratively processing a layer, and 2) minimization of the number of external rows and columns in $\mathcal{N}_S$ (which is known as the cutnet metric [17]). We use PaToH [17], a multi-constraint hypergraph partitioner, to obtain such a balanced K-partitioning of the staircase matrix.

It may be that a column of $W^{(i)}$ is internal while its corresponding row of $W^{(i+1)}$ is external, or vice versa. This creates dependencies between internal and external parts of the resulting partitioning of layers. Handling these efficiently during inference is possible using known techniques (see, e.g., Yzelman and Roose [18]); we save that exercise for future work, however. Instead and for simplicity, we here artificially move the internal rows or columns to the external set $\mathcal{N}_S$ so that such conflicts will not arise in the subsequent presentation.

B. Data-parallel inference

Data-parallel inference partitions the feature matrix $X^{(0)} \in \mathbb{R}^{n \times n_0}$ into p parts using a row-wise partitioning. We map row i to thread $\lfloor i/\lceil n/p \rceil \rfloor$ 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 $X^{(k)}_{\{s_q,:\}}$ and $X^{(k+1)}_{\{s_q,:\}}$ 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 $\Omega(\sum_{k=0}^{d-1}(\mathrm{nz}\,(W^{(k)}) + n_k)(p_t g + (p - p_t)h))$ as all threads need to refer to the shared weights; replication of the weights on each socket would reduce this to $\Omega(\sum_{k=0}^{d-1}(\mathrm{nz}\,(W^{(k)}) + n_k)p_t g)$. 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 $W^{(k)}_q$ be the matrix consisting of the set of weights of $W^{(k)}$ distributed to thread q, and $\mathbf{R}^{(k)} = \{r^{(k)}_0, \ldots, r^{(k)}_{p-1}; r^{(k)}_S\}$ be a partitioning of row indices of $W^{(k)}$ such that all weights on rows in $R^{(k_i)}$ are in $W^{(k_i)}$ (internal rows) while $R^{(k_S)}$ are rows shared by multiple parts of $\mathbf{W}^{(k)}$ (external rows). Let $\mathbf{Z}^{(k)} = \{z^{(k)}_0, \ldots, z^{(k)}_{p-1}; z^{(k)}_S\}$ be a similar partitioning of columns, and let $\lambda^{(k)}_i$ and $\lambda^{(k)}_j$ refer to the connectivities of row i and column j, respectively. When the layer number k can be inferred from context we shall omit the superscripts from the $r^{(k)}_q$, $z^{(k)}_q$, and $\lambda^{(k)}_{\{i,j\}}$.

The matrix $W^{(k)}_q$ is stored locally by thread q, as are the corresponding input and output feature matrices $X^{(k)}_{\{:,r_q \cup r_S\}}$ and $X^{(k+1)}_{\{:,z_q \cup z_S\}}$ Since the column sets $r^{(k)}_q, z^{(k)}_q$ are disjoint, each thread q stores a disjoint set of features. The parts corresponding to separator columns $r^{(k)}_S, z^{(k)}_S$, however, are replicated across all threads. Since a thread q may refer to remote data such as the separator features $X^{(k)}_{\{s_q,r_S\}_r}$ 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 $W^{(k)}_q$ can be logically subdivided into four parts from $W^{(k)}_{\{r_q \cup r_S, z_q \cup z_S\}_q}$, each with different data dependencies: (i) processing $W^{(k)}_{\{r_q,z_q\}_q}$ only depends on the locally computed data $X^{(k)}_{\{:,r_q\}_q}$; (ii) processing $W^{(k)}_{\{r_S,z_q\}_q}$ requires collaboratively computed data $X^{(k)}_{\{s_q,r_S\}_r}$ for all r; (iii) processing $W^{(k)}_{\{r_q,z_S\}_q}$ requires collaborative computation of the output $\sum_{r=0}^{p-1} X^{(k+1)}_{\{:,z_S\}_r}$; and (iv) processing $W^{(k)}_{\{r_S,z_S\}_q}$

requires both parallel read access on $X^{(k)}_{\{s_q,r_S\}_r}$ for all r and collaborative computation of $\sum_{r=0}^{p-1} X^{(k+1)}_{\{:,z_S\}_r}$. We assume the number of rows n is larger than or equal to p and employ two-phase reduction to collaboratively compute $\sum_{r=0}^{p-1} X^{(k+1)}_{\{:,z_S\}_r}$. 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 $X^{(k+1)}_{\{s_q,z_S\}} = \sum_{r=0}^{p-1} X^{(k+1)}_{\{s_q,z_S\}_r}$ and stores that result locally in $X^{(k+1)}_{\{s_q,z_S\}_q}$. Algorithm 1 summarizes the resulting approach.

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

Input: Local input $X^{(k)}_{\{:,r_q\}_q}$, separator part $X^{(k)}_{\{s_q,r_S\}_q}$, and weight matrix $W^{(k)}_{\{r_q\cup r_S,z_q\cup z_S\}_q}$.

Output: Local output $X^{(k+1)}_{\{:,z_q\}_q}$ and separator $X^{(k+1)}_{\{s_q,z_S\}_q}$.

1: **for** $r \leftarrow 0$ to $p-1$ **do**
2: $\quad X^{(k+1)}_{\{s_r,z_q\}_q} \leftarrow f(X^{(k)}_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_q\}_q} +$
3: $\qquad X^{(k)}_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_q\}_q})$
4: $\quad X^{(k+1)}_{\{s_r,z_S\}_q} \leftarrow X^{(k)}_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_S\}_q} +$
5: $\qquad X^{(k)}_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_S\}_q}$ ▶ Without ReLU
6: **exec** barrier ▶ Only for $k \neq 0$
7: $X^{(k+1)}_{\{s_q,z_S\}_q} \leftarrow f(\sum_{r=0}^{p-1} X^{(k+1)}_{\{s_q,z_S\}_r})$
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 $W^{(k)}_q$ can be cached, memory movement overhead is solely due to reading the input separator parts $X^{(k)}_{\{s_r,r_S\}_r}$ 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

$$\Theta\left(\sum_{k=0}^{d-1}\Big[\mathrm{nz}(X^{(k)}_{\{:,r_S\}}) + \max_q \sum_{\substack{r=0,\\ r\neq q}}^{p-1} \mathrm{nz}(X^{(k+1)}_{\{s_q,z_S\}_r}) + n\Big]ph\right).$$

Comparing to the movement overhead of the data-parallel method, $\Theta(p\sum_{k=0}^{d-1}\mathrm{nz}(W^{(k)})h)$, the model-parallel variant is preferable whenever the combined size of all $X^{(k)}_{\{:,r_S\}}$ and $X^{(k+1)}_{\{:,z_S\}}$ is smaller than that of all $W^{(k)}$. 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 $W^{(k)}_q$ do not fit in cache, an additional $\Theta(\sum_{k=0}^{d-1}\max_q[\mathrm{nz}(W^{(k)}_q) + |r_q \cup r_S|]g)$ data movement occurs, resulting in an additional overhead of $\Theta(((p\sum_{k=0}^{d-1}\max_q|r_q| + |r_S| - n_k/p) + \varepsilon\sum_k \mathrm{nz}(W^{(k)}))g)$, where ε is the attained load imbalance of the partitioning.

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

$$\mathcal{O}\left(\max_{0\leq k<d}\sum_{q=0}^{p-1}\left[\mathrm{nz}(X^{(k+1)}_{\{:,z_S\}_q})\right] + p(n + \sum_{k=1}^{d}|r_s|)\right).$$

The synchronization overhead is $\Theta(dpL)$.

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 $t = \lceil n/b_{size}\rceil$ batches of b_{size} rows of $X^{(k)}$. To reduce the barrier overhead of $\Theta(dpL)$, each thread q overlaps the computation of local results $X^{(k+1)}_{\{:,z_q\}_q}$ with the barrier, implemented as weak point-to-point synchronizations, before processing the contributions of remote threads $X^{(k)}_{\{s_r,r_S\}_{r\neq q}}$.

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 $|r^{(k)}_S|$ and $|z^{(k)}_S|$ 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 $\mathcal{O}(tdpL)$ 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 $p_0 \geq \lfloor p/p_1\rfloor$ 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.

for $m \leftarrow 0$ to $\lceil n/b_{size} \rceil - 1$ **do**
  Let $b_m = \{mb_{size}, \ldots, \min\{(m+1)b_{size}, n\} - 1\}$.
  Let $\{s_0, \ldots, s_{p-1}\}$ be a block partitioning of b_m into p parts.
  Let C_q be a $b_m \times n_{k+1}$ in-cache matrix
  Let D_q be a $b_m \times n_k$ in-cache matrix
  $D_q \leftarrow X^{(0)}_{\{b_m,:\}}$ ▶ Read into cache
  $resetAvail(C_{\{:,r_S\}_q})$, $resetAvail(D_{\{s_q,r_S\}_q})$
  for $k = 0$ to d **do**
    if $k \neq 0$ **then** ▶ Inter-thread data movement
      $r = (q+1) \bmod p$, $done = 1$
      while $done < p$ **do**
        if $checkAvail(D_{\{:,r_S\}_r})$ **then**
          $D_{\{s_q,r_S\}_q} \leftarrow D_{\{s_q,r_S\}_q} + D_{\{s_q,r_S\}_r}$
          $done \leftarrow done + 1$
        $r \leftarrow (r+1) \bmod p$
        if r equals q **then**
          $r = (q+1) \bmod p$
      $D_{\{s_q,r_S\}_q} \leftarrow f(D_{\{s_q,r_S\}_q})$ ▶ Delayed ReLU
    $resetAvail(C_{\{s_q,r_S\}_q})$, $signalAvail(D_{\{s_q,r_S\}_q})$
    if $k \neq d$ **then** ▶ Intra- and inter-thread data movement
      $r = q$, $done = 0$
      while $done < p$ **do**
        if $checkAvail(D_{\{s_r,r_S\}_r})$ **then**
          $C_{\{s_r,z_q\}_q} \leftarrow f(D_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_q\}_q} +$
            $D_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_q\}_q})$
          $C_{\{s_r,z_S\}_q} \leftarrow D_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_S\}_q} +$
            $D_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_S\}_q}$ ▶ Without ReLU
          $done \leftarrow done + 1$
        $r \leftarrow (r+1) \bmod p$
    $resetAvail(D_{\{:,r_S\}_q})$, $signalAvail(C_{\{:,r_S\}_q})$
    Swap C and D
  $X^{(d)}_{\{b_m,z_q\}} \leftarrow D_{\{:,z_q\}_q}$ ▶ Write back to memory
  $X^{(d)}_{\{s_q,z_S\}} \leftarrow D_{\{s_q,z_S\}_q}$

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.

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 $W^{(k)}_q$ 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 $b_{size} > 0$ 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

	1k-neuron				16k-neuron				64k-neuron			
	Separator	Tile	Time (in s)		Separator	Tile	Time (in s)		Separator	Tile	Time (in s)	
p	(in %)	size	Baseline	Tiling (speedup)	(in %)	size	Baseline	Tiling (speedup)	(in %)	size	Baseline	Tiling (speedup)
2	11.79	11392	2.01	2.22 (0.91)	0.80	158	33.62	19.86 (1.69)	2.08	220	119.90	67.64 (1.77)
3	20.61	8067	1.32	1.63 (0.81)	5.37	117	22.07	14.77 (1.49)	2.41	210	77.99	46.05 (1.69)
4	16.25	6628	1.02	1.32 (0.77)	4.89	116	16.92	10.35 (1.63)	4.55	200	61.57	34.57 (1.78)
5	34.79	4575	0.81	1.55 (0.52)	6.02	95	13.44	10.01 (1.34)	1.69	200	47.22	27.65 (1.71)
6	26.67	4278	0.68	1.09 (0.62)	4.24	114	11.60	7.82 (1.48)	2.53	180	39.24	23.91 (1.64)
7	40.70	3304	0.58	1.15 (0.50)	7.34	63	9.55	6.80 (1.40)	7.79	140	33.53	24.75 (1.35)
8	21.25	3160	0.53	0.79 (0.67)	7.97	72	9.78	5.87 (1.67)	3.68	160	32.64	18.53 (1.76)
9	41.81	2601	0.46	0.99 (0.46)	6.77	81	7.56	6.81 (1.11)	3.26	180	26.28	17.58 (1.49)
10	42.71	2500	0.42	0.83 (0.51)	8.27	90	7.00	5.37 (1.30)	5.23	100	24.45	17.93 (1.36)
20	46.16	1340	0.22	0.53 (0.53)	16.74	180	3.66	3.87 (0.95)	6.17	200	12.58	10.06 (1.25)

Table III

THE TILING INFERENCE RESULTS OVER THE FIRST 5 LAYERS OF 1K-, 16K-, AND 64K-NEURON NETWORKS ON IVY BRIDGE. THE SEPARATOR IS GIVEN IN PERCENTAGE OF THE TOTAL INPUT AND OUTPUT COLUMN SIZE. SPEEDUPS ARE RELATIVE TO THE DATA-PARALLEL BASELINE.

				Time (in s)	
Network	p	p_0	p_1	Baseline	Tiling (speedup)
16k	20	5	4	3.66	2.20 (1.66)
16k	40	20	2	2.56	1.81 (1.41)
64k	20	4	5	12.58	7.62 (1.65)
64k	40	8	5	9.72	6.49 (1.50)

Table IV

THE HYBRID TILING INFERENCE RESULTS OVER THE FIRST 5 LAYERS OF 16K-, AND 64K-NEURON NETWORKS ON IVY BRIDGE.

				Time (in s)	
Network	p	p_0	p_1	Baseline	Tiling (speedup)
16k	20	5	4	4.65	2.61 (1.78)
16k	22	11	2	4.49	2.23 (2.01)
16k	24	6	4	3.96	2.19 (1.81)
16k	44	22	2	2.36	1.18 (2.00)
16k	44	4	11	2.36	1.63 (1.45)
64k	20	5	4	16.99	9.06 (1.88)
64k	22	11	2	15.85	8.49 (1.87)
64k	40	5	8	8.91	4.75 (1.88)
64k	44	22	2	8.68	4.47 (1.94)
64k	44	4	11	8.68	5.10 (1.70)

Table V

THE HYBRID TILING INFERENCE RESULTS OVER THE FIRST 5 LAYERS OF 16K-, AND 64K-NEURON NETWORKS ON CASCADE LAKE.

	16k-neuron			64k-neuron		
	Tile	Time (in s)		Tile	Time (in s)	
p	size	Baseline	Tiling (speedup)	size	Baseline	Tiling (speedup)
2	158	43.53	23.05 (1.89)	220	166.34	82.58 (2.01)
3	117	28.80	17.28 (1.67)	210	103.39	56.03 (1.85)
4	116	21.84	12.27 (1.78)	200	79.19	42.03 (1.88)
5	95	17.94	11.93 (1.50)	200	62.33	33.29 (1.87)
6	114	15.10	9.33 (1.62)	180	52.40	28.63 (1.83)
7	63	12.80	8.29 (1.54)	140	45.48	31.42 (1.45)
8	72	12.17	7.32 (1.66)	160	42.87	22.54 (1.90)
9	81	10.47	8.07 (1.30)	180	34.88	21.04 (1.66)
10	90	9.16	6.52 (1.40)	100	32.36	21.86 (1.48)
11	99	9.10	6.28 (1.45)	110	32.24	19.90 (1.62)
16	144	6.73	5.26 (1.28)	160	23.55	12.50 (1.88)
22	198	4.73	5.20 (0.91)	220	15.85	13.41 (1.18)

Table VI

THE TILING INFERENCE RESULTS OVER THE FIRST 5 LAYERS OF 16K-, AND 64K-NEURON NETWORKS ON CASCADE LAKE.

employed between (hyper)threads.

The fastest inference 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 took various combinations of $p_0 p_1$ some of which result in using less than the 22 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 different architectures. On Cascade Lake, the fastest inference is 101% and 94% faster compared to the data-parallel method for the 16k- and 64k-neuron experiment, respectively, using all 22 cores with hyperthreads in a $22 \cdot 2$ configuration.

IV. CONCLUSIONS

We propose a hypergraph model of sparsely connected neural networks by applying the fine-grain model on the staircase matrix. This matrix defines a block structure of all weight matrices that accurately captures dependencies and communication between layers. By applying multi-constraint load balance criteria we use this model to partition the layers while minimizing overhead during model-parallel inference.

We then propose an efficient, hybrid model- and data-parallel sparse neural network inference method that performs tiling through layers to increase cache reuse, and exploits weak synchronization to hide load imbalance and inter-thread synchronization costs. The hybrid tiling algorithm offers best results for inference on the first five layers of the deep RadiX-Net NNs, provided that the input feature matrix is sufficiently large as confirmed using the 16k- and 64k-neuron networks. We demonstrated the approach works across architectures.

Optimizations such as the use of algorithms for sparse matrix–dense matrix [19] 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

[1] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse Deep Neural Network Graph Challenge," *arXiv e-prints*, p. arXiv:1909.05631, Sep 2019.

[2] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "The GraphBLAS C API specification," *GraphBLAS. org, Tech. Rep.*, 2017.

[3] B. van der Lugt and R. H. Bisseling, "Banded Sparse Neural Networks and their parallel computation," 2018, report.

[4] S. Alford, R. Robinett, L. Milechin, and J. Kepner, "Pruned and structurally sparse neural networks," *CoRR*, vol. abs/1810.00299, 2018. [Online]. Available: http://arxiv.org/abs/1810.00299

[5] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[6] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[7] M. Bisson and M. Fatica, "A GPU implementation of the sparse deep neural network graph challenge," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2019.

[8] T. Davis, M. Aznaveh, and S. Kolodziej, "Write quick, run fast: Sparse deep neural network in 20 minutes of development time in SuiteSparse:GraphBLAS," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2019.

[9] J. A. Ellis and S. Rajamanickam, "Scalable inference for sparse deep neural networks using Kokkos kernels," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2019.

[10] X. Wang, Z. Lin, C. Yang, and J. D. Owens, "Accelerating DNN inference with GraphBLAS and the GPU," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2019.

[11] J. Wang, Z. Huang, L. Kong, J. Xiao, P. Wang, L. Zhang, and C. Li, "Performance of training sparse deep neural networks on GPUs," in *IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2019.

[12] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2019.

[13] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, A. Reuther, R. Robinett, and S. Samsi, "Graphchallenge.org sparse deep neural network performance," 2020.

[14] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on Intel KNL and multicore architectures," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, 2018.

[15] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley–Teubner, 1990.

[16] Ü. V. Çatalyürek and C. Aykanat, "A fine-grain hypergraph model for 2d decomposition of sparse matrices." in *IPDPS*, vol. 1, 2001, p. 118.

[17] Ü. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.3*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at https://www.cc.gatech.edu/ umit/software.html, 1999.

[18] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2013.

[19] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Computing*, vol. 59, pp. 71–96, 2016.

[20] B. Uçar and C. Aykanat, "Partitioning sparse matrices for parallel preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 29, no. 4, pp. 1683–1709, 2007.