Combinatorial Tiling for Sparse Neural Networks

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

Combinatorial Tiling for Sparse Neural Networks

Filip Pawłowski, Rob H. Bisseling, Bora Uçar, Albert-Jan Yzelman

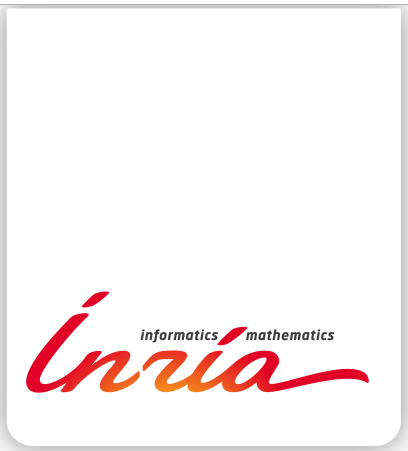
RESEARCH REPORT

N° 9357

August 2020

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9357--FR+ENG



Combinatorial Tiling for Sparse Neural Networks

Filip Pawłowski*, Rob H. Bisseling†, Bora Uçar‡, Albert-Jan Yzelman§

Project-Team ROMA

Research Report n° 9357 — August 2020 — 14 pages

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 synchronisation 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× speedup versus a data-parallel baseline.

Key-words: Sparse deep neural networks, shared-memory parallel machines

* Huawei Technologies France and ENS Lyon
† Department of Mathematics, Utrecht University, the Netherlands
‡ CNRS and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.
§ Huawei Zürich Research Centre, Zürich, Switzerland

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Pavage pour les réseaux de neurones artificiels creux

Résumé : Les réseaux de neurones artificiels profonds et creux (DNN) ont émergé à la suite de la recherche des réseaux nécessitant moins de stockage et une faible complexité de calcul. L'inférence DNN consiste à utiliser ces réseaux DNN formés pour classer un lot de données. Nous proposons une inférence DNN efficace en utilisant des modèles d'hypergraphes pour avoir du parallélisme hybride du modèle et des données. Nous exploitons du pavage et la faible synchronisation pour augmenter la réutilisation du cache, masquer le déséquilibre de charge et masquer les coûts de synchronisation. Enfin, une approche de blocage permet l'application de cette nouvelle approche d'inférence hybride. Nous expérimentons initialement en utilisant les cinq premières couches de réseaux du IEEE HPEC 2019 Graph Challenge, et nous atteignons jusqu'à 2 $\times$ d'accélération par rapport à une ligne de base ayant du parallélisme des données.

Mots-clés : Réseaux de neurones artificiels profonds, machines parallèles à mémoire partagée

parse 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 synchronisation 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× speedup versus a data-parallel baseline.

1 Introduction

The Graph Challenge encourages new developments in graph analytics, which, in its 2019 edition [**?**] focused on fast inference for sparse deep neural networks. It described the problem using the GraphBLAS C API [**?**], 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 [**?**], or result from training a fixed sparse topology as in the RadiX-Net [**?**]. 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 [**?**].

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 [**?**], with the number of neurons, layers, and bytes in Table 1. 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

Number of neurons	Input matrix	Number of layers d 120	480	1920
1024	48.86	30.47	121.88	487.51
4096	191.11	121.88	487.50	1950.01
16384	754.46	487.50	1950.00	7800.01
65536	2992.42	1950.00	7800.00	31200.01

Table 1 – The sizes (in MB) of the input feature matrix $X^{(0)}$ and the HPEC neural networks with different numbers of layers and neurons.

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 [**?**]. 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 [**?** **?** **?** **?** **?** **?**] providing performance data for analysis [**?**]. 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 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 net. The latter is enabled by the use of existing hypergraph partitioners, yielding an effective partitioning that leverages the sparsity of weight matrices. We evaluate our techniques on the data and neural networks defined by the sparse DNN challenge.

1.1 The challenge dataset

We extract some statistics of the 1024-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 here m' is the number of nonempty rows. Table 2 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 de-

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 2 – 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 $X^{(0)}$.

creases, 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.

1.2 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 multiplies are accumulated [**?**]. Our own sequential codes, alike Davis et al.'s [**?**], 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 1.1, holds for NN inference. Finally, since our inference algorithms will require two consecutive SpGEMMs where one input matrix is in common, 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 $\Theta(\sum_{i=0}^{n-1} \operatorname{nz}(X^{(k)}_{\{:,i\}}) \operatorname{nz}(W^{(k)}_{\{i,:\}}))$. The data movement incurred is equal to the combined CRS sizes of $X^{(k)}$ and $X^{(k+1)}$, which are touched once; to recall, this size is $\Theta(\operatorname{nz}(X^{(k)}) + n + 1)$ for $X^{(k)}$. The CRS of $W^{(k)}$ is touched a maximum of n_k times and a minimum of once, according to the sparsity of $X^{(k)}$ and assuming it does not hold empty rows. The SPAC structure is accessed $\Theta(\operatorname{nz} X^{(k)})$ times, which has a storage requirement of $\Theta(\max_k n_k)$.

1.3 Sequential inference

The work complexity of the sparse inference (1) derived by consecutively applying the preceding SpGEMM analysis is $\Theta\left(\sum_{k=0}^{d-1}\left[\sum_{i=0}^{n_k-1} \operatorname{nz}(X^{(k)}_{\{:,i\}}) \operatorname{nz}(W^{(k)}_{\{i,:\}})\right]\right)$. 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

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

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

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

2 Approach

In deep learning, data-parallel inference (see Section 2.2) is the de-facto standard. To enable tiled inference, we first discuss the model-parallel approach in 2.3 and extend that to a tiled one in Section 2.4. We then complete our approach by combining tiling with data-parallel inference and by describing how to apply the resulting method for deep neural networks in Section 2.5. Our model-parallel, tiled, and hybrid inference methods all rely on hypergraph modeling of the inference process. We introduce this first.

2.1 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 [**?**] 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 [**?**] 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 [**?**]). We

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

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

use PaToH [**?**], a multi-constraint hypergraph partitioner, to obtain such a balanced p-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 [**?**]); 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.

2.2 Data-parallel inference

Data-parallel inference partition 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 $i/\lceil n/p \rceil$ to achieve this, and have each thread perform sequential inference on different sets of (approximately) n/p input data elements in an embarrassingly parallel fashion.

More precisely, each thread q uses 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. The former two matrices are allocated by each thread, while the weight matrices are allocated once and shared. This approach has no work overhead compared to a sequential method. There is, however, a data movement overhead of $\Theta((\mathrm{nz}\,(W^{(k)}) + n_k)((p_t - 1)g + p_t(p_s - 1)h))$ as all threads need to refer to the shared weights; replication of the weights on each socket would reduce this to $\Theta((\mathrm{nz}\,(W^{(k)}) + n_k)(p_t - 1)g)$. The storage overhead is $\Theta(p)$ since every thread has to maintain a CRS for the various $X^{(k)}_{\{s_q,:\}}$.

2.3 Model-parallel inference

Model-parallel inference partitions the neural network, in our case corresponding to a partitioning of nonzeros of all $W^{(k)}$, mapping each individual weight to a unique thread. Let $W^{(k)_q}$ be 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 $r_q^{(k)}$, $z_q^{(k)}$, and $\lambda_{\{i,j\}}^{(k)}$. The matrix $W_q^{(k)}$ is stored locally by thread q, as are the corresponding input and output matrices $X^{(k)}_{\{:,r_q \cup r_S\}_q}$ and $X^{((k+1))}_{\{:,z_q \cup z_S\}_q}$. Since the column sets $r_q^{(k)}, z_q^{(k)}$ are disjoint, each thread q stores a disjoint set of features. The parts corresponding to separator columns $r_S^{(k)}, z_S^{(k)}$, however, are replicated across all threads. Since a thread q may refer to remotely data such as to remotely separator features $X^{(k)}_{\{:,r_S\}_r}$ $0 \leq r < p$, $r \neq q$, we retain the subscript r to emphasize which data is allocated to what thread.

A nonzero in $W_q^{(k)}$ may lie at the intersection of an internal row and an external column, or at the intersection of an external row and an internal column. Its nonzeros thus can logically be 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 in all $X^{(k)}_{\{:,r_S\}_r}$; (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 $X^{(k)}_{\{:,r_S\}_r}$ 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 summarises the resulting approach.

Algorithm 1 Model-parallel layer-k inference at thread q.

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

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

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

both writing and reading of partial results at Line 6. This yields a data movement overhead of

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

Comparing to the movement overhead of the data-parallel method, $\Theta(p\sum_{k=0}^{d-1}\operatorname{nz}(W^{(k)})h)$, the model-parallel variant is preferable whenever the combined size of all $W^{(k)}$ is smaller than that of all $X^{(k)}_{\{:,r_S\}}$. The latter 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 $\Theta(g\sum_k \max_q \operatorname{nz} W^{(k)}_q + |r_q \cup r_S|)$ data movement occurs, resulting in an additional overhead of $\Theta(pg\sum_k |r^{(k)}_S| + \varepsilon\sum_k \operatorname{nz} W^{(k)})$, with ε the attained load imbalance of the partitioning used.

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

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

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

2.4 Tiled 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 tiled model-parallel inference, each thread q invokes Algorithm 2 to run a model-parallel inference separately for $m = n/b_{size}$ batches of $X^{(k)}$ containing b_{size} rows each. To reduce the barrier overhead of $\Theta(dpL)$, each thread q overlaps the barrier with computation of local results $X^{((k+1))}_{\{:,z_q\}_q}$, implemented by weak point-to-point synchronisations before processing contributions of remote partial results $X^{(k+1)}_{\{:,z_S\}_{r\neq q}}$ at Line 22. The costs of tiled inference are similar to that of model-parallel, with overheads multiplied by m 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 m– there exists a trade-off between choosing higher block sizes that tile for smaller caches versus how well underlying neural nets can be partitioned. Since there is no explicit barrier, synchronisation overheads lie far below $\mathcal{O}(mdpL)$ in practice.

2.5 Hybrid and deep inference

We may combine the data-parallel and tiled inference methods. Assume $p = p_0 p_1$ threads, then hybrid inference splits n into p_0 parts using a block distribution and processes each part using the tiled inference with p_1 threads. This uses p_0 threads to replicate the p_1-way partitioned weights, and ensures that we may simply 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 $p_0 > \lfloor p/p_1 \rfloor$ then enables use of (almost) all available cores.

Algorithm 2 Model-parallel tiled 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 $C_{\{b_m,:\}_q}$ be a $b_m \times n_{k-1}$ in-cache matrix
  Let $D_{\{b_m,:\}_q}$ be a $b_m \times n_k$ in-cache matrix
  $D_{\{b_m,:\}_q} \leftarrow X^{(0)}_{\{b_m,:\}}$ ▷ 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
      $r = (q+1)$ **mod** p, $done = 1$
      while $done < p$ **do**
        if *checkAvail*$(D_{\{:,r_S\}_r})$ **then**
          $D_{\{s_q,r_S\}_q} \leftarrow D_{\{s_q,r_S\}_q} + D_{\{s_q,r_S\}_r}$
          $done \leftarrow done + 1$
        $r \leftarrow (r+1)$ **mod** p
        if r equals q **then**
          $r = (q+1)$ **mod** p
    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**
          $C_{\{s_r,z_q\}_q} \leftarrow D_{\{s_r,r_q\}_q} W^{(k_q)}_{\{r_q,z_q\}} +$
            $D_{\{s_r,r_S\}_r} W^{(k_q)}_{\{r_S,z_q\}}$
          $C_{\{s_r,z_S\}_q} \leftarrow D_{\{s_r,r_q\}_q} W^{(k_q)}_{\{r_q,z_S\}} +$
            $D_{\{s_r,r_S\}_r} W^{(k_q)}_{\{r_S,z_S\}}$ ▷ Without ReLU
          $done \leftarrow done + 1$
        $r \leftarrow (r+1)$ **mod** 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
  $X^{(d)}_{\{s_q,z_S\}} \leftarrow D_{\{s_q,z_S\}_q}$

Deep neural networks where weights do not fit local cache sizes can still benefit from tiling as follows. Cut the layers of the network into successive blocks, applying 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 tiled algorithm. This requires streaming from main memory not only $X^{(0)}$, but all of $X^{(5k)}$, $0 \leq k \leq 24$. 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 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 tiled approach as presented here should lie at the core of a competitive sparse neural network inference method. Assuming an appropriate partitioning, a blocked tiled inference would identify the block size, i.e., the maximal number of consecutive layers of the network which combined 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 in case of the datasets and neural networks proposed in the Graph Challenge.

3 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 [**?**]. 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 tiled 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 tiled 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 tiled methods, we take the maximum tile size for which thread-local tiles of $X^{(k)}$ remain in L3 cache, together with all thread-local parts of $W_q^{(k)}$. The sparsity of C and D are unknown a priory and change as inference proceeds. We therefore assume those matrices are dense and correct the resulting storage requirements by a parameter $\alpha \leq 1$ to account for the sparsity of the $X^{(k)}$. If the combined local weight matrices do not fit in cache we aim to fit only the intermediate tiles of $X^{(k)}$. We use the density of the input matrix $X^{(0)}$ as an estimate for sparsity during inference, ensure that $b_{size} > 0$ is divisible by p_1.

We experiment on the first five layers of the 120-layer HPEC networks to confirm the beneficial effects of caching the intermediate results, and to confirm our understanding of the performance characteristics of the proposed tiled and hybrid inference methods. We expect that if the $X^{(k)}$ fit in cache, there will be no benefit from tiling versus the standard data-parallel inference. This is the case on the Ivy Bridge machine for both the 1024-neuron and 4096-neuron HPEC neural nets (where additionally the $X^{(k)}$ have decreasing densities as k increases, see Table 1). Indeed the results in Table 3 confirm slowdowns of the tiled method versus the data-parallel baseline, which become more with increasing p due to the parallel overhead increasing with the separator size. The same effect is observed for the 4096-neuron network, and hence omitted from the table.

	1024-neuron				16384-neuron				65536-neuron			
	Separator	Tile	Time (in s)		Separator	Tile	Time (in s)		Separator	Tile	Time (in s)	
p	(in %)	size	Baseline	Tiled (speedup)	(in %)	size	Baseline	Tiled (speedup)	(in %)	size	Baseline	Tiled (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 3 – Tiled inference results over the first 5 layers of 1024-, 16384-, and 65536-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	Tiled (speedup)
16384	20	5	4	3.66	2.20 (1.66)
16384	40	20	2	2.56	1.81 (1.41)
65536	20	4	5	12.58	7.62 (1.65)
65536	40	8	5	9.72	6.49 (1.50)

Table 4 – Hybrid tiled inference results over the first 5 layers of 16384-, and 65536-neuron networks on Ivy Bridge.

After 16384 and 65536 neurons the $X^{(0)}$ do not fit in cache, though due to decreasing density it may be that later $X^{(k)}$ could fit, which would be beneficial to data-parallel inference. Regardless, we expect better performance from our tiling variant, certainly 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 3 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, and use the remainder available threads for hybrid inference. Table 4 shows the results for the 5-layer 16384- and 65536-neuron network. These confirm that the hybrid tiled inference scales, maintaining similar speedups versus the data-parallel baseline as for the pure tiling method with $p = p_1$ threads. We additionally tested performance with hyperthreads in Table 4 and conclude they benefit both the data-parallel and tiling methods, in the latter case presumably made possible by the almost synchronization-free method employed between (hyper)threads.

The fastest inference on the first five HPEC layers using the full Ivy Bridge machine are obtained by our hybrid tiled method, which are 41% faster versus the data-parallel method for the 16384 neuron network, and 50% faster on the 65536 one.

Finally, we repeat both experiments for the Cascade Lake machine in Tables 6 and 5. For the hybrid 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 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 65k-neuron experiment, respectively, using all 22 cores with hyperthreads in a $22 \cdot 2$ configuration.

				Time (in s)	
Network	p	p_0	p_1	Baseline	Tiled (speedup)
16384	20	5	4	4.65	2.61 (1.78)
16384	22	11	2	4.49	2.23 (2.01)
16384	24	6	4	3.96	2.19 (1.81)
16384	44	22	2	2.36	1.18 (2.00)
16384	44	4	11	2.36	1.63 (1.45)
65536	20	5	4	16.99	9.06 (1.88)
65536	22	11	2	15.85	8.49 (1.87)
65536	40	5	8	8.91	4.75 (1.88)
65536	44	22	2	8.68	4.47 (1.94)
65536	44	4	11	8.68	5.10 (1.70)

Table 5 – Hybrid tiled inference results over the first 5 layers of 16384-, and 65536-neuron networks on Cascade Lake.

	16384-neuron			65536-neuron		
	Tile	Time (in s)		Tile	Time (in s)	
p	size	Baseline	Tiled (speedup)	size	Baseline	Tiled (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)

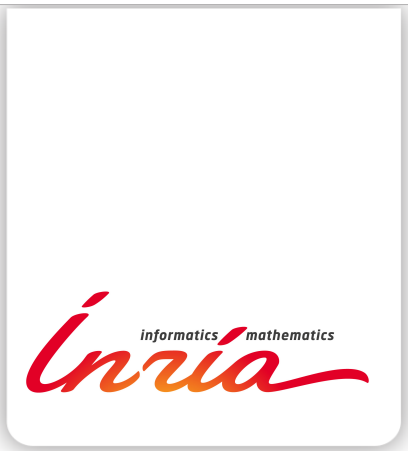
Table 6 – Tiled inference results over the first 5 layers of 16384-, and 65536-neuron networks on Cascade Lake.

4 Conclusions

We proposed 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 in such a way it accurately captures dependencies and communication between those layers. By applying multi-constraint load balance criteria we use this model to create partitioning of layers that minimise overhead during model-parallel inference.

We then proposed 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 synchronisation costs. The hybrid tiled algorithm offers best results for inference on the first five layers of the deep RadiX-Net NNs, provided the input feature matrix is sufficiently large as confirmed using the 16384 and 65536 neuron networks. We demonstrated the approach works across architectures.

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 [**?**]. Allowing permutations between layers, which can also be modeled via hypergraphs [**?**], may allow for further decreased separator size. Work is in progress to efficiently apply the proposed hybrid tiled 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.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399