High-performance dense tensor and sparse matrix kernels for machine learning

Filip Igor Pawlowski

▶ **To cite this version:**

Filip Igor Pawlowski. High-performance dense tensor and sparse matrix kernels for machine learning. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Lyon, 2020. English. NNT : 2020LYSEN081 . tel-03116812

HAL Id: tel-03116812
https://theses.hal.science/tel-03116812

Submitted on 20 Jan 2021

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.

Numéro National de Thèse : 2020LYSEN081

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N°512
École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 11/12/2020, par :

Filip Igor PAWLOWSKI

High-performance dense tensor and sparse matrix kernels for machine learning

Noyaux de calcul haute-performance de tenseurs denses et matrices creuses pour l'apprentissage automatique

Devant le jury composé de :

Alfredo	BUTTARI	Chercheur, CNRS	*Rapporteur*
X. Sherry	LI	Directrice de recherche, Lawrence Berkeley National Lab., Etats-Unis,	*Rapporteure*
Ümit V.	ÇATALYÜREK	Professeur, Georgia Institute of Tech., Etats-Unis,	*Examinateur*
Laura	GRIGORI	Directrice de recherche, Inria	*Examinatrice*
Bora	UÇAR	Chercheur, CNRS	*Directeur de thèse*
Albert-Jan N.	YZELMAN	Chercheur, Huawei Zürich Research Center, Suisse	*Co-encadrant de thèse*

Contents

Résumé français vi

1 Introduction **2**
1.1 General background 2
1.1.1 The cache memory and blocking 3
1.1.2 Machine and cost model 4
1.1.3 Parallel algorithm analysis 6
1.1.4 Memory allocation and partitioning 7
1.2 Thesis outline 7
1.2.1 Tensor products 8
1.2.2 Sparse inference 9

2 Tensor computations **12**
2.1 Introduction 13
2.2 Related work 14
2.3 Sequential tensor–vector multiplication 17
2.3.1 Tensor layouts 17
2.3.2 Two state-of-the-art tensor–vector multiplication algorithms . . . 19
2.3.3 Block tensor–vector multiplication algorithms 20
2.3.4 Experiments 22
2.4 Shared-memory parallel tensor–vector multiplication 35
2.4.1 The loopedBLAS baseline 37
2.4.2 Optimality of one-dimensional tensor partitioning 38
2.4.3 Proposed 1D *TVM* algorithms 40
2.4.4 Analysis of the algorithms 42
2.4.5 Experiments 46
2.5 Concluding remarks 50

3 Sparse inference **53**
3.1 Introduction 54
3.1.1 Sparse inference 54
3.1.2 Sparse matrix–sparse matrix multiplication 55
3.1.3 Hypergraph partitioning 56
3.1.4 Graph Challenge dataset 57
3.1.5 State of the art 58

3.2 Sequential sparse inference 60
3.2.1 SpGEMM-inference kernel 60
3.2.2 Sparse inference analysis 62
3.2.3 SpGEMM-inference kernel for partitioned matrices 63
3.3 Data-, model- and hybrid-parallel inference 64
3.3.1 Data-parallel inference 64
3.3.2 Model-parallel inference 67
3.3.3 Hybrid-parallel inference and deep inference 72
3.3.4 Implementation details 74
3.4 Experiments 74
3.4.1 Setup 75
3.4.2 The tiling model-parallel inference results 76
3.4.3 The tiling hybrid-parallel inference results 80
3.5 Concluding remarks 82

4 Conclusions **83**
4.1 Summary 83
4.1.1 Summary of Chapter 2 83
4.1.2 Summary of Chapter 3 84
4.2 Future work 86
4.2.1 Tensor computations 86
4.2.2 Sparse networks 87

Bibliography **93**

List of Algorithms

2.1 The looped tensor–vector multiplication 20
2.2 The unfold tensor–vector multiplication. 21
2.3 The block tensor–vector multiplication algorithm 22
2.4 The next block according to a ρ_π layout. 23
2.5 The next block according to a Morton layout. 24
2.6 A basic higher-order power method 35
2.7 The q-sync parallel TVM algorithm. 41
2.8 The interleaved $q(i)$-sync parallel TVM algorithm. 42
2.9 The explicit $q(e)$-sync parallel TVM algorithm. 42
3.1 The SpGEMM-inference kernel. 61
3.2 The SpGEMM-inference kernel for partitioned matrices. 65
3.3 The model-parallel inference at layer k. 69
3.4 The tiling model-parallel inference. 73
3.5 The latency-hiding model-parallel layer-k inference. 75

List of Figures

2.1 Plot of the effective bandwidth (in GB/s) of the copy kernels. 26
2.2 Illustration of elements lying on all axes going through tensor element(s). 39

3.1 The staircase matrix of a neural network. 71
3.2 Plot of the run time of the data-parallel and the tiling model-parallel inference using 5 layers on Ivy Bridge. 77
3.3 Plot of the run time of the data-parallel and the tiling model-parallel inference using 5 layers on Cascade Lake. 78

Résumé français

Dans cette thèse, nous développons des algorithmes à haute performance pour certains calculs impliquant des tenseurs denses et des matrices creuses. Nous abordons les noyaux de calculs qui sont utiles pour les tâches d'apprentissage automatique, telles que l'inférence avec les réseaux neuronaux profonds (DNN). Nous développons des structures de données et des techniques pour réduire l'utilisation de la mémoire, pour améliorer la localisation des données et donc pour améliorer la réutilisation du cache des opérations du noyau. Nous concevons des algorithmes séquentiels et parallèles à mémoire partagée.

Dans la première partie de la thèse, nous nous concentrons sur les noyaux de calculs de tenseurs denses. Les noyaux de calculs de tenseurs comprennent la multiplication tenseur-vecteur (TVM), la multiplication tenseur-matrice et la multiplication tenseur-tenseur. Parmi ceux-ci, la TVM est la plus limitée par la bande passante et constitue un élément de base pour de nombreux algorithmes. Nous nous concentrons sur cette opération et développons une structure de données et des algorithmes séquentiels et parallèles pour celle-ci. Nous proposons une nouvelle structure de données qui stocke le tenseur sous forme de blocs, qui sont ordonnés en utilisant la courbe de remplissage de l'espace connue sous le nom de courbe de Morton (ou courbe en Z). L'idée principale consiste à diviser le tenseur en blocs suffisamment petits pour tenir dans le cache et à les stocker selon l'ordre de Morton, tout en conservant un ordre simple et multidimensionnel sur les éléments individuels qui les composent. Ainsi, des routines BLAS haute performance peuvent être utilisées comme micro-noyaux pour chaque bloc. Nous évaluons nos techniques sur un ensemble d'expériences. Les résultats démontrent non seulement que l'approche proposée est plus performante que les variantes de pointe jusqu'à 18%, mais aussi que l'approche proposée induit 71% de moins d'écart-type d'échantillon pour le TVM dans les différents modes possibles. Enfin, nous montrons que notre structure de données s'étend naturellement à d'autres noyaux de calculs de tenseurs en démontrant qu'elle offre des performances jusqu'à 38% supérieures pour la méthode de puissance d'ordre supérieur. Enfin, nous étudions des algorithmes parallèles en mémoire partagée pour la TVM qui utilisent la structure de données proposée. Plusieurs algorithmes parallèles alternatifs ont été caractérisés théoriquement et mis en œuvre en utilisant OpenMP pour les comparer expérimentalement. Nos résultats sur un maximum de 8 processeurs montrent une performance presque maximale pour l'algorithme proposé pour les tenseurs à 2, 3, 4 et 5 dimensions.

Dans la deuxième partie de la thèse, nous explorons les calculs creux dans les réseaux de neurones en nous concentrant sur le problème d'inférence profonde creuse à haute

performance. L'inférence creuse de DNN représente la tâche où les DNN creux classifient un lot d'éléments de données formant, dans notre cas, une matrice creuse. La performance de l'inférence creuse dépend de la parallélisation efficace de la multiplication matrice creuse-matrice creuse (SpGEMM) répétée pour chaque couche de la d'inférence. Nous caractérisons d'abord les algorithmes SpGEMM séquentiels efficaces pour notre cas d'utilisation. Nous introduisons ensuite l'inférence modèle-parallèle, qui utilise un partitionnement bidimensionnel des matrices de poids obtenues à l'aide du logiciel de partitionnement des hypergraphes. La variante modèle-parallèle utilise des barrières pour synchroniser entre les couches. Enfin, nous introduisons les algorithmes de tuilage modèle-parallèle et de tuilage hybride, qui augmentent la réutilisation du cache entre les couches, et utilisent un module de synchronisation faible pour cacher le déséquilibre de charge et les coûts de synchronisation. Nous évaluons nos techniques sur les données du grand réseau de l'IEEE HPEC 2019 Graph Challenge sur les systèmes à mémoire partagée et nous rapportons jusqu'à 2 fois de l'accélération par rapport à la référence.

Chapter 1

Introduction

This thesis is in the field of high-performance computing and focuses on finding high-performance algorithms which solve computational problems involving dense tensors and sparse matrices. Traditional computational problems involve dense matrix and sparse matrix computations with potentially only a few matrices, as in the preconditioned iterative methods or multigrid methods for solving linear systems. Up-and-coming tasks in data analysis and machine learning deal with multi-modal data and involve a large number of matrices. We see two classes of computational problems associated with these tasks: those having dense tensors, which are collections of dense matrices, and those having a large number of sparse matrices. We identify frequently used kernels in these problems. We discuss a machine model and a cost model to quantify the data movement of parallel algorithms. We develop data structures and sequential kernel implementations which increase cache reuse. We then develop shared-memory parallel algorithms and analyze their practical effects using the proposed cost model as well as the effects of the proposed data structures on high-end computer systems. We use those kernels in applications concerning tensor decomposition and analysis, and in inference with sparse neural networks.

In this chapter, we first give a general background that underlines our approach to the two subjects of the thesis in Section 1.1. Section 1.2 then gives an outline of the thesis by summarizing the problems, contributions, and the main results.

1.1 General background

In this thesis, we are interested in designing algorithms and techniques that apply to modern architectures. As the memory throughput grows much slower than the machine's computational power, increasingly many workloads are bottlenecked by memory, on modern machines. On modern hardware, a computing unit has multiple local memory storages connected to it. One such memory, caches, have lower latency and higher bandwidth but smaller size when compared to main memory. The cache memory is a hardware optimization design to store the data a computing unit requires as it might be used again in nearby future. On parallel machines, when these computing units are

connected and share their local memories with the other computing units, a common characteristic is that the time to access local memory is much shorter than to access the local memories of other units. This gives rise to the notion of local and remote memories for each unit. Shared-memory architectures with this characteristic are known as the Non-Uniform Memory Access (NUMA) machines, as the main memory accesses have different latency and throughput. To design efficient parallel algorithms we must be able to estimate their costs on such hardware.

This section is organized as follows. In Section 1.1.1 we describe the cache memory and the programming techniques that we adopt in this thesis to use the cache memory effectively. In Section 1.1.2, we describe a bridging model of NUMA hardware which includes a model of parallel computation and a cost model we use to analyze parallel algorithms. In Section 1.1.3, we introduce the concept of parallel overheads we use to compare parallel algorithms with respect to the best sequential alternative. In Section 1.1.4 we discuss allocation techniques which allow to effectively distribute data on a shared-memory machine.

1.1.1 The cache memory and blocking

The cache memory is an on-chip memory which is closer to the CPU and hence faster to access, but also very small due to its high production costs. Typically, CPUs contain multiple levels of caches where the higher level caches have higher latencies and lower bandwidths, but greater size. This is done as programs have tendency to reuse the same data in a short period of time, which is referred to as *temporal locality*. The computing unit first looks up data in each level of cache. If the cache contains the required element, it is called a *cache hit*. Otherwise, if the element has not been found, it is fetched from the main memory and loaded into cache and processor registers; this is called a *cache miss*. The hardware policy governs which elements to then evict from cache when it reaches its capacity. A perfect architecture usually assumes the Least Recently Used (LRU) policy, which evicts the least recently used elements first.

In the thesis, we use a well-known software optimization technique to increase the temporal locality of algorithms known as the *loop blocking*, or *loop tiling*. This technique reorganizes the computation in the program to repeatedly operate on small parts of data, known as blocks or tiles. The size of a block should be such that a block's data fits in cache. If the block's data fits in cache, then the number of cache misses reduces and hence one sees improved performance. We note that such a blocking is justified only under two conditions. First, a program must reuse the data in blocks as opposed to *streaming* them, that is, touching the data only once. Second, the original data should be much larger than the cache size; otherwise, blocking becomes an overhead. Optimal block sizes could be determined statically on a per-machine basis, either analytically or via (manual or automated) experimentation. The latter approach results in a process known as *parameter tuning*.

Spatial locality is another property of programs which is crucial to performance. It occurs when a program accesses a data item and subsequently requires another data item lying in close-by memory areas; a special case is when a program iterates over successive

memory addresses in a streaming fashion. We note that blocking does not necessarily achieve spatial locality, which itself relates not only to the computation but also to the data layout. In the thesis, we modify the storage of input data and propose new data layouts which improve spatial locality of blocking algorithms.

1.1.2 Machine and cost model

At an abstract level, a shared-memory parallel NUMA machine consists of p_s connected processors, or sockets. Each socket consists of p_t threads thus yielding a total of $p = p_s p_t$ threads. Each socket has local memory connected to it: the cache, RAMs, and the main memory. The sockets are connected with each other via a communication bus such that the memory is shared and forms a global address space. A thread executing on a socket may access the *local* memory of the socket faster than the *remote* memory of the other sockets; a NUMA effect.

We use a set of parameters to characterize such a shared-memory NUMA machine:

- r, the time a flop operation takes in seconds;
- L, the time in which a barrier completes in seconds;
- g, the time required to move a byte from local memory to a thread in seconds; and
- h, the time to move a byte from remote memory to the thread in seconds.

Thus, g is inversely proportional to the intra-socket throughput while h is inversely proportional to the inter-socket memory throughput per socket. These parameters together with p_s and p_t fully characterize a machine and allow quantifying the behavior of a program running on that machine.

The following model allows quantifying the cost of a parallel program. Any parallel program may be viewed as a series of S barriers with $S + 1$ phases in between them. Therefore, each thread $q \in \{0, \ldots, p-1\}$ executes $S + 1$ different phases, which are numbered using integer s, $0 \leq s \leq S$. We quantify the computation in terms of the number of floating point operations, or flops, which includes all scalar operations. We use $W_{q,s}$ to count the local computation at thread q in flops at phase s. For each thread, we distinguish between the intra-socket and inter-socket data movement. While the intra-socket data movement $U_{q,s}$ counts the data thread q reads and writes in the local memory in words at phase s, the inter-socket data movement $V_{q,s}$ counts the data items in words q reads and writes in the remote memory in words at phase s. We use $M_{q,s}$ to denote the storage requirement by thread q at phase s in bytes. While these fully quantify an algorithm, their values depend on the number of threads p as well as on any problem parameters.

In general, the cost of each phase s is proportional to the slowest thread, which completes in $\max_q\{W_{q,s}r + U_{q,s}g + V_{q,s}h\}$. Thus, the time in which an algorithm consisting of $S + 1$ phases completes is

$$T(n,p) = \sum_{s=0}^{S} \left[\max_{q \in \{0,\ldots,p-1\}} \{W_{q,s}r + U_{q,s}g + V_{q,s}h\} \right] + SL \text{ seconds.} \tag{1.1}$$

However, if the local computational may be overlapped with data movement, the same algorithm completes in

$$T(n,p) = \sum_{s=0}^{S} \left[\max\{ \max_{q\in\{0,\ldots,p-1\}} W_{q,s}r, \max_{q\in\{0,\ldots,p-1\}} \{U_{q,s}g + V_{q,s}h\} \right] \} + SL \text{ seconds.} \quad (1.2)$$

Note that for any s:

$$\max_{q\in\{0,\ldots,p-1\}} \{W_{q,s}r + U_{q,s}g + V_{q,s}h\} / \max\{ \max_{q\in\{0,\ldots,p-1\}} W_{q,s}r, \max_{q\in\{0,\ldots,p-1\}} \{U_{q,s}g + V_{q,s}h\}\} \leq 2,$$

and that this upper bound is reached only if

$$\max_{q\in\{0,\ldots,p-1\}} W_{q,s}r = \max_{q\in\{0,\ldots,p-1\}} \{U_{q,s}g + V_{q,s}h\}.$$

Minimizing each of the total asymptotic costs of computation, data movement and synchronization for any input and number of threads, i.e., minimizing

$$\begin{aligned} T_{work} &= \sum_{s=0}^{S} \max_{q\in\{0,\ldots,p-1\}} W_{q,s}r \\ T_{data} &= \sum_{s=0}^{S} \max_{q\in\{0,\ldots,p-1\}} \{U_{q,s}g + V_{q,s}h\} \\ T_{sync} &= SL, \end{aligned} \quad (1.3)$$

thus minimizes both the overlapping and non-overlapping versions of $T(n,p)$. The final cost metric we use is the storage requirement of a parallel algorithm. It is the maximum storage requirement $M(n,p) = \max_{q\in\{0,\ldots,p-1\}} \sum_{s=0}^{S} M_{q,s}(n,p)$ per thread during all phases.

We note that in practice, many problems do not have an algorithmic solution which minimizes all three costs simultaneously. In the thesis, we propose algorithms which have various ratios between these costs. The final implementation may be chosen based on the number of threads, the problem size, and the parameters values r, g, h and L of the machine.

Related models. The Random Access Machine (RAM) is an agreed upon model of computation for sequential machines, while for parallel computing the Bulk Synchronous Parallel (BSP) and Communicating Sequential Processes (CSP) models quantify the costs of parallel programs effectively under different scenarios. While our model focuses on data movement, the BSP [61] is another bridging model of hardware which explicitly models communication. The originally proposed BSP model is also known as the flat BSP model, as it states that all processes take equal time to communicate with each other and does not capture the NUMA effects. Variants of the BSP model which capture NUMA effects and provide a cost model are the BSPRAM [57] and the multi-BSP [62]. The multi-BSP model also considers multiple levels of memory locality, e.g., a shared-memory and distributed-memory parallel machine where memory may be seen at three

levels. Another parallel model of computation is the Parallel Random Access Machine (PRAM) [22] which assumes that the synchronization issues are resolved by the hardware itself and that communication has a constant cost. Therefore, it cannot serve as a tool to model data movement costs.

1.1.3 Parallel algorithm analysis

A basic metric to measure performance of a parallel algorithm is the ratio of a sequential algorithm run time T_{seq} to the parallel run time for p threads,

$$S(n,p) = \frac{T_{seq}(n)}{T(n,p)}.$$

Ideally, the speedup of a parallel program grows linearly with p for a constant problem size n. However, Amdahl's law states that this is not attainable in practice as most parallel algorithms possess a *parallel overhead*

$$O(n,p) = pT(n,p) - T_{seq}(n).$$

It may be thought of as the execution time of the part of the program which cannot be parallelized, i.e., its critical section. Except for trivially parallel algorithms, the critical section increases with n and p. Another metric to evaluate performance of a parallel algorithm is the *parallel efficiency* defined as the ratio between speedup and the number of threads:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{seq}(n)}{pT(n,p)}.$$

Gustafson's law states that the speedup of a parallel program grows linearly with p, that is, its efficiency remains constant, if n increases. This is true for most parallel algorithms as T_{seq} grows with n as well, thus amortizing the impact of the overhead on efficiency:

$$E(n,p) = 1 - \frac{O(n,p)}{O(n,p) + T_{seq}(n)}.$$

The above formula allows to compute how fast n should grow to retain the same parallel efficiency as p increases and vice versa, giving rise to the concept of *iso-efficiency*. *Strongly scaling* algorithms have that the overhead $O(n,p)$ is independent of p, which is unrealistic, while *weakly scaling* algorithms have that the ratio $O(pn,p)/T_{\mathrm{seq}}(pn)$ is constant; iso-efficiency instead tells us a much wider range of conditions under which the algorithm scales. We note that the total number of threads an algorithm employs need not be equal to the number of cores a given machine holds; it can be less when considering strong scalability, and it can be more when exploring the use of hyperthreads.

Using the parallel costs defined in (1.3), we subdivide the parallel overhead and define the following parallel overheads for each of the corresponding parallel costs to quantify parallel algorithms:

$$\begin{aligned}
O_{work}(n,p) &= pT_{work}(n,p) - T_{seq\text{-}work}(n) \\
O_{data}(n,p) &= pT_{data}(n,p) - T_{seq\text{-}data}(n) \\
O_{sync}(n,p) &= pT_{sync}(n,p) - T_{seq\text{-}sync}(n).
\end{aligned}$$

We note that when counting the data movement overhead $O_{data}(n, p)$ for algorithms which use blocking, we assume a perfect caching occurs for blocks and that they do not contribute to the overhead. All overheads should compare against the best performing sequential algorithm, which may be drastically different than the parallel algorithm for $p = 1$. Finally, we define the parallel storage overhead $O_{mem}(n, p) = pM(n, p) - M_{seq}(n)$, where M_{seq} is the storage size required by the best sequential algorithm.

The final metric we use is the *arithmetic intensity*, which is a ratio between the number of floating point operations an algorithm performs versus the memory size it touches during the computation. We determine the arithmetic intensity of the best sequential algorithm to determine how likely the algorithm is to fully utilize the memory bandwidth of a machine rather than its computational power. While the bottleneck of *compute-bound* algorithms is the computational power of a machine, the bottleneck of *bandwidth-bound* algorithms is the memory speed. However, this does not preclude memory optimizations from benefiting compute-bound kernels, but rather states that these will improve the run time to a smaller degree.

1.1.4 Memory allocation and partitioning

On shared-memory systems, each processor has local memory to which it accesses faster than remote memory areas. We assume that threads taking part in a parallel computation are *pinned* to a specific core, meaning that threads will not move from one core at run time. A pinned thread has a notion of local memory: namely, all addresses that are mapped to the memory controlled by the processor the thread is pinned to. This gives rise to two distinct modes of use for shared memory areas: the *explicit* versus *interleaved* modes. If a thread allocates, initializes, and remains the only thread using this memory area, we dub its use explicit. In contrast, if the memory pages associated with an area cycle through all available memories, then the use is called interleaved. The latter mode is enabled by NUMA-ctl library [47]. If a memory area is accessed by all threads in a uniformly random fashion, then it is advisable to interleave it to achieve high throughput.

1.2 Thesis outline

The main contributions of the thesis are discussed in two chapters. Chapter 2 focuses on problems in dense tensor computations, while Chapter 3 treats the problem of sparse inference. In both chapters, the main elements of our approach are as follows. We propose algorithms and implement them on shared-memory systems. We analyze the proposed algorithms theoretically using the metrics discussed in Section 1.1.3. We also analyze the proposed algorithms experimentally, where parallel codes use OpenMP and are run on shared memory systems.

1.2.1 Tensor products

In Chapter 2, we investigate computations on dense tensors (or multidimensional arrays) in d modes (dimensions). Much like matrices can be multiplied with vectors or matrices, tensors can be multiplied with vectors, matrices, or tensors. These multiplication operations are called tensor–vector multiplication (TVM), tensor–matrix multiplication (TMM), and tensor–tensor multiplication (TTM). These multiplication operations apply to specific modes; each multiplication can operate on a subset of modes of the input tensor. We dub algorithms such as the TVM, TMM, and TTM *kernels*. Much in line with the original Basic Linear Algebra Subprograms (BLAS) definition [19, 20], we classify the TVM as a generalized BLAS level-2 (BLAS2) kernel, while we classify the TMM, TTM, and Khatri-Rao products [27] as generalized BLAS3 ones. These kernels form the core components in tensor computation algorithms [4]; one example is the computation of Candecomp/Parafac decomposition of tensors using the alternating least squares method [5] and its computationally efficient implementations [27, 31, 52].

In Chapter 2, we first focus on sequential algorithms to optimize the TVM operations. We define a tensor kernel to be *mode-aware* if its performance strongly depends on the mode in which the kernel is applied; otherwise, we define the kernel to be *mode-oblivious*. This informal definition is in-line with the more widely known concept of cache-aware versus cache-oblivious algorithms [23]. We propose block-wise storage for tensors to mode-obliviously support common tensor kernels. We closely investigate the TVM kernel, which is the most bandwidth-bound due to low arithmetic intensity (defined in Section 1.1.3). Thus, among the three multiplication operations, TVM is the most difficult one to achieve high performance. However, as guaranteeing efficiency for bandwidth-bound kernels is harder, the methods used for them can be extended to others. Efficient TMM and TTM kernels, in contrast, often make use of the compute-bound general matrix–matrix multiplication (BLAS3).

Tensors are commonly stored in an *unfolded* fashion, which corresponds to a higher-dimensional equivalent of row-major or column-major storage for matrices. While a matrix can be unfolded in two different ways, a d-dimensional tensor can be stored in $d!$ different ways, depending on the definition of precedence of the modes. We discuss previous work in tensor computations, including tensor storage and develop a notation for precisely describing a tensor layout in computer memory and for describing how an algorithm operates on tensor data stored that way.

We discuss various ways for implementing the TVM. The first one notes TVM's similarity to the matrix–vector multiplication (MVM). It takes a tensor, the index of a mode, and a vector of size conformal to that mode's size and performs scalar multiply and add operations. In fact the MVM kernel can be used to carry out a TVM by either (i) reorganizing the tensor in memory (*unfolding* the tensor) followed by a single MVM; or (ii) reinterpreting the tensor as a series of matrices, on which a series of MVM operations are executed. We describe how to implement them using BLAS2, resulting in two highly optimized baseline methods. We then introduce our proposed blocked data structure for efficient, mode-oblivious performance. A blocked tensor is a tensor with smaller equally-sized tensors as its elements. We consider only the case where smaller tensor blocks are

stored in an unfolded fashion and are processed using one or more BLAS2 calls. We define two *block layouts*, which determine the order of processing of the smaller blocks: either a simple, natural ordering of dimensions or one inferred from the Morton order [45].

The experiments show that the Morton order blocked data structure offers higher performance on the *TVM* kernel when compared to the state-of-the-art methods. It also maintains a significantly lower standard deviation of performance when the *TVM* is applied on different modes, thus indeed achieving mode-oblivious behavior. We use the proposed data structure and *TVM* algorithm to implement a method used in tensor decomposition and analysis, and show that the superior performance observed for the *TVM* is retained.

We then turn our attention to shared-memory parallel *TVM* algorithms based on the proposed sequential algorithm and Morton-blocked layout. We prove that a one-dimensional partitioning is communication optimal under an assumption that the *TVM* kernel is applied in a series, each time to the input tensor. We characterize several alternative parallel algorithms which follow the algorithmic bound in (1.1) and implement two variants using OpenMP which we compare experimentally. Our results on up to 8 socket systems show near peak performance for the proposed algorithm for 2, 3, 4, and 5-dimensional tensors.

The work we present in this chapter has been published in a journal [50] and a conference [49]:

- F. Pawłowski, B. Uçar, and A. N. Yzelman, A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations, Journal of Computational Science, 33 (2019), pp. 34–44. This paper discusses the mode-oblivious storage and the sequential kernel.

- F. Pawłowski, B. Uçar, and A. N. Yzelman. High performance tensor–vector multiplication on shared-memory systems. In R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 38–48, Cham, 2020. Springer International Publishing. This paper discusses the parallel tensor–vector multiplication algorithm design.

1.2.2 Sparse inference

In Chapter 3, we explore the sparse computations in neural networks by focusing on the high-performance deep inference problem with sparsely connected neural networks. Sparse inference is the task of classifying a number of data items using a sparse neural network. This problem was posed by the IEEE HPEC Graph Challenge 2019 [33]. In the case of the Graph Challenge, the data items are also sparse, thus forming a sparse input as well.

In summary, a *neural network* (NN) consists of $d \in \mathbb{N}$ layers of neurons, where the neurons at each level combine outputs of all neurons from the previous layer by a weighted sum, potentially add biases, and apply an activation function to produce an output for the next layer. The first layer is called the input layer and the last layer is called the output layer. If a neuron does not combine the output of all neurons in the previous

layer, then the neural network becomes sparse. In the sparse inference, the aim is to classify a given input into one of the classes decided so far. The typical example [36] is to classify or map a given hand-written digit into the intended digit $0, \ldots, 9$.

The weighted combination at layer k of the outputs of the neurons from the previous layer can be accomplished by a matrix–matrix multiply of the form $X^{(k)}W^{(k)}$, where $W^{(k)}$ lists the weights and $X^{(k)}$ is the output of the previous layer. With the biases and the activation function, the whole inference then can be described as computing the final *classification matrix* $X^{(d)} \in \mathbb{R}^{n \times c}$ from the input feature matrix $X^{(0)}$:

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

Here, $X^{(0)}$ consists of sparse feature vectors, one row for each data instance to be classified, and $X^{(d)}$ has a row for each data instance and a column for each potential output class. The function $f : \mathbb{R} \to \mathbb{R}$ is the 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 $\mathbf{e}_n$ is the vector of ones, $\mathbf{e}_n = (1, 1, \ldots, 1)^T \in \mathbb{R}^{n \times 1}$. In our case, the input data instances are sparse, the connections are sparse and hence the weight matrices are sparse, all together these give rise to the *sparse inference* problem of Chapter 3. There are different activation functions; our application in Chapter 3 uses one called ReLU, which replaces negative entries by zeros.

The performance of sparse inference hinges on the performance of sparse matrix–sparse matrix multiplication (SpGEMM) as can be seen in the equation above. We first observe that the computation of SpGEMM may be combined with the activation function and the bias addition at each layer. We propose a modified SpGEMM kernel which iteratively computes sparse inference by computing all operations and a variant of this kernel to be used within one of the parallel inference variants in which the input and output matrices are partitioned.

After analyzing the state-of-the-art inference algorithm, the data-parallel inference, we propose a model-parallel one. These algorithms differ in the way the feature and the weight matrices are partitioned. Partitioning the feature matrix row-wise yields the data-parallel inference, in which each thread executes a series of SpGEMMs interleaved with the activation function without synchronizations, in an embarrassingly parallel fashion. Partitioning the neural network yields the model-parallel inference, which induces a partitioning on the feature matrix and requires barriers to synchronize at layers.

Typically, neural networks do not change after they are trained, and they are used for classification of many new data items. Thus, we propose to obtain more efficient partitioning of the weight matrices during a preprocessing stage, which lowers the data movement incurred by the proposed model-parallel inference algorithm. Optimal sparse matrix partitionings exploit the nonzero structure of matrices and may be obtained using hypergraph partitioning [13,65]. We propose a hypergraph model to transform the neural network into a hypergraph. Using the hypergraph partitioning software as a black box, we obtain two-dimensional partitioning of the weight matrices. We propose loop tiling in the model-parallel variant such that threads compute the sparse inference on small batches of data items that fit cache. We lower parallel inference costs by overlapping

computation with barrier synchronization.

The experiment using the tiling model-parallel inference shows that it obtains the best speedups for $p \leq 8$. Therefore, we propose a tiling hybrid-parallel algorithm to utilize all threads of a machine by combining both parallel variants. The tiling hybrid-parallel method executes the proposed tiling model-parallel variant on each of the row-wise partitions of the feature matrix. Here, the number of threads running the tiling model-parallel inference is the one achieving the best speedup in an earlier experiment. Experimental results show the hybrid-parallel method achieves up to $2\times$ speedup against the state-of-the-art data parallel algorithm running on the same number of threads, provided sufficiently large inference problems.

The work we present in this chapter has been accepted to be published at a conference [48]:

- F. Pawłowski, R. H. Bisseling, B. Uçar, and A. N. Yzelman. "Combinatorial tiling for sparse neural networks" in proc. 2020 IEEE High Performance Extreme Computing (HPEC), September, 2020, Waltham, MA, United States (accepted to be published). This paper describes the proposed algorithms and presents our experiments; the thesis contains extended material.

With a unique view of the whole inference computation as a single matrix, and related algorithms with optimized performance based on this view, the paper received one of the Innovation Awards at the IEEE HPEC Graph Challenge 2020.

Chapter 2

Tensor computations

In this chapter, we investigate high-performance dense tensor computations. We focus on the tensor–vector multiplication kernel and methodologically develop storage, sequential algorithms, and parallel algorithms for this operation. This is a core tensor operation and forms the building block of many algorithms [4]. Furthermore, it is bandwidth-bound and hence its high performance implementation is a challenging endeavor.

Computation on dense tensors, treated as multidimensional arrays, revolve around generalized basic linear algebra subroutines (BLAS). These computations usually apply to one mode of the tensor, as in the right or left multiplication of a matrix with a vector. We propose a novel data structure in which tensors are blocked and blocks are stored in Morton order. This data structure and the associated algorithms bring high performance. They also induce efficiency regardless of which mode a generalized BLAS call is invoked for. We coin the term *mode-oblivious* to describe data structures and algorithms that induce such behavior. The proposed sequential tensor–vector multiplication kernels not only demonstrate superior performance over two state-of-the-art implementations by up to 18%, but additionally show that the proposed data structure induces a 71% less sample standard deviation across d modes, where d varies from 2 to 10. We show that the proposed data structure and the associated algorithms are useful in applications, by implementing a tensor analysis method called the higher order power method (HOPM) [17,18]. Experiments demonstrate up to 38% higher performance with our methods over the state-of-the-art. We then design an efficient shared-memory tensor–vector multiplication algorithm based on a one-dimensional partitioning of the tensor. We prove that one-dimensional partitionings are asymptotically optimal in terms of communication complexity when multiplying an input tensor with a vector on each dimension. We implement a number of alternatives using OpenMP and compare them experimentally. Experimental results for the parallel tensor–vector multiplication on up to 8 socket systems show near peak performance for the proposed algorithms.

This chapter is organized as follows. We first introduce the problem of dense tensor computations, the tensor–vector multiplication (TVM), and the notation we use in Section 2.1. Section 2.2 presents an overview of related work in tensor computations. We then start discussing the sequential tensor–vector multiplication in Section 2.3. We describe a blocking-based storage of dense tensors (Section 2.3.1), two state-of-the art al-

gorithms for the tensor–vector multiplication (Section 2.3.2), and the new tensor–vector multiplication algorithms based on the proposed blocking layouts (Section 2.3.3). We then complete the investigation on sequential *TVM* by presenting a large set of experiments (Section 2.3.4). Completing the investigation of the sequential *TVM* with the experiments allows us to turn our attention to shared-memory parallel *TVM*.

Building on the results obtained for the sequential *TVM*, we propose shared-memory algorithms for *TVM* in Section 2.4. There, we consider a shared-memory *TVM* algorithm based on a for-loop parallelization (Section 2.4.1), reflecting the state-of-the-art. We then discuss that one-dimensional tensor partitionings are asymptotically optimal in terms of communication complexity (Section 2.4.2). We then describe a number of parallel *TVM* algorithms based on one-dimensional tensor partitionings (Section 2.4.3), which is followed by the data movement complexity analyses of all presented parallel *TVM* algorithms (Section 2.4.4) and the experimental comparisons (Section 2.4.5).

We conclude the chapter by giving a summary of our results on the sequential and parallel *TVM* in Section 2.5.

This chapter synthesizes two publications. The first one [50], presents the novel mode-oblivious storage and the associated sequential *TVM* kernel. The second one [49] discusses the parallel tensor–vector multiplication algorithms.

2.1 Introduction

An *order-d* tensor consists of d dimensions, and a *mode* $k \in \{0,\ldots,d-1\}$ refers to one of its d dimensions. We use a calligraphic font to denote a tensor, e.g., $\mathcal{A}$, boldface capital letters for matrices, e.g., $\mathbf{A}$, and boldface lowercase letters for vectors, e.g., $\mathbf{x}$. This standard notation is taken in part from Kolda and Bader [35]. Tensor (and thus, matrix) elements are represented by lowercase letters with subscripts for each dimension. When a subtensor, matrix, vector, or an element of a higher order object is referred, we retain the name of the parent object. For example, $a_{i,j,k}$ is an element of a tensor $\mathcal{A}$. We shall use a flat notation to represent a tensor. For example, the following is an order-3 tensor, where the slices 0 (lower left quadrant) and 1 (top right quadrant) are visually separated:

$$\mathcal{B} = \begin{pmatrix} & & & & 41 & 43 & 47 & 53 \\ & & & & 59 & 61 & 67 & 71 \\ & & & & 73 & 79 & 83 & 89 \\ 2 & 3 & 5 & 7 & & & & \\ 11 & 13 & 17 & 19 & & & & \\ 23 & 29 & 31 & 37 & & & & \end{pmatrix} \in \mathbb{R}^{3\times 4\times 2}. \tag{2.1}$$

We assume tensors have real values; although the discussion can apply to other number fields.

Let $\mathcal{A} \in \mathbb{R}^{n_0\times n_1\times\cdots\times n_{d-1}}$ be an order-d tensor. We use $n = \prod_{k=0}^{d-1} n_k$ to denote the total number of elements in $\mathcal{A}$, and $I_k = \{0,1,\ldots,n_k-1\}$ to denote the index set for mode $k \in \{0,1,\ldots,d-1\}$ of size n_k. Then, $I = I_0 \times I_1 \times \cdots \times I_{d-1}$ is the Cartesian product of

all index sets, whose elements are marked with boldface letters $\mathbf{i}$ and $\mathbf{j}$. For example, $a_{\mathbf{i}}$ is an element of $\mathcal{A}$ whose indices are $\mathbf{i} = i_0, \ldots, i_{d-1}$. We use Matlab colon notation for denoting all indices in a mode. A mode-k fiber $\mathbf{a}_{i_0,\ldots,i_{k-1},:,i_{k+1},\ldots,i_{d-1}}$ is a vector obtained by fixing the indices in all modes except mode k. A hyperslice is a tensor obtained by fixing one of the indices, and varying all others. For example, for third order tensors, a hyperslice becomes a slice, and therefore, a matrix, i.e., $\mathbf{A}_{i,:,:}$ is the ith mode-1 slice of $\mathcal{A}$.

The k-mode tensor–vector multiplication (TVM) multiplies the input tensor with a suitably sized vector $\mathbf{x}$ along a given mode k and is denoted by the symbol $\times_k$. Formally,

$$\mathcal{Y} = \mathcal{A} \ \times_k \ v \ \text{ where } \ \mathcal{Y} \in \mathbb{R}^{n_0 \times n_1 \times \ldots n_{k-1} \times 1 \times n_{k+1} \ldots \times n_{d-1}} \ ,$$

where for all $i_0, i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{d-1}$,

$$y_{i_0,\ldots,i_{k-1},1,i_{k+1},\ldots,i_{d-1}} = \sum_{i_k=0}^{n_k-1} a_{i_0,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{d-1}} v_{i_k},$$

Here, $y_{i_0,\ldots,i_{k-1},1,i_{k+1},\ldots,i_{d-1}}$ is an element of $\mathcal{Y}$, and $a_{i_0,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{d-1}}$ is an element of $\mathcal{A}$. The kth mode of the output tensor $\mathcal{Y}$ is of size one. The above formulation is a contraction of the tensor along the kth mode. Thus, we assume that the operation does not drop the contracted mode, and the resulting tensor is always d-dimensional; for the advantages of this formulation see Bader and Kolda [4, Section 3.2].

The number of floating point operations (flops) of a k-mode TVM is $2n$. The minimum number of data elements touched is:

$$n + \frac{n}{n_k} + n_k \ , \tag{2.2}$$

where n is the size of the input tensor, $\frac{n}{n_k}$ is the size of the output tensor, and n_k is the size of the input vector. This makes the operation special from the computational point of view. The size of one of its inputs, $\mathcal{A}$, is much greater than the other input, v. The arithmetic intensity of a k-mode TVM is the ratio of its floating point operations to its memory accesses, which in our case is

$$\frac{2n}{w(n + \frac{n}{n_k} + n_k)} \ \textit{flops per byte} \ , \tag{2.3}$$

where w is the number of bytes required to store a single element. This lies between $1/w$ and $2/w$ and thus amounts to a heavily bandwidth-bound computation even for sequential execution. The matrix–vector multiplication operation is in the same range of arithmetic intensity. The multi-threaded case is even more challenging, as cores on a single socket compete for the same local memory bandwidth.

We summarize the symbols used in this chapter in Table 2.1.

2.2 Related work

To the best of our knowledge, ours is the first work discussing a blocking approach for obtaining efficient, mode-oblivious sequential and parallel tensor computations. Other

$\mathcal{A}, \mathcal{Y}$	An input and output tensor, respectively
$\mathbf{x}$	An input vector
d	The order of $\mathcal{A}$ and one plus the order of $\mathcal{Y}$
n_i	The size of $\mathcal{A}$ in the ith dimension
n	The number of elements in $\mathcal{A}$
I_i	The index set corresponding to n_i
I	The Cartesian product of all I_i
$\mathbf{i}$ and $\mathbf{j}$	Members of I
k	The mode of a *TVM* computation
b	Individual block size of tensors blocked using hypercubes
s	The ID of a given thread
P	The set of all possible thread IDs
π	Any distribution of $\mathcal{A}$
$\pi_{1\text{D}}$	A 1D block distribution
$b_{1\text{D}}$	The block size of a load-balanced 1D block distribution
ρ_π	A unfold layout for storing a tensor
ρ_Z	A Morton order layout for storing a tensor
$\rho_Z \rho_\pi$	Blocked tensor layout with a Morton order on blocks
m_s	The number of fibers in each slice under a 1D distribution
$\mathcal{A}_s, \mathcal{Y}_s$	Thread-local versions of $\mathcal{A}, \mathcal{Y}$

Table 2.1 – Notation used throughout this chapter for the *TVM* operation $\mathcal{Y} \leftarrow \mathcal{A} \times_k v$.

work that uses space-filling curves include Lorton and Wise [41] who use the Morton order within a blocked data structure for dense matrices, for the matrix–matrix multiplication operation. Yzelman and Bisseling [71] discuss the use of the Hilbert space-filling curve for the sparse matrix–vector multiplication, combined with blocking [72]. Both studies are motivated by cache-obliviousness and did not consider mode-obliviousness. Walker [66] investigates Morton ordering for 2D arrays to obtain efficient memory access in parallel systems for matrix multiplication, Cholesky factorization and fast Fourier transform algorithms. In recent work [39], Li et al. propose a data structure for sparse tensors which uses the Morton order to sort individual nonzero elements of a sparse tensor to organize them in blocks, for efficient representation of sparse tensors.

A dense tensor–vector multiplication routine may be expressed in BLAS2 routines. There are many BLAS implementations, including OpenBLAS [70], ATLAS [69], and Intel MKL [30]. BLIS is a code generator library that can emit BLAS kernels which operate without the need to reorganize input matrices when the elements are strided [64]. However, strided algorithms tend to perform worse than direct BLAS calls when those calls can be made [38].

Early approaches to tensor kernels reorganize the whole tensor in the memory, a so-called tensor unfolding, to then complete the operation using a single optimized BLAS call directly [35]. The unfolding-based approach not only requires unfolding of the input, but also requires unfolding of the output. Li et al. [38] instead propose a parallel loop-based algorithm for the *TMM* kernel: a loop of the BLAS3 kernels, which operate in-place on parts of the tensor such that no unfold is required. They propose an auto-tuning approach based on heuristics and two microbenchmarks and use heuristics to decide on the size of the *MM* kernel and the distribution of the threads among the loops. A recent study [7] proposes a parallel loop-based algorithm for the *TVM* using a similar approach. Ballard et al. [6] investigate the communication requirements of a well-known operation called MTTKRP and discuss a blocking approach. MTTKRP is usually formulated by matrix–matrix multiplication using BLAS libraries. Kjolstad et al. [34] propose The Tensor Algebra Compiler (taco) for tensor computations. It generates code for different modes of a tensor according to the operands of a tensor algebraic expression. Supported formats aside from the dense unfolded storage are sparse storages such as Compressed Sparse Row (otherwise known as Compressed Row Storage, CSR/CRS), its column-oriented variant, and (by recursive use of CSR/CSC) Compressed Sparse Fibers [54].

A related and more computationally involved operation, tensor–tensor multiplication (*TTM*), or tensor contraction, has received considerable attention. This operation is the most general form of the multiplication operation in (multi)linear algebra. CTF [55], TBLIS [42], and GETT [56] are recent libraries carrying out this operation based on principles and lessons learned from high performance matrix–matrix multiplication. Apart from not explicitly considering *TVM*, they do not adapt the tensor layout. As a consequence, they all require transpositions, one way or another. Our *TVM* routines address a very special case of *TMM*.

2.3 Sequential tensor–vector multiplication

In this section, we first describe tensor storages including our proposed blocked data structure for efficient, mode-oblivious performance in Section 2.3.1. We then discuss the various ways for implementing the *TVM*. A *TVM* operation is similar to a matrix–vector multiplication (*MVM*) operation. It takes a tensor, the index of a mode, and a vector of size conformal to that mode's size and performs scalar multiply and add operations. Therefore, we take advantage of the existing BLAS2 routines concerning the *MVM* kernel, which are the left-hand sided multiplication *vm* ($\mathbf{u} = \mathbf{vA}$), and the right-hand sided multiplication *mv* ($\mathbf{u} = \mathbf{Av}$). Section 2.3.2. presents the state-of-the-art *TVM* algorithms, which enable the use of standard BLAS2 routines in the state-of-the-art BLAS libraries. Finally, we propose two block algorithms to perform tensor–vector multiplication in Section 2.3.3. We consider only the case where smaller tensor blocks are stored in an unfolded fashion and are processed using one or more BLAS2 calls. A large set of experiments (Section 2.3.4) are carried out to tune the parameters of the resulting sequential algorithms to be used later in developing parallel algorithms.

2.3.1 Tensor layouts

A layout of a tensor defines the order in which tensor elements are stored in computer memory. We always assume that a tensor is stored in a contiguous memory area. Specifically, a layout $\rho(\mathcal{A})$ is a function which maps tensor elements $a_{i_0,i_1,\ldots,i_{d-1}}$ onto an array of size n:

$$\rho(\mathcal{A}) : \{0,1,\ldots,n_0-1\} \times \{0,1,\ldots,n_1-1\} \times \cdots \times \{0,1,\ldots,n_{d-1}-1\} \mapsto \{0,1,\ldots,n-1\}.$$

For example, the layout $\rho(\mathcal{B})$ maps $\mathcal{B}$'s elements to a contiguous block of memory storing $3 \cdot 4 \cdot 2 = 24$ elements. For performance, we do not modify the data structure while performing a *TVM*.

Most commonly, dense tensors are stored as multidimensional arrays, i.e., in an *unfolded* fashion. While a matrix can be unfolded in two different ways (row-major and column-major), d-dimensional tensors can be stored in $d!$ different ways. Let $\rho_\pi(\mathcal{A})$ be a layout and π an associated permutation of $(0,1,\ldots,d-1)$ such that

$$\rho_\pi(\mathcal{A}) : (i_0, i_1, \ldots, i_{d-1}) \mapsto \sum_{k=0}^{d-1} \left(i_{\pi_k} \prod_{j=k+1}^{d-1} n_{\pi_j} \right) , \tag{2.4}$$

with the convention that products over an empty set amount to 1, that is $\prod_{j=d}^{d-1} n_{\pi_j} = 1$. Conversely, the ith element in memory corresponds to the tensor element with coordinates given by the inverse of the layout $\rho_\pi^{-1}(\mathcal{A})$:

$$i_k = \left\lfloor \frac{i}{\prod_{j=k+1}^{d-1} n_{\pi_j}} \right\rfloor \mod n_k, \text{ for all } k \in \{0,\ldots,d-1\} .$$

For matrices, this relates to the concept of row-major and column-major layout, which, using the layout definition (2.4), correspond to $\rho_{(0,1)}(\mathbf{A})$ and $\rho_{(1,0)}(\mathbf{A})$, respectively. Such a permutation-based layout is called a tensor *unfolding* [35] and describes the case where a tensor is stored as a regular multidimensional array.

Let $\rho_Z(\mathcal{A})$ be a *Morton layout* defined by the space-filling Morton curve [45]. The Morton order is defined recursively, where at every step the covered space is subdivided into two within every dimension; for 2D planar areas this creates four cells, while for 3D it creates eight cells. In every two dimensions the order between cells is given by a (possibly rotated) Z-shape. Let w be the number of bits used to represent a single coordinate, and let $i_k = (l_0^k l_1^k \dots l_{w-1}^k)_2$ for $k = \{0, 1, \dots, d-1\}$ be the bit representation of each coordinate. The Morton order in d dimensions $\rho_Z(\mathcal{A})$ can then be defined as

$$\rho_Z(\mathcal{A}) : (i_0, i_1, \dots, i_{d-1}) \mapsto (l_0^0 l_0^1 \dots l_0^{d-1} l_1^0 l_1^1 \dots l_1^{d-1} \dots l_{w-1}^0 l_{w-1}^1 \dots l_{w-1}^{d-1})_2 \,. \tag{2.5}$$

The inverse $\rho_Z^{-1}(\mathcal{A})$ yields the coordinates of the ith consecutively stored element in memory, where $i = (l_0 l_1 \dots l_{d^w-1})_2$:

$$\rho_Z^{-1}(\mathcal{A}) : i \mapsto (i_0, i_1, \dots, i_{d-1}) \quad \text{where} \quad i_k = (l_{k+0d} l_{k+1d} \dots l_{k+(w-1)d})_2 \,, \tag{2.6}$$

for all $k \in \{0, 1, \dots, d-1\}$. Such layout improves performance on systems with multi-level caches due to the locality preserving properties of the Morton order. However, $\rho_Z(\mathcal{A})$ is an irregular layout, and thus unsuitable for processing with standard BLAS routines.

Let $M_{\rho_\pi}^{k \times l}(\mathcal{A})$ be the *matricization* of $\mathcal{A}$ which views a tensor layout $\rho_\pi(\mathcal{A})$ as a $k \times l$ matrix:

$$M_{\rho_\pi}^{k \times l}(\mathcal{A}) : \mathbb{R}^{n_0 \times n_1 \times \cdots \times n_{d-1}} \mapsto \mathbb{R}^{k \times l} \,,$$

where π is a permutation of $(0, \dots, d-1)$, $k = \Pi_{k=0}^{b} n_{\pi_k}$ for some $0 \leqslant b \leqslant d-1$, and $l = n/k$. We relate the entries $(M_{\rho_\pi}^{k \times l}(\mathcal{A}))_{i,j}$ to $\mathcal{A}_{i_0, i_1, \dots, i_{d-1}}$ by

$$i = \sum_{k=0}^{b} i_{\pi_k} \prod_{j=k+1}^{b} n_{\pi_j} \quad \text{and} \quad j = \sum_{k=b+1}^{d-1} i_{\pi_k} \prod_{j=k+1}^{d-1} n_{\pi_j} \,,$$

where $i \in \{0, 1, \dots, k-1\}$, $j \in \{0, 1, \dots, l-1\}$ and $i_k \in \{0, 1, \dots, n_k - 1\}$. For example, $M_{\rho_\pi}^{3 \times 8}(\mathcal{B})$ corresponds to the following $n_0 \times n_1 n_2$ matricization of $\rho_{(0,1,2)}(\mathcal{B})$ (2.1):

$$M_{\rho_\pi}^{3 \times 8}(\mathcal{B}) = \begin{pmatrix} 2 & 41 & 3 & 43 & 5 & 47 & 7 & 53 \\ 11 & 59 & 13 & 61 & 17 & 67 & 19 & 71 \\ 23 & 73 & 29 & 79 & 31 & 83 & 37 & 89 \end{pmatrix} \in \mathbb{R}^{3 \times 8} \,.$$

A *blocked tensor* is a tensor with smaller equally-sized tensors as its elements. Formally, an order-d blocked tensor $\mathcal{A} \in \mathbb{R}^{n_0 \times n_1 \times \cdots \times n_{d-1}}$ consists of a total of $\prod_{i=0}^{d-1} a_i$ blocks $\mathcal{A}_j \in \mathbb{R}^{b_0 \times b_1 \times \cdots \times b_{d-1}}$, where $j \in \{0, \dots, (\prod_{i=0}^{d-1} a_i) - 1\}$ and $n_k = a_k b_k$ for all $k \in \{0, 1, \dots, d-1\}$. A *blocked layout* organizes elements into blocks by storing the blocks consecutively in memory while the blocks themselves use a uniform layout to store their elements. Formally, a blocked layout $\rho_0 \rho_1(\mathcal{A})$ stores a block as the $\rho_0(\mathcal{A})(i_0, i_1, \dots, i_{d-1})$th block in the

memory occupied by the tensor, while a scalar is stored as the $\rho_1(A_0)(i_0, i_1, \ldots, i_{d-1})$th in the memory occupied by the block. It is thus a combination of two layouts: ρ_0 at the block-level, and ρ_1 within blocks. We propose two blocked layouts which vary in the layout used for the blocks: (i) $\rho_\pi\rho_\pi$, where the blocks are ordered using a permutation of dimensions; and (ii) $\rho_Z\rho_\pi$, where blocks are ordered according to the Morton order [45].

2.3.2 Two state-of-the-art tensor–vector multiplication algorithms

Assuming a matrix $\mathbf{A}$ is stored using $\rho_{(0,1)}$-layout, the BLAS subroutines vm and mv effectively compute mode-0 TVM and mode-1 TVM of $\mathbf{A}$, respectively. In general, for any tensor stored using a ρ_π layout, a TVM may be carried out using the MVM kernel (using BLAS2) either by:

- reorganizing the tensor in memory (*unfolding* the tensor into a matrix) followed by a single MVM, or
- reinterpreting the tensor as a series of matrices, on which a series of MVM operations are executed.

These two approaches result in two highly optimized state-of-the-art algorithms. Assuming a tensor with ρ_π layout, Algorithm 2.1 computes a k-mode TVM by repeatedly invoking a column-major MVM on consecutive parts of the tensor in-place, by matricization. Algorithm 2.2, instead, computes a k-mode TVM by reordering the tensor in memory such that the data is aligned for a single column-major MVM. Both take care of the position of the mode in the permutation such that the appropriate MVM routine is chosen.

Both algorithms rely on a single MVM kernel for the case when $\pi_0 = k$ or $\pi_{d-1} = k$, in which case the memory touched explicitly by the MVM kernel corresponds to the minimum number of elements touched (2.2). For the remaining $d-2$ modes, the two algorithms exhibit different behavior. Algorithm 2.1 touches at least

$$\prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i} + \prod_{i=\pi_k^{-1}+1}^{d-1} n_{\pi_i} + n_k \tag{2.7}$$

data elements for each of the $\prod_{i=0}^{\pi_k^{-1}-1} n_{\pi_i}$ MVM calls. This brings the data movement overhead of Algorithm 2.1 to

$$\left(\left[\prod_{i=0}^{\pi_k^{-1}-1} n_{\pi_i} \right] - 1 \right) n_k. \tag{2.8}$$

Algorithm 2.2 performs an explicit unfold of the tensor memory which instead incurs a more signifcant overhead of $2n$. We choose a $\rho_{(1,0)}$-layout for the unfolded $\mathbf{U}$ instead of a $\rho_{(0,1)}$-layout since the latter would require element-by-element copies, while the former can copy ranges of size $\prod_{i=\pi_k^{-1}+1}^{d-1} n_{\pi_i}$. Furthermore, the former accesses the input tensor consecutively while individual accesses on the unfold matrix are interleaved; this is faster than the reverse.

Algorithm 2.1 *tvLooped*$(\mathcal{A}, \mathbf{v}, k, \pi)$: The looped tensor–vector multiplication.

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ tensor $\mathcal{A}$ with $\rho_\pi(\mathcal{A})$,
an $n_k \times 1$ vector $\mathbf{v}$,
a mode of multiplication $k \in \{0, 1, \ldots, d-1\}$,
a permutation of modes π.
Output: An $n_0 \times n_1 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ tensor $\mathcal{B}$,
$\mathcal{B} = \mathcal{A} \times_k \mathbf{v}$ with $\rho_\pi(\mathcal{B})$.

1: $n = \prod_{i=0}^{d-1} n_i$ ▶ Number of tensor elements.
2: **if** π_{d-1} **equals** k **then**
3: Let $\mathbf{A} = M_{\rho_\pi}^{n/n_k \times n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a tall-skinny $\rho_{(0,1)}$-matrix.
4: $\mathbf{u} \leftarrow mv(\mathbf{A}, \mathbf{v})$ ▶ A single mv computes $\mathcal{B}$.
5: **return** $\mathcal{B} = (M_{\rho_\pi}^{n/n_k \times 1})^{-1}(\mathbf{u})$ ▶ Reinterpret $\mathbf{u}$ as a tensor.
6: **else**
7: Let $r = \prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i}$ and $s = r/n_k$
8: Let $\mathbf{A} = M_{\rho_\pi}^{(n/r) n_k \times s}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as n/r wide $\rho_{(0,1)}$-matrices.
9: Let $\mathbf{B}$ be an $n/r \times s$ matrix with layout $\rho_{(0,1)}$ ▶ n/r vectors of length s.
10: **for** $i = 0$ to $(n/r) - 1$ **do**
11: $\mathbf{b}_{i,:} \leftarrow vm(\mathbf{v}^T, \mathbf{A}_{in_k:(i+1)n_k,:})$ ▶ ith row of $\mathbf{B}$ computed.
12: **return** $\mathcal{B} = (M_{\rho_\pi}^{n/r \times s})^{-1}(\mathbf{B})$ ▶ Reinterpret $\mathbf{B}$ as a tensor.

2.3.3 Block tensor–vector multiplication algorithms

We store blocks with ρ_π layout to take advantage of the *TVM* algorithms from Section 2.3.2 that exploit highly optimized BLAS2 routines. Algorithm 2.3 is a general block *TVM* algorithm which visits the blocks in the order imposed by any layout ρ_0. When the *TVM* of a block finishes, the next block offset in the output tensor and the associated positions of the vector entry are computed using the *nextBlock* function, which implements the block order.

Depending on the layout ρ_0, the *nextBlock* function in Algorithm 2.3 then corresponds to $nextBlock_{\rho_\pi}$ or $nextBlock_{\rho_Z}$. The $nextBlock_{\rho_\pi}$ function (Algorithm 2.4) has an efficient $\Theta(1)$ implementation which avoids explicitly evaluating ρ_π and ρ_π^{-1}.

The $nextBlock_{\rho_Z}$ function (Algorithm 2.5) when moving from one block to the next increments the result index by default and and the block coordinates according to the Morton order using *mortonInc* function (Line 2), which is a modified version of a binary counter. If it increments coordinate i_k, then the $nextBlock_{\rho_Z}$ function swaps the incremented result index with an index stored in *resultIndices*[*lvl*] (Line 4), where *lvl* is the level of recursion of the Morton curve. It does not explicitly evaluate ρ_Z and ρ_Z^{-1}, but instead requires Algorithm 2.3 to maintain a counter for each dimension $(i_0, \ldots, i_{d-1})$ and an array for result indices at each level of the Morton order *resultIndices* yielding the memory overhead of $\Theta(d + \log_2 \max_i a_i)$. The amortized analysis [15, ch. 17] of the *mortonInc* function yields a run time complexity of $\Theta(\Pi_{i=0}^{d-1} a_i)$ over the whole $\rho_Z \rho_\pi$-block

Algorithm 2.2 *tvUnfold*$(\mathcal{A}, \mathbf{v}, k, \pi)$: The unfold tensor–vector multiplication.

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ tensor $\mathcal{A}$ with $\rho_\pi(\mathcal{A})$,
an $n_k \times 1$ vector $\mathbf{v}$,
a mode of multiplication $k \in \{0, 1, \ldots, d-1\}$,
a permutation of modes π.

Output: An $n_0 \times n_1 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ tensor $\mathcal{B}$,
$\mathcal{B} = \mathcal{A} \times_k \mathbf{v}$ with $\rho_\pi(\mathcal{B})$.

1: $n = \prod_{i=0}^{d-1} n_i$ ▶ Number of tensor elements.
2: **if** π_{d-1} **equals** k **then**
3: Let $\mathbf{A} = M_{\rho_\pi}^{n/n_k \times n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a tall-skinny $\rho_{(0,1)}$-matrix.
4: $\mathbf{u} \leftarrow mv(\mathbf{A}, \mathbf{v})$
5: **return** $\mathcal{B} = (M_{\rho_\pi}^{n/n_k \times 1})^{-1}(\mathbf{u})$
6: **else if** π_0 **equals** k **then**
7: Let $\mathbf{A} = M_{\rho_\pi}^{n_k \times n/n_k}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as a wide $\rho_{(0,1)}$-matrix.
8: $\mathbf{u} \leftarrow vm(\mathbf{v}^T, \mathbf{A})$
9: **return** $\mathcal{B} = (M_{\rho_\pi}^{1 \times n/n_k})^{-1}(\mathbf{u})$
10: **else**
11: Let $r = \prod_{i=\pi_k^{-1}}^{d-1} n_{\pi_i}$ and $s = r/n_k$
12: Let $\mathbf{A} = M_{\rho_\pi}^{(n/r)n_k \times s}(\mathcal{A})$ ▶ Reinterpret $\mathcal{A}$ as n/r wide matrices.
13: Let $\mathbf{U}$ be an empty $n_k \times n/n_k$ matrix with layout $\rho_{(0,1)}$
14: **for** $i = 0$ to $(n/r) - 1$ **do**
15: **for** $j = 0$ to $n_k - 1$ **do**
16: $\mathbf{u}_{j,is:(i+1)s} \leftarrow \mathbf{a}_{in_k+j,:}$ ▶ Rearrange $\mathbf{A}$ into $\mathbf{U}$ (tensor unfolding).
17: $\mathbf{x} \leftarrow vm(\mathbf{v}^T, \mathbf{U})$ ▶ A single vm can now compute $\mathcal{B}$.
18: **return** $\mathcal{B} = (M_{\rho_\pi}^{1 \times n/n_k})^{-1}(\mathbf{x})$ ▶ Reinterpret $\mathbf{x}$ as a tensor.

TVM computation, while the if statement block (Lines 3-7) executes at most

$$\Theta\left(\sum_{i=0}^{log_2 a-1} 2^{di+k} + \sum_{i=0}^{log_2 a-2} 2^{di+log_2 n-2+k}\right) = \Theta(a/2^{d-k})$$

times.

The overhead of $\Theta(d\Pi_{i=0}^{d-1}a_i)$ is much smaller than the number of operations the bandwidth-bound *TVM* performs. Thus, we expect neither the $\rho_\pi\rho_\pi$- nor the $\rho_Z\rho_\pi$-block *TVM* to slow down for this reason while we expect an increased performance in both algorithms due to cache reuse, with more mode-oblivious performance of the $\rho_Z\rho_\pi$-block *TVM* algorithm.

Algorithm 2.3 $btv(\mathcal{A}, \mathbf{v}, k, \pi, nextBlock_{\rho_0}, tv)$: The block tensor–vector algorithm

Input: An $n_0 \times n_1 \times \cdots \times n_{d-1}$ blocked tensor $\mathcal{A}$ with $\rho_0\rho_\pi(\mathcal{A})$ consisting of $\Pi_{i=0}^{d-1}a_i$ blocks $\mathcal{A}_j \in \mathbb{R}^{b_0\times b_1\times\ldots\times b_{d-1}}$,
an $n_k \times 1$ vector $\mathbf{v}$,
a mode of multiplication $k \in \{0, 1, \ldots, d-1\}$,
a permutation of modes π for each block layout,
a $nextBlock_{\rho_0}$ function for indices of result o and vector i_k,
a *TVM* algorithm tv for ρ_π layouts.

Output: An $n_0 \times \cdots \times n_{k-1} \times 1 \times n_{k+1} \times \cdots \times n_{d-1}$ blocked tensor $\mathcal{B}$ consisting of $\Pi_{i=0}^{d-1}a_i/a_k$ blocks $\mathcal{B}_k \in \mathbb{R}^{b_0\times\cdots\times b_{k-1}\times 1\times b_{k+1}\times\cdots\times b_{d-1}}$,
$\mathcal{B} = \mathcal{A} \times_k \mathbf{v}$ with $\rho_0\rho_\pi(\mathcal{B})$.

1: Let $\mathcal{B}$ be a blocked tensor with layout $\rho_0\rho_\pi(\mathcal{B})$ with entries initialized to 0.
2: $(i_0, i_1, \ldots, i_{d-1}) \leftarrow \rho_0^{-1}(\mathcal{A})(0)$ ▶ Get coordinates of the first block.
3: $o \leftarrow \rho_0(\mathcal{B})(i_0, \ldots i_{k-1}, 0, i_{k+1}, \ldots, i_{d-1})$ ▶ Get output block index.
4: **for** $i = 0$ to $\Pi_{j=0}^{d-1}a_j - 1$ **do**
5: $\quad\mathcal{B}_o \leftarrow \mathcal{B}_o + tv(\mathcal{A}_i, \mathbf{v}_{(i_k)b_k:(i_k+1)b_k}, k, \pi)$
6: $\quad(o, i_k) \leftarrow nextBlock_{\rho_0}(k, i, o, i_k)$
7: **return** $\mathcal{B}$

2.3.4 Experiments

We evaluate the proposed blocked tensor layouts for the *TVM* computation, evaluate their mode-obliviousness, and compare the proposed blocked *TVM* algorithms against the state of the art. Section 2.3.4.1 first presents our experimental setup and methodologies. To ascertain practical upper bounds for the performance of a *TVM*, Section 2.3.4.2 presents microbenchmarks designed to find realistic bounds on data movement and computation. We then follow with the assessment of the state-of-the-art *TVM* algorithms in Section 2.3.4.3, the block *TVM* algorithms in Section 2.3.4.4, and compare them with the codes generated by the Tensor Algebra Compiler (taco) in Section 2.3.4.5. To show our proposed tensor layouts are useful in applications, we apply them to the iterative higher-order power method (HOPM) [17, 18] in Section 2.3.4.6.

Algorithm 2.4 $nextBlock_{\rho_\pi}(k, i, o, i_k)$: The output and vector indices used by the next block in a ρ_π layout.

Input: A mode k,
current block index i,
current block result index o,
current block vector index i_k.
Output: A next block result index o
and next block vector index i_k.

1: Let $m_{right} = \prod_{i=\pi_k^{-1}}^{d-1} a_{\pi_i}$ and $m_{mode} = m_{right}/a_k$
2: $i \leftarrow i + 1$
3: **if** $(k > 0)$ **and** $((i$ **mod** $m_{right})$ **equals** $0)$ **then**
4: $o \leftarrow o + 1$
5: $i_k \leftarrow 0$
6: **else if** $((i$ **mod** $m_{mode})$ **equals** $0)$ **then**
7: $o \leftarrow o - m_{mode} + 1$
8: $i_k \leftarrow i_k + 1$
9: **else**
10: $o \leftarrow o + 1$
11: **return** (o, i_k)

2.3.4.1 Setup

We run our experiments on a single Intel Ivy Bridge node, containing two Intel Xeon E5-2690 v2 processors that are each equipped with 10 cores. The cores run at 3.0 GHz with AVX capabilities, amounting to 240 Gflop/s per processor. Each processor has 32 KB of L1 cache memory per core, 256 KB of L2 cache memory per core, and 25 MB of L3 cache memory shared amongst the cores. Each processor has 128 GB of local memory configured in quad-channel at 1600 MHz, yielding a theoretical bandwidth of 47.68 GB/s per socket. The system uses CentOS 7 with Linux kernel 3.10.0 and software is compiled using GCC version 6.1. We use Intel MKL version 2018.1.199 and LIBXSMM version 1.9-864.

Benchmarking methodology. We benchmark tensors of order-two ($d = 2$) up to order-10 ($d = 10$) and for simplicity assume square tensors of size n. We assume users are interested in input tensors that do not fit into cache, and thus choose n such that the combined input and output memory areas during a single TVM call have a combined size of at least several GBs to make sure we capture out-of-cache behavior.

To benchmark a kernel, we first time a single run and calculate the number m of calls required to reach at least one second of run time. We then conduct 10 experiments as follows:

1. issue a sleep command for 1 second,
2. run the kernel once without timing,

Algorithm 2.5 $nextBlock_{\rho_Z}(k, i, o, i_k)$: The output and vector indices used by the next block in a Morton layout.

Input: A mode k,
current block result index o,
current block vector index i_k.
The index i is retained for genericity.
This algorithm requires access to global data structures:
current block coordinates $(i_0, \ldots, i_{d-1})$,
and *resultIndices* array of size $log_2 max_k(n_k)$.
Output: A next block result index o
and next block vector index i_k.

```
1: o ← o + 1                                   ▶ Increment the result index.
2: dim, lvl ← mortonInc(i_0, ..., i_{d-1})     ▶ Increment indices according to ρ_Z^{-1}(A).
3: if dim = k then                             ▶ Reset o if kth dimension was incremented.
4:     swap(resultIndices[lvl], o)
5:     if i_k < n_k − 1 then
6:         for j = 0 to level − 1 do
7:             resultIndices[j] ← o
```

3. time m runs of the kernel,

4. store the time taken divided by m as t_i.

Based on 10 experiments, we compute the average time $t_{\text{avg}} = (\sum_{i=0}^{m-1} t_i)/10$ and the (unbiased) sample standard deviation $t_{\text{std}} = \sqrt{\frac{1}{9} \sum_{i=0}^{9} (t_i - t_{\text{avg}})^2}$. Throughout experiments, we make sure that the t_{std} are less than or equal 5% of t_{avg}, so as to exclude bad hardware and suspicious system states.

Block size selection. We evaluate the performance of block *TVM* algorithms by varying the block size with respect to the cache hierarchy. As for the input tensors, we assume square blocks which have equal length b along all dimensions. Recall that a block *TVM* algorithm relies on the existing *TVM* kernels of Section 2.3.2 being called for each individual block; the size b thus controls the cache level that we block for. We say that a kernel fits level-L cache if the number of elements it touches is less than or equal to $\alpha z_L / s$, where $0 < \alpha \leq 1$ is the cache saturation coefficient, z_L is the level-L cache size in bytes, and s is the size of a single tensor data element in bytes. The saturation coefficient is such that we obtain the typical cache behavior; setting it too close to 1 usually loses performance, while setting it too close to 0 amounts to benchmarking lower level caches. We find that for best performance, b should be even or a multiple of four, presumably to make optimal use of SIMD instructions, and observe typical throughput is attained at $\alpha = 0.5$. Table 2.2 summarizes our choices for b in the b_{L1}, b_{L2} and b_{L3} columns such that

d	b_{L1}	b_{L2}	b_{L3}
2	44	124	1276
3	12	24	116
4	6	10	34
5	4	6	16
6	3	4	10
7	2	4	7
8	2	3	5
9	2	3	4
10	2	-	4

Table 2.2 – Values of b such that square order-d blocks of b^d elements together with the input vector (b elements) and the output block (b^{d-1} elements) all fit in the L1, L2, and L3 caches, assuming double-precision tensor and vector elements. Note that there is no integer b for $d = 10$ such that the *TVM* routine fits the L2 cache without also fitting L1 cache.

the *TVM* of a block touching $b^d + b^{d-1} + b$ elements fits L1, L2 and L3 cache, respectively. Note that this parameter is not fine-tuned for better performance.

Implementation. The *tvUnfold* (Algorithm 2.2) relies on a custom memcpy routine *ntmemcpy*, which flags the source memory for early cache eviction. This leads to better performance once the matrix U is multiplied (line 14), since the full cache is available to work on the unfolded tensor. The *ntmemcpy* uses non-temporal reads followed by aligned or unaligned streaming writes, as appropriate.

Both *tvUnfold* and *tvLooped* rely on the *mv* and *vm* matrix–vector multiplications kernels. We always use MKL if these kernels are used on unblocked tensors, but, when called for *MVMs* on individual blocks of a blocked layout, we also consider LIBXSMM [28]—a library especially optimized for repeated dense small matrix–matrix multiplications. We observe that the performance of the *MVM* kernel strongly depends on the ratio between rows and columns: for short-wide and tall-skinny matrices LIBXSMM usually outperforms MKL, while otherwise MKL exhibits better performance. We tune the selection of MKL or LIBXSMM to our Ivy Bridge machine based on the aspect ratio, the kernel orientation (*mv* or *vm*), and the number of bytes the computation touches.

2.3.4.2 Microbenchmarks

As *TVM* algorithms are bandwidth-bound, we retrieve an upper bound on their performance by benchmarking the peak bandwidth our machine attains in practice. We benchmark using STREAM variants, the C standard *memcpy*, and the hand-coded *ntmemcpy*. To measure an upper bound on computation time for in-cache blocks, we separately benchmark the *mv* and *vm MVM* kernels for cache-sized matrices and for much larger matrices as well, as a proxy for the overall expected *TVM* performance.

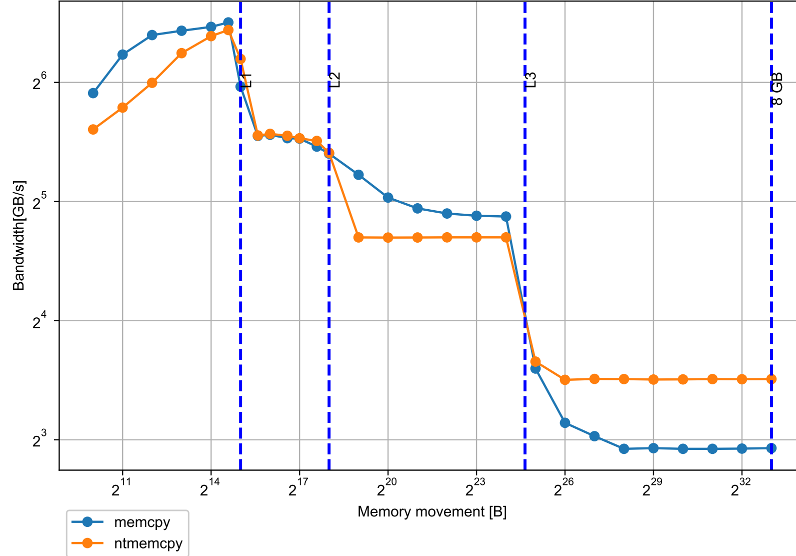

Figure 2.1 – Plot of the effective bandwidth (in GB/s) of the copy kernels versus the amount of bytes moved by the copies, in bytes. The results are stable for sizes larger than 8 GB.

We also investigate the performance and mode-obliviousness of using one of the state-of-the-art *TVM* algorithms on tensors that fit in cache, as this is the inner kernel of the block *TVM* algorithms. Since bandwidth is the overall limiting factor, all measurements are in Gigabyte per second (GB/s).

Upper bounds on effective bandwidth. We measure the maximum bandwidth of our system using several variants of the STREAM benchmark, reporting the maximum measured performance only. Using the full machine we attain 76.7 GB/s (using two processors and ten threads each); however, since our proposed *TVM* algorithms are sequential, STREAM performance of a single core yields the upper bound of interest at 18.3 GB/s. Both results are consistent with the theoretical peak.

The tensor blocked layouts require blocks to be streamed from RAM into cache. For the *tvUnfold*, we unfold the tensor with a series of copies. Figure 2.1 benchmarks the standard *memcpy* and *ntmemcpy* for different sizes, including the cache-sized copies in order to attain upper bounds for data movements when processing a single block. The *ntmemcpy* performance is indeed better for RAM-sized copies since it avoids caching source memory areas. Table 2.3 summarizes the results by selecting representative performance figures for each level of the memory hierarchy.

Matrix–vector multiplication using *mv* and *vm*. The matrix–vector multiplication *mv* or the vector transpose–matrix multiplication *vm* are the innermost kernels of all

Copy kernel	L1	L2	L3	RAM
memcpy	88.41	46.15	29.33	7.62
ntmemcpy	83.78	46.25	25.98	11.39

Table 2.3 – Sample effective bandwidths (in GB/s) of the copy kernels when copy size fits into different levels of the memory hierarchy. These are representative values taken from Figure 2.1 at 16 KB for L1, 128 KB for L2, 16 MB for L3, and 8 GB for RAM.

d	*mv*	*vm*
2	12.79	11.64
3	14.25	9.58
4	10.79	9.83
5	9.45	9.85
6	11.29	9.71
7	12.77	10.03
8	13.52	9.82
9	13.46	10.72
10	13.16	9.43

Table 2.4 – Effective bandwidth (in GB/s) of a single *mv* and *vm*, given a tall-skinny and short-wide $\rho_{(0,1)}$-matrix, respectively. Matrix sizes n are such that at least several GBs of memory are required. The order d determines the aspect ratio of the matrix as n^{d-1} to n. All experiments use MKL.

TVM algorithms. The block algorithms call either *tvLooped* or *tvUnfold* algorithms for individual blocks, while those two algorithms, in turn, execute one or more *MVM* kernels. To gauge the overall computational performance of both in-cache matrices and *tvLooped* and *tvUnfold* algorithms, we benchmark the speed of single calls to *mv* and *vm* over a range of d-dimensional tensors interpreted as tall-skinny or short-wide matrices. Table 2.4 summarizes the results for matrices that do not fit in cache, while Table 2.5 contains those for in-cache matrices.

For large matrices, the *mv* has better performance than the *vm*, since the former operates on the output vector via a single stream, while the latter is forced to either i) access the input matrix with stride, or ii) access the output vector multiple times; which both result in reduced performance. The *mv* attains better performance than Table 2.3 would predict presumably because of the reuse of cached input vector elements, which could only benefit the *vm* if it was implemented using accesses with stride.

Comparing the results for cache-sized matrices to Table 2.3 would indicate that for L1-sized and L2-sized matrices, the *MVM* becomes compute-bound. We also observe that the *vm* outperforms *mv*, especially for lower cache levels and higher d, and that blocking for L2 typically is preferred. Furthermore, the *mv* exhibits slowdowns when the aspect ratio increases while the *vm* is oblivious to it; we exploit this property to attain mode-oblivious behavior for the block *TVM* algorithms.

	mv			*vm*		
d	b_{L1}	b_{L2}	b_{L3}	b_{L1}	b_{L2}	b_{L3}
2	*35.42*	***37.84***	*26.46*	**45.25**	*38.94*	*26.13*
3	*26.21*	***32.51***	*26.18*	38.26	***42.54***	*22.81*
4	20.94	***27.29***	*25.09*	**43.14**	40.07	26.77
5	19.29	*21.08*	***23.39***	29.76	**35.95**	21.26
6	18.47	*18.34*	***23.58***	23.56	**33.70**	31.10
7	12.01	**20.53**	19.84	11.18	**33.36**	28.86
8	14.43	**19.14**	18.96	19.09	**35.88**	27.76
9	16.82	**19.27**	18.59	29.35	**33.27**	26.47
10	18.51	-	18.58	**42.45**	-	26.50

Table 2.5 – Effective bandwidth (in GB/s) of a single *mv* (left) or *vm* (right) given a $b^{d-1} \times b$ and $b \times b^{d-1}$ $\rho_{(0,1)}$-matrix, respectively, with b as defined in Table 2.2. MKL results are in *italics* while LIBXSMM results are in regular font; we report only the best-performing variant. The best results for any given d are in **bold**. Note that there is no integer b for $d = 10$ such that the *TVM* routine fits the L2 cache without also fitting L1 cache.

Single-block tensor–vector multiplication. We discard *tvUnfold* as the inner kernel of the block *TVM* algorithms because it requires a complete unfold of each block which would double the data movement, resulting in a major performance overhead at least for L2- and L3-sized tensors. Therefore, we will only use *tvLooped* as the inner kernel of the block *TVM* algorithms.

We measure the performance of *tvLooped* (Algorithm 2.1) on cache-sized tensors in Table 2.6, for each mode $0 \leq k < d$, and report the average performance over all modes (left). Additionally, we measure the unbiased sample standard deviation between the modes (right) as a measure of mode-obliviousness—the lower this value, the more consistent the performance when computing in arbitrary modes. The *tvLooped* algorithm uses the *vm* kernel for all modes $k < d - 1$ and uses the slower-performing and less mode-oblivious *mv* only for the mode $k = d - 1$.

We achieve best performance on L2-sized tensors, which is consistent with the microbenchmarks for the *vm* and *mv* kernels. However, we achieve best mode-obliviousness for L3-sized blocks, while a lower d benefits both performance and mode-obliviousness for both L2 and L3. Since in practical application we still require retrieving the input blocks from main memory and since both L2 and L3 performances are higher than the bounds in Table 2.4, we expect the best results for L3-sized blocks.

2.3.4.3 The state-of-the-art tensor–vector multiplication algorithms

Here, we benchmark *tvLooped* and *tvUnfold* algorithms for large tensors with general unfolded layouts ρ_π: *tvlooped* (Algorithm 2.1), which repeatedly makes BLAS2 calls over the given input tensor as-is, and *tvUnfold* (Algorithm 2.2) which first unfolds the input tensor into a layout appropriate for the multiplication mode and then calls BLAS2 once.

	Average performance			Sample stddev. (%).		
d	b_{L1}	b_{L2}	b_{L3}	$\text{std}_{b_{L1}}$	$\text{std}_{b_{L2}}$	$\text{std}_{b_{L3}}$
2	**40.34**	38.39	25.84	17.23	2.03	**0.68**
3	33.08	**35.56**	25.11	18.74	17.05	**7.83**
4	31.95	**33.69**	24.76	35.14	16.29	**5.86**
5	24.30	**29.64**	22.87	21.66	21.03	**17.96**
6	20.11	27.53	**28.37**	18.69	24.24	**11.48**
7	10.31	**28.96**	25.70	**11.98**	19.45	15.60
8	15.57	**26.39**	24.64	20.66	27.39	**14.48**
9	22.64	**26.72**	25.55	29.83	24.29	**13.21**
10	**30.68**	-	24.58	36.81	-	**11.83**

Table 2.6 – Average effective bandwidth (in GB/s) and relative standard deviation of *tvLooped* (Algorithm 2.1), in percentage versus the average bandwidth over all possible $k \in \{0, 1, \ldots, d-1\}$. The input tensor is a square block of size b as given in Table 2.2. The highest bandwidth and lowest standard deviation for each d are stated in **bold**. Note that there is no integer b for $d = 10$ such that the TVM routine fits the L2 cache without also fitting L1 cache.

The *tvLooped* algorithm. Table 2.7 shows the results of *tvLooped* for $k \in \{1, \ldots, d-2\}$; like for Table 2.4, these are higher than the raw memory-copy speeds in Table 2.3 due to cache reuse. We omit the results for $k = 0$ and $k = d-1$ since their equivalence to a single *vm* and *vm*, respectively, for which the results are in Table 2.4.

We previously learned that mode $k = d-1$ (*mv*) is preferred over $k = 0$ (*vm*) for matrices that cannot be cached; Table 2.7, however, shows several modes $0 < k < d-1$ that exhibit higher performance than a single *mv* call. This is due to *tvLooped* dividing the computation into multiple *vm* calls on smaller matrices: when the output matrix fits cache, the number of cache misses may be significantly reduced. To test this hypothesis, the results which correspond to calls to *vm* on matrices for which the output matrix fits L2 cache size (and not L1) are printed in *italic*, while marking the best results in **bold**. From the table, we indeed observe that the fastest results for given d are obtained for computations making optimal use of the L2 cache.

For all d, one may observe that the performance decreases with increasing k, even if the output matrix fits in the L2 cache. This conforms to the data movement overhead (2.7) of input vector elements.

The *tvUnfold* algorithm. Table 2.8 benchmarks *tvUnfold* (Algorithm 2.2) for large tensors for $k \in \{1, \ldots, d-2\}$. The results for modes 0 and $d-1$ are equivalent to those of a single *vm* and *vm* in Table 2.4, respectively.

The *tvUnfold* performance is suboptimal as it is equivalent to performing two operations in sequence: a large memory copy (the unfold), followed by a single *vm*. These are bounded by the copy and MVM microbenchmarks of Section 2.3.4.2, respectively. The performance of *tvUnfold* thus is half of the fastest one at best and half of the slowest one at worst; indeed, most results are very close to the raw *memcpy* performance. The effec-

$d\backslash k$	1	2	3	4	5	6	7	8	avg_{tvU}	std_{tvU}
2	-	-	-	-	-	-	-		12.22	6.66
3	***13.57***	-	-	-	-	-	-	-	12.47	20.24
4	***11.28***	11.25	-	-	-	-	-	-	10.79	6.27
5	11.16	***13.03***	10.05	-	-	-	-	-	10.71	13.49
6	9.92	10.98	***13.11***	10.87	-	-	-	-	10.98	11.07
7	10.18	11.32	10.81	***12.81***	10.89	-	-	-	11.26	10.07
8	10.24	11.44	11.39	***12.54***	10.46	9.62	-	-	11.13	12.29
9	10.41	10.70	10.64	11.32	***12.56***	9.85	7.72	-	10.82	14.98
10	9.18	9.86	10.84	9.08	11.67	***12.59***	10.11	6.65	10.26	18.56

Table 2.7 – Effective bandwidth (in GB/s) of *tvLooped* (Algorithm 2.1), for $k \in \{1, \ldots, d-2\}$ for each d. The two columns on the right are the average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) for $k \in \{0, 1, \ldots, d-1\}$. Tensor sizes n are such that at least several GBs of memory are required. For each d, the best bandwidth (between modes 1 and $d-2$) is in bold, while the result for which the *MVM* kernel fits L2 is in italics. All experiments use MKL.

$d\backslash k$	1	2	3	4	5	6	7	8	avg_{tvU}	std_{tvU}
2	-	-	-	-	-	-	-		12.22	6.66
3	3.48	-	-	-	-	-	-	-	6.36	83.75
4	3.62	3.46	-	-	-	-	-	-	4.50	80.03
5	3.64	3.63	3.07	-	-	-	-	-	3.76	77.73
6	3.65	3.64	3.65	1.95	-	-	-	-	3.46	88.34
7	3.74	3.73	3.73	3.69	3.46	-	-	-	3.64	81.39
8	3.78	3.77	3.78	3.78	3.71	3.72	-	-	3.68	75.93
9	3.95	3.94	3.94	3.94	3.93	3.79	3.71	-	3.54	77.26
10	3.90	3.90	3.90	3.90	3.87	3.81	3.70	3.24	3.77	74.71

Table 2.8 – Effective bandwidth (in GB/s) of *tvUnfold* (Algorithm 2.2), for $k \in \{1, \ldots, d-2\}$ for each d. The two columns on the right are the average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) for $k \in \{0, 1, \ldots, d-1\}$. Tensor sizes n are such that at least several GBs of memory are required. All experiments use MKL.

	Average performance			Sample stddev. (%).		
d	b_{L1}	b_{L2}	b_{L3}	$\mathrm{std}_{b_{\mathrm{L1}}}$	$\mathrm{std}_{b_{\mathrm{L2}}}$	$\mathrm{std}_{b_{\mathrm{L3}}}$
2	9.25	9.92	**13.94**	11.54	2.57	**2.48**
3	8.67	**11.81**	10.29	46.28	11.79	**6.80**
4	6.42	**11.30**	11.12	56.47	15.07	**12.79**
5	5.40	9.62	**10.61**	47.43	29.69	**10.82**
6	3.79	7.71	**11.47**	56.02	27.53	**10.78**
7	3.04	6.58	**8.97**	51.98	**41.09**	45.89
8	3.27	4.67	**8.11**	48.10	49.31	**43.83**
9	3.47	4.99	**7.30**	44.63	46.40	**35.82**
10	3.67	-	**7.77**	42.16	-	**33.79**

Table 2.9 – Average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) of the $\rho_\pi\rho_\pi$-block algorithm with *tvLooped*, for $k \in \{0, 1, \ldots, d-1\}$ for each d. Blocked tensor sizes n are such that at least several GBs of memory are required, while block sizes defined in Table 2.2 hitting different L1, L2 and L3 cache.

tive bandwidths are often $3x$ slower than those achieved by *tvLooped* and performance is highly unpredictable given standard deviations of up to 88% of average performance; unfold-based *TVM* implementations should be avoided.

2.3.4.4 Block tensor–vector multiplication algorithms.

Here we benchmark the two blocked tensor layouts proposed in Section 2.3.3: $\rho_\pi\rho_\pi$ and $\rho_Z\rho_\pi$. We expect both block algorithms to improve mode-obliviousness over *tvLooped* and *tvUnfold*, and, for sufficiently small block sizes, expect the $\rho_Z\rho_\pi$-algorithm to cache-obliviously improve reuse of input vector and output tensor elements. Conforming to earlier experiments, we only consider the *tvLooped* (Algorithm 2.1) for performing the *TVM* on a single block by fixing the *tv* parameter to the block *TVM* (Algorithm 2.3).

Tables 2.9 and 2.10 contain the experimental results of $\rho_\pi\rho_\pi$-block and $\rho_Z\rho_\pi$-block algorithm, respectively. The $\rho_Z\rho_\pi$-block algorithm achieves a mode-oblivious behavior similar to that of *tvLooped*, while both performance and mode-obliviousness of the $\rho_\pi\rho_\pi$-block algorithm drop as d increases. Compared to the $\rho_Z\rho_\pi$-block algorithm, performance losses are up to 67% while standard deviations are multiplied several times. This attests that the natural order blocking alone is not enough to induce mode-oblivious behavior; the Morton order based blocking is necessary.

The results in Table 2.9 show that $\rho_\pi\rho_\pi$-block *TVM* has much lower performance and much higher standard deviation between modes than *tvLooped* in Table 2.7, for $d \geq 7$. This confirms that the mode-dependent behavior is inherent to ρ_π layout and is amplified by the $\rho_\pi\rho_\pi$ layout.

In line with experiments from Section 2.3.4.2 (single-block tensor-vector multiplication), both block algorithms generally achieve the highest performance for L3-sized blocks, and if not, in all but two cases achieve it on L2-sized blocks instead. In terms of mode-obliviousness, the $\rho_Z\rho_\pi$-block algorithm performs best using L2- or L3-sized blocks.

	Average performance			Sample stddev. (%).		
d	b_{L1}	b_{L2}	b_{L3}	$\mathrm{std}_{b_{\mathrm{L1}}}$	$\mathrm{std}_{b_{\mathrm{L2}}}$	$\mathrm{std}_{b_{\mathrm{L3}}}$
2	10.43	9.93	**13.91**	5.14	3.21	**2.75**
3	**12.30**	11.67	10.35	6.58	11.99	**6.32**
4	11.73	**12.13**	11.31	**6.19**	7.71	10.23
5	10.89	**11.71**	11.17	**4.14**	6.01	10.06
6	10.03	10.88	**10.99**	10.32	**4.69**	15.08
7	8.70	10.74	**11.03**	13.10	**4.62**	8.40
8	9.10	10.13	**10.87**	11.13	7.96	**5.85**
9	9.25	10.21	**10.34**	8.88	**7.39**	9.44
10	9.55	-	**10.56**	**8.22**	-	9.17

Table 2.10 – Average effective bandwidth (in GB/s) and relative standard deviation (in percentage versus the average bandwidth) of the $\rho_Z\rho_\pi$-block algorithm with *tvLooped*, for $k \in \{0, 1, \ldots, d-1\}$ for each d. Blocked tensor sizes n are such that at least several GBs of memory are required, while block sizes defined in Table 2.2 hitting different L1, L2 and L3 cache.

Blocking for L2 incurs a small performance penalty, however, so blocking for L3 is preferred. The $\rho_Z\rho_\pi$-block *TVM* maintains high performance and low standard deviations across all values of d tested; the increase in cache efficiency on input and output elements that the Morton order induces proves crucial to blocked tensor layouts. This algorithm performs slower than $\rho_\pi\rho_\pi$ only for $d = 2$ and 6, and only slightly so.

We measure the highest performance for order-2 tensors at almost 14 GB/s for L3-sized blocks, for both block algorithms. This is higher than the raw *vm* and *mv* performance from Section 2.3.4.3 and 1.5 GB/s higher than the unblocked *tvLooped*, showing the benefit of blocked data layouts in general for regular matrices.

2.3.4.5 Experimental results of *TVM* algorithms for large-sized tensors

This section compares all presented *TVM* algorithms with the code generated by The Tensor Algebra Compiler (taco). Tables 2.11 and 2.12 compare the average bandwidths and standard deviations of the state-of-the-art algorithms, the taco-generated *TVM* kernels, the block algorithms $\rho_\pi\rho_\pi$, $\rho_Z\rho_\pi$ and $\rho_Z\rho_\pi^\star$, where the last one uses a hand-tuned blocking parameter. For the blocked layouts, we select the block sizes b_{L3} in Table 2.2 which correspond to $0.5z_{\mathrm{L3}}$ for reasons discussed in the previous subsection. For the $\rho_Z\rho_\pi^\star$ block algorithm, we used a block size such that the *TVM* of a block fits a memory of size $0.1z_{\mathrm{L3}}$ which not only performs better, but also is more suitable for any future threaded use of the Morton block layout.

While taco enables very generic generation of possibly quite complex tensor computation codes and its performance improves on the *tvUnfold* except for $d = 2$, it lags behind all other variants except once for $\rho_\pi\rho_\pi$ at $d = 7$. The lack of performance is explained by taco not reverting to optimized BLAS2 kernels in its generated codes. It also does not generate mode-oblivious code until $d > 7$, curiously reaching parity with the $\rho_Z\rho_\pi$-blocked results for $d = 9$ and 10. Between the two blocked variants, the $\rho_Z\rho_\pi$-block

d	*tvUnfold*	*tvLooped*	taco	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block	$\rho_Z\rho_\pi^\star$-block
2	12.22	12.22	9.36	13.94	13.91	**14.1**
3	6.36	**12.47**	11.92	10.29	10.35	11.06
4	4.50	10.79	10.09	11.12	11.31	**11.86**
5	3.76	10.71	10.69	10.61	11.17	**12.06**
6	3.46	10.98	9.93	11.47	10.99	**11.48**
7	3.64	11.26	9.55	8.97	11.03	**11.52**
8	3.68	**11.13**	6.94	8.11	10.87	10.87
9	3.54	**10.82**	6.75	7.30	10.34	10.36
10	3.77	10.26	7.05	7.77	10.56	**10.62**

Table 2.11 – Average effective bandwidth (in GB/s) of different algorithms for large order-d tensors. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

d	*tvUnfold*	*tvLooped*	taco	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block	$\rho_Z\rho_\pi^\star$-block
2	6.66	6.66	18.50	2.48	2.75	**0.65**
3	83.75	20.24	38.25	6.80	**6.32**	12.99
4	80.03	6.27	38.18	12.79	**10.23**	10.31
5	77.73	13.49	33.65	10.82	10.06	**7.08**
6	88.34	11.07	30.30	10.78	15.08	**8.58**
7	81.39	10.07	28.50	45.89	8.40	**4.73**
8	75.93	12.29	11.53	43.83	5.85	**5.82**
9	77.26	14.98	10.21	35.82	9.44	**9.44**
10	74.71	18.56	11.03	33.79	9.17	**9.17**

Table 2.12 – Relative standard deviation (in percentage versus the average bandwidth) of different algorithms for large order-d tensors. The lowest standard deviation, signifying the best mode-oblivious behavior, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

generally performs better in both performance and mode-obliviousness, while the $\rho_\pi\rho_\pi$ blocked layout additionally incurring unusably high standard deviations for $d > 6$. Both block algorithms dominate *tvUnfold* and taco in terms of performance (except for $d = 7$) and mode-obliviousness. The $\rho_Z\rho_\pi^\star$-block algorithm dominates all other variants except *tvLooped* for $d = 8, 9$, where they perform equally. Considering mode-obliviousness, the block algorithms following the $\rho_Z\rho_\pi$ and $\rho_Z\rho_\pi^\star$ tensor layouts achieve the best results by very comfortable margins.

2.3.4.6 Practical effects in an algorithm

Here we study the proposed tensor layouts in the context of the iterative higher-order power method (HOPM) [17, 18]. Given a square d-dimensional tensor and d initial vectors the HOPM proceeds as in Algorithm 2.6. Each of the $d(d-1)$ *TVM* calls per iteration can be computed using *tvLooped*; this requires a buffer space of n^{d-1} and yields a straightforward baseline implementation. The number of floating point operations is $d(3n + \sum_{i=2}^{d} 2n^i)$ per iteration for both the normalization and the *TVMs*. The number of data elements touched per iteration is d^2n for all vectors, plus $d(n^d + \sum_{i=2}^{d-1} 2n^i)$ for streaming the input tensor and repeatedly streaming intermediate tensors, plus $2dn$ for the normalization step. Hence the arithmetic intensity is

$$\frac{d\left(3n + \sum_{i=2}^{d} 2n^i\right)}{wd\left(2n + dn + n^d + \sum_{i=2}^{d-1} 2n^i\right)} \text{ flop per byte}, \tag{2.9}$$

with w being the number of bytes required to store a single element. Like for the arithmetic intensity of the *TVM* 2.3, this is bounded from above by $2/w$; the HOPM remains a bandwidth-bound operation.

When assuming a block layout, however, the loops on lines 4–7 can be implemented as a single kernel: the *tensor times a sequence of vectors*, or *ttsv*. Just as the block *TVM* algorithm calls *tvLooped* on each block, our blocked layouts allow making $d-1$ *tvLooped* calls on that single block. Compared to non-blocked layouts, we expect significant gains due to computing each of the d batches of $d-1$ *TVMs* entirely in cache, while for Morton-ordered blocks we additionally expect to observe an accumulated gain from increased cache efficiency on the input and output vectors. The blocked *ttsv* requires a buffer with size bounded by b^d, which coincides perfectly with our choice for $\alpha = 0.5$ from the preceding *TVM* analyses. The output vectors must be reset to zero before each of the d blocked *ttsv* calls, causing a $\Theta(dn)$ overhead per iteration in both time and data movement. However, this is negligible compared to the $d\sum_{i=2}^{d-1} 2n^i$ intermediate tensor elements it saves from being streamed from main memory.

Tables 2.13 and 2.14 show experimental results of a single iteration of HOPM on two different compute nodes, Ivy Bridge and Haswell, respectively. We consider only *tvLooped* and the proposed block algorithms, as Table 2.11 clearly indicated that these algorithms have better performance than others. We choose the input tensor size n to yield close to 8 GB sized data. For the block algorithms, where n is a multiple of b, we choose the latter to result in block sizes that fit L3 cache. Since HOPM computation involves all tensor

Algorithm 2.6 A basic higher-order power method

Input: A square tensor $\mathcal{A}$ of order d and size n,
d vectors $u^{(k)}$ of size n, the maximum number of iterations *maxIters*
Output: d vectors $u^{(k)}$ of size n

1: **for** *iters* = *0* to *maxIters* − 1 **do**
2: **for** $k = 0$ to $d - 1$ **do**
3: $\tilde{u}^{(k)} \leftarrow \mathcal{A}$
4: **for** $t = 0$ to $k - 1$ **do**
5: $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \times_t u^{(t)}$
6: **for** $t = k + 1$ to $d - 1$ **do**
7: $\tilde{u}^{(k)} \leftarrow \tilde{u}^{(k)} \times_t u^{(t)}$
8: $u^{(k)} \leftarrow \frac{\tilde{u}^{(k)}}{\|\tilde{u}^{(k)}\|}$
return $(u^{(0)}, u^{(1)}, \ldots, u^{(d-1)})$

modes, we do not evaluate mode-obliviousness in this case study. We always compute the speed in GB/s according to the data movement equation (2.9).

On the Ivy Bridge, the performance of *tvLooped* drops below 11 GB/s for d larger than 3, while the proposed block algorithms generally remain steady between 11–13 GB/s. They improve over *tvLooped* by 14.1 to 30.4 per cent. Haswell has 26.2 per cent less bandwidth per core, as measured by the STREAM variants of Section 2.3.4.2. Nonetheless, we achieve speedups of up to 15.4 per cent using block algorithms over *tvLooped*. As expected, the blocked variants' performances in HOPM (Table 2.13) are better than those reported in Table 2.11—while their improvement over the RAM memory speed from Table 2.3 can only be due to improved cache reuse. Since the ratio of input tensor elements versus those of the vectors u_i is high and increases with d, the benefits of cache reuse on vectors for Morton ordered curves decline and even out to memory copy speed. Finally, we give an example of an absolute run time. An iteration of HOPM of an order-5 tensor of 8 GB takes 4.5 seconds with *tvLooped* and 3.5 seconds with $\rho_Z \rho_\pi$-block on the Ivy Bridge. A straightforward implementation with nested for-loops, where the inner kernel multiplies the tensor with all input vectors simultaneously and accumulates into the output vector without using BLAS (as in $\hat{u}_i^{(0)} \leftarrow \hat{u}_i^{(0)} + \mathcal{A}_{i,j,k} u_j^{(1)} u_k^{(2)}$ for an order-3 tensor $\mathcal{A}$), takes 7 seconds; resulting in 6.29 GB/s, twice slower than the proposed $\rho_Z \rho_\pi$-block.

2.4 Shared-memory parallel tensor–vector multiplication

In this section, we build on the insights we have gained when developing the sequential *TVM* to develop shared-memory parallel algorithms for *TVM* considering both parallelizations of for-loops as well as parallelizations following the Single Program, Multiple Data (SPMD) paradigm.

A distribution of an order-d tensor of size $n_0 \times \cdots \times n_{d-1}$ over p threads is given by

d	*tvLooped*	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block
2	11.10	13.96	**13.98**
3	**13.99**	9.85	9.80
4	9.64	**11.32**	11.29
5	9.83	12.80	**12.82**
6	10.88	**12.65**	12.63
7	10.90	12.47	**12.50**
8	10.82	12.34	**12.35**
9	10.30	11.74	**11.76**
10	9.69	11.42	**11.46**

Table 2.13 – Average effective bandwidth (in GB/s) of different algorithms for HOPM of large order-d tensors on an Intel Ivy Bridge node. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

d	*tvLooped*	$\rho_\pi\rho_\pi$-block	$\rho_Z\rho_\pi$-block
2	10.22	**10.89**	10.73
3	**12.16**	8.59	8.55
4	8.47	8.86	**8.87**
5	8.65	8.77	**8.79**
6	8.97	9.23	**9.25**
7	8.58	**9.40**	**9.40**
8	8.54	**9.40**	**9.40**
9	8.04	**9.28**	**9.28**
10	7.88	**8.89**	**8.89**

Table 2.14 – Average effective bandwidth (in GB/s) of different algorithms for HOPM of large order-d tensors on an Intel Haswell node with two Intel Xeon E5-2690 v3 processors. Each processor has 12 cores sharing 30 MB of L3 cache and the upper bound on bandwidth per core of 14.5 GB/s. The highest bandwidth, signifying the best performance, for each d is shown in **bold**. Tensor sizes n are such that at least several GBs of memory are required.

a map $\pi : I \to \{0, \ldots, p-1\}$. Let π_{1D} be a regular 1D *block distribution* such that

$$\pi_{1D}(\mathcal{A}) : (i_0, i_1, \ldots, i_{d-1}) \mapsto \lfloor i_0/b_{1D} \rfloor,$$

where *block size* $b_{1D} = \lceil n_0/p \rceil$ refers to the number of hyperslices. Let $m_s = |\pi_{1D}^{-1}(s)|$ count the number of elements local to thread s. We demand that a 1D distribution be *load-balanced*,

$$\max_{s \in P} m_s - \min_{s \in P} m_s \leq n/n_0.$$

The choices to distribute over the first mode and to use a block distribution are without loss of generality; the dimensions of $\mathcal{A}$ and their fibers could be permuted to fit any other load-balanced 1D distribution. When $n_0 < p$ the dimensions could be reordered or fewer threads could be used.

We first describe a shared-memory *TVM* algorithm based on a for-loop parallelization in Section 2.4.1, as it serves as the baseline parallel *TVM* algorithm. Before describing the SPMD variants, we first motivate why it is sufficient to only consider one-dimensional partitionings of $\mathcal{A}$ in Section 2.4.2. We describe a number of SPMD *TVM* algorithms based on one-dimensional tensor partitionings in Section 2.4.3. Section 2.4.4 contains analyses of all presented parallel *TVM* algorithms, We present experimental results on up to 8-socket 120-core systems in Section 2.4.5.

2.4.1 The loopedBLAS baseline

The baseline algorithm uses a for-loop where each iteration can be processed concurrently without causing race conditions. Iterations of such a for-loop can either be scheduled *statically* or *dynamically*; the former cuts a loop of size n in exactly p parts and has each thread execute a unique part of the loop, while the latter typically employs a form of work stealing to assign parts of the loop to threads. In both cases, we assume that one does not explicitly control which thread will execute which part of the loop.

We assume $\mathcal{A}$ and $\mathcal{Y}$ have the default unfold layout. The *TVM* operation could naively be written using d nested for-loops, where the outermost loop that does not equal the mode k of the *TVM* is executed concurrently using OpenMP. Such code is generated by the tensor algebra compiler (taco). For a better performing parallel baseline, however, we observe that the $d-k$ inner for-loops correspond to a dense matrix–vector multiplication if $k < d$; we can thus write the parallel *TVM* as a loop over BLAS2 calls, and use highly optimized libraries for their execution. For $k = d-1$, the naively nested for-loops correspond to a dense matrix–transpose–vector multiplication, which is a standard BLAS2 call as well. Matrices involved with these BLAS2 calls generally are rectangular.

We execute the loop over the BLAS2 calls in parallel using OpenMP. For $k = d-1$, and for smaller tensors, this may not expose enough parallelism to make use of all available threads; we use any such left-over threads to parallelize the BLAS2 calls themselves, while taking care that threads collaborating on the same BLAS2 call are pinned close to each other to exploit shared caches as much as possible. Since all threads access both the input tensor and input vector, and since it cannot be predicted which thread accesses which part of the output tensor, all memory areas corresponding to $\mathcal{A}$, $\mathcal{Y}$, and $\mathbf{x}$ must

be interleaved. We refer to the described algorithm as *loopedBLAS*, which for $p = 1$ is equivalent to *tvLooped* in our earlier work [50].

2.4.2 Optimality of one-dimensional tensor partitioning

In this section, we prove that a one-dimensional partitioning of $\mathcal{A}$ yields an asymptotic lower bound on the number of elements which require a reduction when computing all possible *TVMs* of $\mathcal{A}$ in a series. Recall that the distribution π defines which thread multiplies an input tensor element $a_{\mathbf{i}}$ with its corresponding input vector element x_{i_k}, for any $\mathbf{i} \in I$. We say that the thread(s) in $\pi(i_0, \ldots, i_{k-1}, I_k, i_{k+1}, \ldots, i_{d-1})$ *contribute* to the reduction of $y_{\mathbf{j}}$ for $\mathbf{j} = (i_0, \ldots, i_{k-1}, 1, i_{k+1}, \ldots, i_{d-1})$ as they perform local reductions of multiplicands to $y_{\mathbf{j}}$ during the sequence of d TVMs.

Theorem 1. *Let $\mathcal{A}$ be a tensor of size $n = \prod_{k=0}^{d-1} n_k$, with $n_i \geq n_{i+1}$ for $0 \leq i < d-1$. Let $n_0 \geq p \geq 2$, $\prod_{i=1}^{d-1} n_i > d-1$, and $d > 2$. Let π_{1D} be a load-balanced one-dimensional distribution of $\mathcal{A}$ and $\mathcal{Y}$ such that thread s has at least $\lfloor n_0/p \rfloor n/n_0$ and at most $\lceil n_0/p \rceil n/n_0$ elements of $\mathcal{A}$. Then, this partitioning π_{1D} yields a lower bound on the number of elements which require a reduction during the series of computations $\mathcal{Y}_k = \mathcal{A} \times_k \boldsymbol{v}_k$ for all $k \in \{0, \ldots, d-1\}$.*

Proof. We will show that a partitioning over more dimensions violates the load balance assumption.

Let π be *any* load-balanced distribution of $\mathcal{A}$ and $\mathcal{Y}$ such that thread s has at least $2d\lfloor n_0/p \rfloor n/n_0$ and at most $2d\lceil n_0/p \rceil n/n_0$ work. For any $\mathbf{i} \in I$, let $J_{\mathbf{i}} = \{\mathbf{j} \in I \mid \vee_{k=0}^{d-1} i_k = j_k\}$ be the set of elements lying on d different axes which all go through $\mathbf{i}$, as illustrated in Figure 2.2 (left). Let $X_{\mathbf{i}} = \pi(J_{\mathbf{i}})$, where π is any distribution, describe the set of threads to which elements in $J_{\mathbf{i}}$ are mapped. Should $|X_{\mathbf{i}}| > 1$ for all $\mathbf{i} \in I$, then there is at least one *TVM* for which all elements of $\mathcal{Y}$ are involved in a reduction, as at least two threads contribute to $y_{\mathbf{j}}$. Therefore, the number of reductions for a series of *TVMs* is at least n/n_0. This lower bound on the number of reductions occurs for a 1D distribution over mode 0, showing that it is attainable. We will now consider if we can improve this lower bound by allowing $\mathbf{i}$ for which $|X_{\mathbf{i}}| = 1$, and if so, by how much.

Suppose there exist $r = \prod_{k=0}^{d-1} r_k$ coordinates $\mathbf{i} \in I$ such that $X_{\mathbf{i}} = \{s\}$, which form a hyper-rectangular subtensor $\mathcal{B}$ of side length $r_k < n_k$ contained in $\mathcal{A}$, as in Figure 2.2 (right). We choose a hyper-rectangular shape so that the r elements create the minimum amount of redundant work. Since $|X_{\mathbf{i}}| = 1$, the number of coordinates which must then also lie on thread s is $r(\sum_{k=0}^{d-1} \lceil n_k/r_k \rceil - (d-1))$. If $r_k = 2^{1/(d-1)} n_k / p^{1/(d-1)}$, this already corresponds to a load exceeding the assumed load balance $(2n - n/n_0)/p$; see Section 2.4.2.1 for details behind the constant factor $2^{1/(d-1)}$. Furthermore, with $r = 2^{d/(d-1)} n / p^{d/(d-1)}$ such coordinates, the lower bound on the number of reductions may only be reduced to $n/n_0(1 - 2/p)$, where $r/r_1 = 2n/pn_0$ is the projection of the cube r onto the $d-1$-dimensional output tensor. We consider only the data movement on the output tensor as the data movement on the input vector is at most $(d-1)n_0$, which is asymptotically less than the data movement associated with the output tensor since $d - 1 < \prod_{i=1}^{d-1} n_i$ and $d > 2$. □

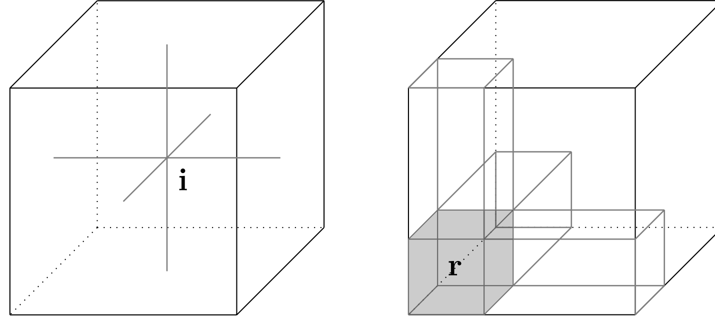

Figure 2.2 – Illustrations of elements in $J_{\mathbf{i}}$, indicated via thick gray lines, for an arbitrarily chosen $\mathbf{i}$ depicted by a filled dot (left), and for a cube of r elements $\mathbf{i}$ (right).

As seen in the proof, we do not assume a specific reduction algorithm and count the minimal work involved.

2.4.2.1 Finding the minimal number of elements r

Let $r = \prod_{k=0}^{d-1} r_k$ be a hyperrectangular subtensor $\mathcal{B}$ contained in $\mathcal{A}$. The number of elements e in $\mathcal{A}\backslash\mathcal{B}$ that lie orthogonal to any of the d dimensions is $(\sum_{k=0}^{d-1} rn_k/r_k) - r(d-1)$, which simplifies to $r(\sum_{k=0}^{d-1} [n_k/r_k] - (d-1))$. Assuming $p \leq n_0$ (which is the largest dimension in $\mathcal{A}$), we know that $\lceil n_0/p \rceil n/n_0 < 2n/p$:

$$\lceil n_0/p \rceil n/n_0 \leq (n_0 n + pn - n)/pn_0 \leq (2n - n/n_0)/p < 2n/p$$

We are looking for a minimal r for which

$$e = r(\sum_{k=0}^{d-1} n_k/r_k - (d-1)) > 2n/p$$

for $r, n, d, p \in \mathbb{N}$ and under the assumptions that $p > 1$, $n \geq p$, $d > 2$, and $r_k < n_k$.

A reasonable guess would be $r_k = cn_k/p^{1/(d-1)}$, where $p^{1/(d-1)} > c > 0$ is some to-be-determined constant (thus the total size $r = c^d n/p^{d/(d-1)}$, and $n_k/r_k = p^{1/(d-1)}/c$):

$$\begin{aligned} c^d n/p^{d/(d-1)}(dp^{1/(d-1)}/c - (d-1)) &> 2n/p \\ c^{(d-1)} n(d - 2/c^{(d-1)})/p &> c^d n(d-1)/p^{d/(d-1)}, \end{aligned}$$

which we can further simplify to

$$c^{(d-1)} n(d - 2/c^{(d-1)})/p \geq c^d n(d-1)/p^{d/(d-1)}$$

which is satisfied when $c^{(d-1)}/p \geq c^d/p^{d/(d-1)}$ and $-2/c^{(d-1)} \geq -1$. From the second equation, $c^{d-1} \geq 2$ and thus $c \geq 2^{1/(d-1)}$. From the first equation, $p \geq c^{(d-1)}$ and thus $p \geq 2$. The total number of elements r we can take hence must be smaller than $2^{d/(d-1)} n/p^{d/(d-1)}$.

2.4.3 Proposed 1D *TVM* algorithms

We explore a family of shared-memory parallel *TVM* algorithms assuming the π_{1D} distribution of the input and output tensors, as Theorem 1 enables us to focus only on such partitionings for the *TVMs* when our goal is to minimize the data movement. Such partitioning results in p disjoint input tensors $\mathcal{A}_s$ and p disjoint output tensors $\mathcal{Y}_s$, whose unions correspond to $\mathcal{A}$ and $\mathcal{Y}$, respectively. For all but $k = 0$, a parallel *TVM* amounts to a thread-local call to a sequential *TVM* computing $\mathcal{Y}_s = \mathcal{A}_s \times_k \mathbf{x}$; each thread reads from its own part of $\mathcal{A}$ while writing to its own part of $\mathcal{Y}$. We may thus employ the $\rho_Z\rho_\pi$ layout for $\mathcal{A}_s$ and $\mathcal{Y}_s$ and use its high-performance sequential mode-oblivious kernel [50]; here, $\mathbf{x}$ is allocated interleaved while $\mathcal{A}_s$ and $\mathcal{Y}_s$ are explicit. The global tensors $\mathcal{A}$ and $\mathcal{Y}$ are never materialized in shared-memory—only their distributed variants are required. We expect the explicit allocation of these two largest data entities involved with the *TVM* computation to induce much better parallel efficiency compared to the *loopedBLAS* baseline where all data is interleaved.

For $k = 0$, the output tensor $\mathcal{Y}$ cannot be distributed. We define that $\mathcal{Y}$ is then instead subject to a 1D block distribution over mode 1, and assume $n_1 \geq p$. Since the distributions of $\mathcal{A}$ and $\mathcal{Y}$ then do not match, communication ensues. We suggest three variants that minimize data movement, characterized by the number of synchronization barriers they require: zero, one, or $p - 1$.

2.4.3.1 The 0-sync algorithm

We avoid performing a reduction on $\mathcal{Y}$ for $k = 0$ by storing $\mathcal{A}$ twice; once with a 1D distribution over mode 0, another time using a 1D distribution over mode $d-1$. Although the storage requirement is doubled, data movement remains minimal while explicit reduction for $k = 0$ is completely eliminated: the copy with the 1D distribution over mode $d - 1$ can then be used instead. In either case, the parallel *TVM* computation completes after a sequential thread-local *TVM*; this variant requires no barriers to resolve data dependencies.

If this algorithm is to re-use output of mode-0 or mode-$d-1$ *TVM* to perform another *TVM*, each thread must replicate the work such that the unused input tensor storage stores the output tensor partitioned over mode 0 or mode $d - 1$, respectively.

2.4.3.2 The 1-sync algorithm

This variant performs an explicit reduction of the $\mathcal{Y}_s$ for $k = 0$ using two-phase reduction and behaves as the 0-sync variant otherwise. It requires a larger buffer for the $\mathcal{Y}_s$ to cope with $k = 0$, since each thread computes a full output tensor $\mathcal{A}_s \times_0 \mathbf{x}$ that contains partial results only. The output tensor is then reduced by all p threads, such that each thread contains its part according to a π_{1D} distribution over mode 0, i.e., $\mathcal{Y}_t = (\sum_{s=0}^{p-1}(\mathcal{A}_s \times_0 \mathbf{x}))_t$, for all $t \in P$. By load balance, each thread has at most $\lceil n_0/p \rceil n/(n_0 n_0)$ elements of the output tensor. A barrier must separate the local *TVM* from the reduction phase to ensure no incomplete $\mathcal{Y}_s$ are reduced.

2.4.3.3 The q-sync algorithm

This variant stores $\mathcal{A}$ with a 1D distribution over mode 0. It also stores two versions of the output tensor, one interleaved $\mathcal{Y}$ and one thread-local $\mathcal{Y}_s$. The vector $\mathbf{x}$ is interleaved. Both $\mathcal{A}_s$ and $\mathcal{Y}_s$ are split into $q = \prod_{i=1}^{d-1} q_i \geq p$ parts, by splitting each object into q_i parts across mode i. We index the resulting objects as $\mathcal{A}_{s,t}$, which are explicitly allocated to thread s, and $\mathcal{Y}_{s,t}$, which are allocated twice, once explicitly and once interleaved. The input vector $\mathbf{x}$ remains interleaved. Algorithm 2.7 summarizes this approach.

Algorithm 2.7 The q-sync parallel *TVM* algorithm at thread s.

Input: A square tensor $\mathcal{A}$ of order d containing n elements and a vector $\mathbf{x}$ of size n_k.
Output: An output tensor $\mathcal{Y}$ of size n/n_k.

```
1: if k = 0 then
2:     Y = A_{s,s} ×_k x
3:     for t = 1 to q − 1 do
4:         barrier
5:         Y += A_{s,(t+s) mod q} ×_k x
6: else
7:     for t = 0 to q − 1 do
8:         Y_{s,t} += A_{s,t} ×_k x
   return Y
```

If this algorithm is to re-use output of mode-0 *TVM*, then, similarly to the 0-sync variant, each thread must re-synchronize its local $\mathcal{Y}_{s,t}$ with $\mathcal{Y}$. Thus, unless the need explicitly arises, implementations need not distribute $\mathcal{Y}$ over n_1 as part of a mode-0 *TVM* (at the cost of interleaved data movement on $\mathcal{Y}$). This algorithm avoids doubling the storage requirement yet still eliminates explicit reductions for $k = 0$, replacing reduction with synchronization. For $k > 0$, no barriers are required since each thread writes into disjoint areas of $\mathcal{Y}$ due to the 1D block distribution over mode 0. For simplicity, we omit the indexing of the interleaved $\mathcal{Y}$ into one of its q subtensors. Additionally, lines 7 and 8 assume the amount of subtensors in $\mathcal{Y}_s$ is the same as in $\mathcal{A}_s$, i.e., $q_k = 1$. Otherwise, the code has to be adapted to go over q/q_k output subtensors q_k times.

2.4.3.4 Interleaved $q(i)$-sync algorithm

The $q(i)$-sync variant assumes only an interleaved $\mathcal{Y}$ in lieu of thread-local $\mathcal{Y}_s$. Each $\mathcal{A}_s$ is further split into q parts, each stored thread-locally, giving rise to the input tensors $\mathcal{A}_{s,t}$; this split is equal across all threads and is used to synchronise writing into the output tensor. Each thread s executes Algorithm 2.8 to compute k-mode *TVM* using the interleaved $q(i)$ sync approach. For simplicity, we omit any offset into $\mathcal{Y}$.

Algorithm 2.8 The interleaved $q(i)$-sync parallel *TVM* algorithm at thread s.

Input: A square tensor $\mathcal{A}$ of order d containing n elements and a vector $\mathbf{x}$ of size n_k.
Output: An output tensor $\mathcal{Y}$ of size n/n_k.

1: $\mathcal{Y} = \mathcal{A}_{s,s} \times_k \mathbf{x}$
2: **for** $t = 1$ **to** $q - 1$ **do**
3:     **if** $k = 0$ **then**
4:         **barrier**
5:     $\mathcal{Y} \mathrel{+}= \mathcal{A}_{s,(t+s) \bmod q} \times_k \mathbf{x}$
return $\mathcal{Y}$

2.4.3.5 Explicit $q(e)$-sync algorithm

The explicit q-sync variant allocates all $\mathcal{A}_s$ and $\mathcal{Y}_s$ explicitly local to thread s and keeps $\mathbf{x}$ interleaved. We split each $\mathcal{A}_s$ into q subtensors $\mathcal{A}_{s,t}$. Each thread s executes Algorithm 2.9 in order to compute the explicit $q(e)$-sync parallel *TVM*. An inter-socket data movement occurs on the output tensor (Line 5 of Algorithm 2.9). To cope with cases where the output tensors act as input on any subsequent *TVM* calls on all possible modes, $\mathcal{A}_s$ must be split into $q \geq p$ parts *across each dimension*, while $\mathcal{Y}_s$ should likewise be split into at least p^{d-1} parts. An implementation does not need to allocate p^d separate subtensors, but rather use a blocked tensor layout in a recursive fashion for each of the p subtensors.

Algorithm 2.9 The explicit $q(e)$-sync parallel *TVM* algorithm at thread s.

Input: A square tensor $\mathcal{A}$ of order d containing n elements and a vector $\mathbf{x}$ of size n_k.
Output: An output tensor $\mathcal{Y}$ of size n/n_k.

1: **if** $k = 0$ **then**
2:     $\mathcal{Y}_{s,s} = A_{s,s} \times_k \mathbf{x}$
3:     **for** $t = 1$ **to** $q - 1$ **do**
4:         **barrier**
5:         $\mathcal{Y}_{(t+s) \bmod q,t} \mathrel{+}= A_{s,t} \times_k \mathbf{x}$
6: **else**
7:     **for** $t = 0$ **to** $q - 1$ **do**
8:         $\mathcal{Y}_{s,t} = A_{s,t} \times_k \mathbf{x}$
return $\mathcal{Y}$

2.4.4 Analysis of the algorithms

We analyse the parallel *TVM* algorithms from the previous section, restricting ourselves not only to the amount of data moved during a *TVM* computation, but also consider mode-obliviousness, memory, and work. For quantifying data movement we assume perfect caching, meaning that all required data elements are touched exactly once.

Since barriers require active participation by processor cores while they also make use of communication, the time in which a given algorithm completes a *TVM* computation is given by (1.1). This is determined by the quantified costs of a thread s at phase i, $W_{k,i}r$, $U_{k,i}g$, and $V_{k,i}h$, where r is the thread compute speed in seconds per flop, g is the time to transfer a byte from local memory to a thread, and h is the time to transfer a byte from remote memory to a thread; L is the time for a barrier to complete in seconds (Section 1.1.2).

Strongly scaling algorithms have that $O(n,p)$ is independent of p (which is unrealistic), while weakly scalable algorithms have that the ratio $O(pn,p)/T_{\text{seq}}(pn)$ is constant; iso-efficiency instead tells us a much wider range of conditions under which the algorithm scales.

The *TVM* computation is a heavily bandwidth-bound operation. It performs $2n$ flops on $n+n/n_k+n_k$ data elements, and thus has arithmetic intensity equal to $1 < \frac{2n}{n+n/n_k+n_k} <$ 2 flop per element. This amounts to a heavily bandwidth-bound computation even when considering a *sequential TVM* [50]. The multi-threaded case is even more challenging, as cores on a single socket compete for the same local memory bandwidth. In our subsequent analyses we will thus ignore the computational part of the equations for T, O, and E. We also consider memory overhead and efficiency versus the sequential memory requirement $M_{\text{seq}} = n + \max_k (n/n_k + n_k)$ words.

2.4.4.1 The loopedBLAS algorithm

The *loopedBLAS* variant interleaves $\mathcal{A}$, $\mathcal{Y}$, and $\mathbf{x}$, storing them once while it performs $2n$ flops to complete the *TVM*; it is thus both memory- and work-optimal. It does not include any cache-oblivious nor mode-oblivious optimizations, and requires no barrier synchronizations. Since all memory used is interleaved we assume their effective bandwidth is spread over g and h proportional to the number of CPU sockets p_{s}. Assuming a uniform work balance is achieved at run time, all p threads read n/p data from $\mathcal{A}$, write $n/(pn_k)$ to $\mathcal{Y}$, and read the full n_k elements of $\mathbf{x}$. Thus, $U_{s,0} = \frac{1}{p_{\text{s}}}v$ and $V_{s,0} = \frac{p_{\text{s}}-1}{p_{\text{s}}}v$ with $v = \left(\frac{n+n/n_k}{p} + n_k\right)$. The parallel overhead becomes

$$O(n,p) = \frac{p_{\text{s}}-1}{p_{\text{s}}}(n+n/n_k)(h-g) + n_k\left((p_t-1)g + p_t(p_s-1)h\right). \tag{2.10}$$

We see the overhead is dominated by $\mathcal{O}(n(h-g))$ as p_{s} increases, while for $p_{\text{s}} = 1$ the overhead simplifies to $\Theta(p_t n_k g)$. This excludes any underlying overhead of the parallel implementation of BLAS.

2.4.4.2 The 0-sync algorithm

This algorithm incurs n words of extra storage and thus is not memory optimal. In both cases of $k = 0$ and $k > 0$ the amount of work executed remains optimal at $2n$ flops. The cache- and mode-oblivious optimizations from our earlier work are fully exploited by this algorithm, while, like *loopedBLAS*, S remains zero. Here, $\mathcal{A}$ and $\mathcal{Y}$ are allocated explicitly

while $\mathbf{x}$ remains interleaved; hence $U_{s,0} = \frac{1}{p}(n + n/n_k) + \frac{1}{p_\mathrm{s}} n_k$ and $V_{s,0} = \frac{p_\mathrm{s}-1}{p_\mathrm{s}} n_k$, resulting in

$$O(n,p) = (p_\mathrm{t} - 1)n_k g + p_\mathrm{t}(p_\mathrm{s} - 1)n_k h. \tag{2.11}$$

This overhead is bounded by $\Theta(pn_k h)$ for $p_\mathrm{s} > 1$, a significant improvement over *loopedBLAS*.

If the output tensor $\mathcal{Y}$ must assume a similar datastructure to $\mathcal{A}$ for further processing, the output must also be stored twice. The minimal cost for this incurs extra overhead at $\Theta((n/n_k)g)$; i.e., a thread-local data copy, which remains asymptotically smaller than $T_\mathrm{seq} = \mathcal{O}(ng)$. We do note that applications which require repeated *TVMs* such as the higher-order power method (HOPM) actually do *not* require this extra step; multilinearity can be exploited to perform the HOPM block-by-block (see Section 2.3.4.6).

2.4.4.3 The 1-sync algorithm

This variant requires the $\mathcal{Y}_s$ are all of size n/n_k instead of $(n/n_k)/p$ elements, which constitutes a memory overhead of $(p-1)(n/n_k)$. It benefits from the same cache- and mode-oblivious properties as the 0-sync variant, and achieves the same overhead (Eq. 2.11) when $k > 0$. For $k = 0$, however, we must account for the reduction phase on the $\mathcal{Y}_s$ and for the barrier that precedes it. Reduction proceeds with minimal cost by having each thread reduce $(n/n_k)/p$ elements corresponding to those elements it should locally store, resulting in an overhead $O(n,p)$ for a mode-0 *TVM* of

$$p_t(n/n_0)g + p_t(p_s - 1)(n/n_0)h + (1 - 1/p_s)n_0(h - g) + (p - 1)(L + (n/n_0)r)$$

This extra overhead is proportional to $(n/n_0)/p$ for both memory movement and flops.

2.4.4.4 The interleaved $q(i)$-sync algorithm

The interleaved $q(i)$-sync variant requires only the interleaved storage of $\mathcal{Y}$, which implies writing output requires inter-process data movement for all modes, a significant and equal overhead for all modes. Only accesses to $\mathcal{A}_{s,p}$ remain explicit. This variant remains both memory and work optimal, while it incurs $q - 1$ barriers and requires, for $k = 0$, the complete output tensor to be accessed by all p threads. Thus, it results in an overhead $O(n,p)$ for a mode-0 *TVM* of

$$(p_t - 1)(n/n_0)g + p_t(p_s - 1)(n/n_0)h + (1 - 1/p_s)n_0(h - g) + p(p - 1)L,$$

which increases with $p(p-1)L$. Hence, this parallel overhead is a significant increase over that of the 0-sync variant (Eq. 2.11) if $k = 0$, and equivalent otherwise. It still improves significantly over *loopedBLAS* (compare (2.10)). For $k > 0$, the overhead is

$$O(n,p) = (1 - 1/p_s)(n/n_k + n_k)(h - g).$$

2.4.4.5 The explicit $q(e)$-sync and q-sync algorithms

In the explicit variant, and the q-sync variant, accesses to $\mathcal{A}_{s,p}$ and $\mathcal{Y}_{s,p}$ are explicit, while accesses to vector are interleaved. This variant remains work- and memory-optimal but reduces the data movement overhead for all modes other than 0 to

$$O(n,p) = (p_t - 1)n_k g + p_t(p_s - 1)n_k h.$$

This explicit variant improves on the interleaved one in that inter-socket communication related to $\mathcal{Y}$ is now only incurred for $k = 0$. Compared to the 0-sync variant, the q-sync variants trade synchronization and inter-socket communication for reduced memory storage overheads.

2.4.4.6 Mode-obliviousness

The *loopedBLAS* algorithm is highly sensitive to the mode k in which a TVM is executed, while those algorithms based on the $\rho_Z\rho_\pi$ tensor layout are, by design, not sensitive to k [50]. The 0-sync and 1-sync variants exploit the $\rho_Z\rho_\pi$ maximally; the thread-local tensors use a single such layout, and each thread thus behaves fully mode-oblivious.

For the q-sync variants, however, each locally stored input tensor is split into subtensors. Suppose mode k has $\mathcal{A}_s$ split in q_k parts, resulting in $q = \prod_{i=1}^{d-1} q_i \geq p$ parts stored using a $\rho_Z\rho_\pi$ layout. This may hamper both cache efficiency and mode-obliviousness, since reading from $\mathbf{x}$ and writing to $\mathcal{Y}$ now only partially follows a Morton order. Hence we prefer to minimize q and $\max_i q_i$. This occurs when the q_i are the result of an integer factorization of p.

2.4.4.7 Iso-efficiencies

Table 2.15 summarizes the results from this section for arbitrary p_s, while Table 2.16 notes the simplified overheads when $p_\text{s} = 1$. Note that the parallel efficiency depends solely on the ratio $O(n,p)$ versus $T_\text{seq}(n)$. Considering the work- and communication-optimal 0-sync algorithm, efficiency thus is proportional to $\frac{pn_kh}{ng}$; i.e., 0-sync scales as long as p grows proportionally with n/n_k. For *loopedBLAS*, efficiency is proportional to $(h/g-1)(p_\text{s}-1)/p_\text{s}$. Since $h/g-1$ is constant, it drops as the number of sockets increases; that is, *loopedBLAS* does not scale in p_s.

For 1-sync, iso-efficiency is attained whenever $\frac{p}{n_k}(r + h + g/p_s) + \frac{p}{n}L$ is constant; i.e., p should grow linearly with n when computation is latency-bound, and linearly with n_k otherwise. All q-sync variants attain iso-efficiency when p grows linearly with n_k and p^2 grows linearly with n. Having to double n_k when p is doubled ($n_k \sim p$) to retain parallel efficiency is not very favourable as it incurs a type of curse of dimensionality: for hypersquare tensors, doubling of n^k means the total tensor size is multiplied by 2^d, which is unfortunate. Having n or n/n_k grow with p or p^2 would be preferred, and points to 0-sync as the preferred method.

Parallel *TVM* algorithm	Parallel overhead Work	Memory	Movement	Barrier	Mode-Oblivious	Allocation mode Implicit	Explicit	k
loopedBLAS	$\mathbf{0}$	$\mathbf{0}$	$n(h-g)$	$\mathbf{0}$	none	$\mathbf{x}, \mathcal{A}, \mathcal{Y}$	-	-
0-sync	$\mathbf{0}$	n	$\mathbf{pn_0h + p_tn_0g}$	$\mathbf{0}$	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{A}, \mathcal{Y}$	-
1-sync	pn/n_0r	pn/n_0	$pn/n_0h + p_tn/n_0g$	pL	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	-
$q(i)$-sync	$\mathbf{0}$	$\mathbf{0}$	$pn/n_0h + p_tn/n_0g$	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}$	0
$q(e)$-sync	$\mathbf{0}$	$\mathbf{0}$	$pn/n_0h + p_tn/n_0g$	p^2L	good	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	0
q-sync	$\mathbf{0}$	n/n_0	$pn/n_0h + p_tn/n_0g$	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}, \mathcal{Y}$	0

Table 2.15 – Summary of overheads for each parallel shared-memory *TVM* algorithm, plus the allocation mode of $\mathcal{A}, \mathcal{Y}$, and $\mathbf{x}$. We display the worst-case asymptotics; i.e., assuming $p_s > 1$ and the worst-case k for non mode-oblivious algorithms. Optimal overheads are in bold.

Parallel *TVM* algorithm	Parallel overhead Work	Memory	Movement	Barrier	Mode-Oblivious	Allocation mode Implicit	Explicit	k
loopedBLAS	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{pn_0g}$	$\mathbf{0}$	none	$\mathbf{x}, \mathcal{A}, \mathcal{Y}$	-	-
0-sync	$\mathbf{0}$	n	$\mathbf{pn_0g}$	$\mathbf{0}$	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{A}, \mathcal{Y}$	-
1-sync	pn/n_0r	pn/n_0	pn/n_0g	pL	**full**	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	-
$q(i)$-sync	$\mathbf{0}$	$\mathbf{0}$	pn/n_0g	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}$	0
$q(e)$-sync	$\mathbf{0}$	$\mathbf{0}$	pn/n_0g	p^2L	good	$\mathbf{x}$	$\mathcal{A}, \mathcal{Y}$	0
q-sync	$\mathbf{0}$	n/n_0	pn/n_0g	p^2L	good	$\mathbf{x}, \mathcal{Y}$	$\mathcal{A}, \mathcal{Y}$	0

Table 2.16 – Like Table 2.15, but assuming $p_s = 1$.

2.4.5 Experiments

We run our experiments on a number of different Intel Ivy Bridge nodes with different specifications summarized in Table 2.17. As we do not use hyperthreading, we limit the algorithms to use at most $p/2$ threads equal the number of cores (each core supports 2 hyperthreads). We measure the maximum bandwidth of the systems using several variants of the STREAM benchmark, reporting the maximum measured performance only.

The system uses CentOS 7 with Linux kernel 3.10.0 and software is compiled using GCC version 6.1. We use Intel MKL version 2018.2.199 for *loopedBLAS*. We also run with LIBXSMM version 1.9-864 for algorithms based on blocked layouts (0- and q-sync), and retain only the result with the library which runs faster of the two. To benchmark a kernel, we conduct 10 experiments for each combination of dimension, mode, and algorithm.

To illustrate and experiment with the various possible trade offs in parallel *TVM*, we implemented the baseline synchronization-optimal *loopedBLAS*, the work- and communication-optimal 0-sync variant, and the work-optimal q-sync variant. We investigate their performance and mode-obliviousness. We measure algorithmic performance using the formula for effective bandwidth (GB/s). We benchmark tensors of order-2 up to order-5. We choose n such that the combined input and output memory areas during a single *TVM* call have a combined size of at least 10 GBs; The exact array of tensor sizes and block sizes are given in Table 2.18, and Table 2.19. respectively. The block sizes

Node	CPU (clock speed)	p_s	p_t	p	Mem. size (clock speed)	Bandwidth STREAM	Bandwidth Theoretical
1	E5-2690 v2 (3 GHz)	2	20	40	256 GB (1600 MHz)	76.7 GB/s	95.37 GB/s
2	E7-4890 v2 (2.8 GHz)	4	30	120	512 GB (1333 MHz)	133.6 GB/s	158.91 GB/s
3	E7-8890 v2 (2.8 GHz)	8	30	240	2048 GB (1333 MHz)	441.9 GB/s	635.62 GB/s

Table 2.17 – An overview of machine configurations used. Memory runs in quad channel mode on nodes 1, 2, and 3, and in octa-channel on node 4. Each processor has 32 KB of L1 cache memory per core, 256 KB of L2 cache memory per core, and $1.25p_t$ MB of L3 cache memory shared amongst the cores.

d	Node 1	Node 2	Node 3
2	45600×45600 (15.49)	68400×68400 (34.86)	136800×136800 (139.43)
3	$1360 \times 1360 \times 1360$ (18.74)	$4080 \times 680 \times 4080$ (84.34)	$4080 \times 680 \times 4080$ (84.34)
4	$440 \times 110 \times 88 \times 440$ (13.96)	$1320 \times 110 \times 132 \times 720$ (102.81)	$1440 \times 110 \times 66 \times 1440$ (112.16)
5	$240 \times 60 \times 36 \times 24 \times 240$(22.25)	$720 \times 60 \times 36 \times 24 \times 360$(100.11)	$720 \times 50 \times 36 \times 20 \times 720$(139.05)

Table 2.18 – Table of tensor sizes $n_1 \times \cdots \times n_d$ per tensor-order d and the node as given in Table 2.17. The exact size in GBs is given in parentheses.

selected ensure that computing a *TVM* on such a block fits L3 cache. This combination of tensor and block sizes ensures all algorithms run with perfect load balance and without requiring any padding of blocks; to ensure this, we choose block sizes that correspond to 0.5–1 MB of our L3 cache; note that for our parallel *TVM* variants, the Morton order ensures the remainder cache remains obliviously well-used. We additionally kept the sizes of tensors equal through all pairs of (d, p_s), which enables comparison of different algorithms within the same d and p_s.

d	Node 1	Node 2	Node 3
2	570×570	570×570	570×570
3	$68 \times 68 \times 68$	$68 \times 68 \times 68$	$34 \times 68 \times 34$
4	$22 \times 22 \times 22 \times 22$	$22 \times 22 \times 22 \times 12$	$12 \times 22 \times 22 \times 12$
5	$12 \times 12 \times 12 \times 12 \times 12$	$12 \times 12 \times 12 \times 12 \times 6$	$6 \times 10 \times 12 \times 10 \times 6$

Table 2.19 – Table of block sizes $b_1 \times \cdots \times b_d$ per tensor-order d and the node as given in Table 2.17. Sizes are chosen such that all elements of a single block can be stored in L3 cache.

	Average performance			Sample stddev. (%).		
d	*loopedBLAS*	0-sync	q-sync	*loopedBLAS*	0-sync	q-sync
2	40.23	42.28	**42.54**	0.63	**0.55**	0.65
3	36.43	39.34	**39.87**	24.93	2.55	**2.50**
4	37.63	39.02	**39.05**	21.29	**4.35**	4.40
5	34.56	36.53	**36.65**	22.43	5.14	**4.26**

Table 2.20 – Average effective bandwidth (in GB/s) and relative standard deviation (in %, versus the average bandwidth) of algorithms over all modes running on a single processor (Node 1). The highest bandwidth and lowest standard deviation for different dimensions d are stated in **bold**.

2.4.5.1 Single-socket results

Table 2.20 shows the experimental results for the single-socket of Node 1. Note that it drops to half of the peak numbers measured in Table 2.17. Note that as there is no inter-socket communication, all memory regions are exceptionally allocated locally for intra-socket experiments.

As the *loopedBLAS* algorithm relies on the unfold storage, whose structure does require a loop over subtensors for modes 1 and d; thus, no for-loop parallelisation is possible for these modes and the algorithm employs the internal MKL parallelization. Thus, the Table shows its performance is highly mode-dependent, and that the algorithms based on blocked $\rho_Z\rho_\pi$ storage perform faster than the *loopedBLAS* algorithm. The block Morton order storage transfers the mode-obliviousness to parallel $TVMs$ (the standard deviation oscillates within 1%), as the Morton order induces mode-oblivious behavior on each core.

2.4.5.2 Inter-socket results

Table 2.21, 2.22, 2.23, and 2.24 show the parallel-TVM results on machines with different numbers of sockets for tensors of order-2, 3, 4, and 5, respectively. These experimental run times show a lack of scalability of *loopedBLAS*. This is due to the data structures being interleaved (Section 2.4.4.1) instead of making use of a 1D distribution. Interleaving or not only matters for multi-socket results, but since Table 2.20 conclusively shows that approaches based on our $\rho_Z\rho_\pi$ storage remain superior on single sockets, we may conclude our approach is superior at all scales.

The performance drops slightly when d increases for all variants. This is inherent to the BLAS libraries handling matrices with a lower row-to-column ratio better than tall-skinny or short-wide matrices [50]—and this ratio increases when processing higher-order tensors.

For first-mode $TVMs$, the 0-sync algorithm slightly outperforms the q-sync, while they achieve almost identical performance for all the other modes. The cause lies with the 0-sync not requiring any synchronization for $k = 0$, and the effect is the 0-sync achieves the lowest standard deviation. Our measured performances are within the impressive range of 75–88%, 81–95%, and 66–77% of theoretical peak performance for node 1, 2,

	Sample stddev. (%).			Average performance		
	Number of sockets (p_s)			Number of sockets (p_s)		
Algorithm	2	4	8	2	4	8
loopedBLAS	1.74	17.15	3.72	68.52	50.68	9.68
0-sync	**0.03**	**0.10**	**0.34**	**84.19**	**150.82**	**492.32**
q-sync	0.12	0.11	0.74	84.17	150.39	487.38

Table 2.21 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-2 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Sample stddev. (%).			Average performance		
	Number of sockets (p_s)			Number of sockets (p_s)		
Algorithm	2	4	8	2	4	8
loopedBLAS	9.57	16.52	23.05	63.89	55.68	13.66
0-sync	2.80	**1.38**	**3.42**	**77.06**	**145.07**	**467.31**
q-sync	**1.90**	3.86	6.56	76.27	143.17	441.65

Table 2.22 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-3 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Sample stddev. (%).			Average performance		
	Number of sockets (p_s)			Number of sockets (p_s)		
Algorithm	2	4	8	2	4	8
loopedBLAS	16.99	23.43	15.45	61.60	47.73	12.59
0-sync	**2.84**	**2.44**	**1.88**	**77.12**	**138.54**	**446.01**
q-sync	3.67	5.47	9.98	76.82	137.79	424.85

Table 2.23 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-4 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Sample stddev. (%).			Average performance		
	Number of sockets (p_s)			Number of sockets (p_s)		
Algorithm	2	4	8	2	4	8
loopedBLAS	15.37	19.70	32.03	56.11	54.04	12.43
0-sync	**3.47**	**5.01**	**5.02**	**71.71**	**129.80**	**421.98**
q-sync	4.17	9.37	14.83	71.65	129.60	397.25

Table 2.24 – Average effective bandwidth (in GB/s) and relative standard deviation (in % of average) over all possible $k \in \{1, \ldots, d\}$ of order-5 tensors of algorithms executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3). The highest bandwidth and lowest standard deviation for different d are stated in **bold**.

	Number of sockets (p_s)		
Algorithm	2	4	8
loopedBLAS	0.81	0.31	0.02
0-sync	0.99	0.93	0.98
q-sync	0.99	0.93	0.97

Table 2.25 – Parallel efficiency of algorithms on order-2 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket run time on a given node of q-sync algorithm on the same problem size and tensor order.

and 3, respectively.

Tables 2.25, 2.26, 2.27, and 2.28 display the parallel efficiency versus the performance of the q-sync on a single socket. Each node takes its own baseline since the tensor sizes differ between nodes as per Table 2.18; one can thus only compare parallel efficiencies over the *columns* of these tables, and cannot compare rows; we compare algorithms, and do not investigate inter-socket scalability.

The astute reader will note parallel efficiencies larger than one; these are commonly due to cache-effects, in this case likely output tensors that fit in the combined cache of eight CPUs, but did not fit in cache of a single CPU. These tests conclusively show that both 0- and q-sync algorithms scale significantly better than *loopedBLAS* for $p_s > 1$, resulting in up to 35x higher efficiencies (for order-4 tensors on node 3).

2.5 Concluding remarks

In this chapter, we propose a Morton-ordered blocked layout for tensors that achieves high-performance and mode-oblivious computations. Our *TVM* algorithms based on this layout perform as good or better than several state-of-the-art and highly-optimized BLAS-driven variants. We achieve our goal of mode-obliviousness, while all other variants

	Number of sockets (p_s)		
Algorithm	2	4	8
loopedBLAS	0.80	0.34	0.03
0-sync	0.97	0.88	0.96
q-sync	0.96	0.87	0.91

Table 2.26 – Parallel efficiency of algorithms on order-3 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket run time on a given node of q-sync algorithm on the same problem size and tensor order.

	Number of sockets (p_s)		
Algorithm	2	4	8
loopedBLAS	0.79	0.28	0.03
0-sync	0.99	0.83	1.05
q-sync	0.98	0.82	1.00

Table 2.27 – Parallel efficiency of algorithms on order-4 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket run time on a given node of q-sync algorithm on the same problem size and tensor order.

	Number of sockets (p_s)		
Algorithm	2	4	8
loopedBLAS	0.77	0.32	0.05
0-sync	0.98	0.76	1.53
q-sync	0.98	0.76	1.44

Table 2.28 – Parallel efficiency of algorithms on order-5 tensors executed on different nodes (2 socket node 1, 4 socket node 2, and 8-socket node 3), calculated against the single-socket run time on a given node of q-sync algorithm on the same problem size and tensor order.

perform several factors worse in this respect. We characterize the performance and mode-obliviousness using tensor–vector multiplications only, as this is the most bandwidth-bound operation among the other common ones including TMM, TTM, and the Khatri-Rao product. The layouts trivially extend to other operations such as the higher-order power method. They also transfer to other architectures, with significant speedups on both Ivy Bridge and Haswell nodes.

The best-performing non-blocked TVM algorithm, *tvLooped*, performs similar to the blocked $\rho_Z \rho_\pi$ TVM, but, depending on the mode of interest, computation speeds vary from as high as 14.35 GB/s to as low as 6.65 GB/s. Taking the standard deviation of the average behavior over all modes as a measure of mode-obliviousness, our blocked layout instead induces up to 3.2x more stable behavior; this is of particular benefit to use cases where kernels are not applied over each mode successively, especially when it is not known a priori which modes are of interest. Approaches that use an unfold step followed by an optimized BLAS call, although perhaps still standard practice, are inferior in terms of both performance and mode-obliviousness; bandwidth-bound operations such as the TVM incur the worst-case performance degradation of a factor two, while standard deviations magnify up to 88% of average performance. Also in the case of a compute-bound *TMM* or *TTM* one could do without copying the entire input tensor once; which is indeed unnecessary when using our proposed block layouts.

We implemented a tensor-times-a-sequence-of-vectors kernel which computes successive *TVMs* over all modes using our blocked layouts. This kernel is the computational core of the HOPM. In this kernel, the cache effects of blocking are magnified, resulting in even more pronounced performance gains for blocked layouts over successive *tvLooped* calls. For the HOPM, the blocking itself causes the largest increase in spatial locality, while for the TVM, the spatial locality is mainly induced by the Morton order of the blocks. We note that none of our improvements can be achieved on the level of BLAS libraries since we require a change in data layout.

Finally, we investigate the tensor–vector multiplication operation on shared-memory multicore machines. We explore the design space of parallel shared-memory algorithms based on this same mode-oblivious layout, and propose several candidate algorithms. After analyzing them for work, memory, intra- and inter-socket data movement, the number of barriers, and mode obliviousness, we choose to implement two of them. These algorithms, called 0-sync and q-sync, deliver close to peak performance on up four different systems, with 1, 2, 4, and 8 sockets, and surpass a baseline algorithm using *tvLooped* calls that we optimized.

Chapter 3

Sparse inference

We explore the sparse computations in neural networks by focusing on the sparse deep neural network (DNN) inference problem. The sparse DNN inference is the task of using sparse DNNs to classify a batch of data elements forming the input feature matrix. In many applications, including the one at our focus in this chapter, the input feature matrix is also sparse. As highlighted earlier in Section 1.2.2, the performance of sparse inference hinges on efficient parallelization of the sparse matrix–sparse matrix multiplication (SpGEMM) operation performed at each layer of the network. We characterize efficient sequential SpGEMM algorithms for our use case. We introduce the model-parallel inference, which uses a two-dimensional partitioning of the weight matrices. While there are different approaches for finding effective partitionings, we resort to hypergraphs, modeling with hypergraphs, and hypergraph partitioning. We propose a hypergraph model for an efficient two-dimensional nonzero partitioning of the neural network. A conceivable implementation of the model-parallel inference approach needs barriers to synchronize at layers. We propose tiling model-parallel and tiling hybrid-parallel algorithms, which use tiling to increase cache reuse between the layers and a weak synchronization module to hide load imbalance and synchronization costs. We evaluate our techniques on the large network data from the IEEE HPEC Graph Challenge 2019 on shared-memory systems and report up to 2× speed-up versus the state-of-the-art baseline. The work we present in this chapter has been accepted to be published in the proceedings of the 2020 IEEE High Performance Extreme Computing Conference (HPEC) [48].

We describe the sparse neural network inference and give the relevant background in Section 3.1. In the same section, we also review a classical sparse matrix–sparse matrix multiplication algorithm, provide background on hypergraphs, and survey the state-of-the-art parallel sparse inference literature. The Graph Challenge dataset and neural network architectures are also reviewed in this section as they have some important characteristics. We propose computational kernel implementations for sparse inference in Section 3.2. We outline various parallelization strategies for the sparse inference computations and describe the appropriate hypergraph models in Section 3.3. In the same section, we also describe our novel approach to tiling model-parallel inference with sparse deep neural networks. Section 3.4 contains experiments, which showcase the gains over the standard inference approaches. We conclude this chapter in Section 3.5.

d	The number of neural network layers
$W^{(k)}$	The kth weight matrix corresponding to the kth layer
n_k	The number of input features to the kth weight matrix $X^{(k)}$
$n_{k+1},\ k > 0$	The number of output features after applying $W^{(k)}$
$X^{(0)}$	The input feature matrix; rows are data elements, columns are input features
$X^{(d)}$	The final classification matrix, the inference output
$X^{(l)}$	The feature matrices for all $l \in \{0, \ldots, d\}$
n	The number of feature vectors, the row dimension of all $X^{(k)}$ and C
f	The activation function of the neural network
$c(= n_d)$	The number of classes, the output dimension of the last weight matrix $W^{(d-1)}$
$\mathrm{nz}(A)$	The number of nonzeros of a matrix A
w_{val}	The number of bytes to store a (nonzero) value
w_{col}	The number of bytes to store a column index
w_{rptr}	The number of bytes to point to elements of an array of size $\mathrm{nz}(A)$
$\mathrm{flops}(AB)$	The function returning the number of multiplications whenc computing AB

Table 3.1 – Notation used throughout the chapter.

Table 3.1 presents the notation used in this chapter.

3.1 Introduction

We introduce the problem of sparse neural network inference in Section 3.1.1. We present the sparse storage and sparse matrix–sparse matrix multiplication routine in Section 3.1.2. We give some definitions about hypergraphs and summarize a common hypergraph model used in scientific computing for sparse matrix partitioning in Section 3.1.3. We take a deep look at the Graph Challenge dataset in Section 3.1.4 and portray the state-of-the-art in sparse inference by reviewing Graph Challenge contenders in Section 3.1.5.

3.1.1 Sparse inference

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 [63], or result from training a fixed sparse topology as in the RadiX-Net synthetic sparse deep neural networks [2]. The input data, which is represented as a 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). The most commonly known dataset with sparse representation is the MNIST database of handwritten digits [33].

Recall from Section 1.2.2 that 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 $n_k, n_{k+1} \in \mathbb{N}$ for all layers $k \in$

$\{0,\ldots,d-1\}$. An *input feature matrix* $X^{(0)} \in \mathbb{R}^{n\times n_0}$ consists of $n \in \mathbb{N}$ sparse feature vectors, one row for each data instance to be classified. *Sparse inference* refers to the computation of the final *output matrix* $X^{(d)} \in \mathbb{R}^{n\times c}$ from the input feature matrix $X^{(0)}$:

$$X^{(d)} = f(\cdots f(f(X^{(0)}W^{(0)} + \mathbf{e}_n b^{(0)^T})W^{(1)} + \mathbf{e}_n b^{(1)^T})\cdots W^{(d-1)} + \mathbf{e}_n b^{(d-1)^T}) \,, \tag{3.1}$$

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 $\mathbf{e}_n$ is the vector of ones, $\mathbf{e}_n = (1,1,\ldots,1)^T \in \mathbb{R}^{n\times 1}$. We note that the use of biases, $b^{(k)}$ in the inference function (3.1), can be integrated with the element-wise application of f. Therefore, without loss of generality and for the sake of clarity in the expressions, we do not consider the bias explicitly in the formulas from this moment on. We furthermore write $X^{(k+1)} = f(X^{(k)}W^{(k)})$ for $0 \le k < d$ as the operation at a layer. We refer to all $X^{(l)}$ for $l \in \{0,\ldots,d\}$ as feature matrices, including the input feature and output matrices of the inference task.

The inference in this chapter uses the rectified linear unit (ReLU) as the activation function. For $Y = f(Z)$ and $Z, Y \in \mathbb{R}^{m\times n}$, ReLU sets

$$Y_{ij} = f(Z_{ij}) = \begin{cases} 0 \text{ if } Z_{ij} < 0 \\ Z_{ij}, \text{ otherwise} \end{cases} \quad \text{for } 0 \le i < m \text{ and } 0 \le j < n \,.$$

With the integration of the biases into the element-wise application of ReLU and the simple structure of this latter function, we see that for a high performance implementation of the inference function (3.1) one needs efficient sparse matrix–sparse matrix multiplication algorithms. These algorithms need to be adapted to encompass biases and activation functions, and need to be optimized for the layer-wise use case in the inference task.

3.1.2 Sparse matrix–sparse matrix multiplication

Given an $m \times \ell$ matrix A, an $\ell \times n$ matrix B, and an $m \times n$ matrix C, where all matrices are sparse, an SpGEMM computes $C = AB$. There are a number of efficient SpGEMM algorithms. These algorithms differ in the way the input and output matrices are stored, the nonzeros are visited, and the way scalar multiples are accumulated [46]. Gustavson's algorithm [26] is a well-known SpGEMM algorithm which we use in our implementation. This algorithm assumes that the two operands A and B of the multiplication operation are stored in the Compressed Row Storage (CRS) format. We review this format for the sake of completeness.

Let $\mathrm{nz}(A)$ denote the number of nonzeros of matrix A. In this format, the matrix A requires $\mathrm{nz}(A)(w_{val} + w_{col}) + (m+1)w_{rptr}$ bytes, or $\Theta(\mathrm{nz}(A) + m)$, where w_{val} is the number of bytes to store a nonzero value, w_{col} is the number of bytes to store a column index, and w_{rptr} is the number of bytes to store the values in range $\{0,\ldots,\mathrm{nz}(A)\}$. The nonzeros of A are stored in a contiguous array $A.v$ of size $\mathrm{nz}(A)$, where the nonzeros in a row are stored consecutively, starting from the first row to the last one. The column

indices of these nonzeros are stored in an array $A.col$ of size $\text{nz}(A)$ and in the same order as $A.v$. A third array $A.rptr$ of size $m+1$ stores the position of the first element of each row in the array $A.col$, where the last entry flags the end of a list of nonzeros. Hence, $A.v[A.rptr[i],\dots,A.rptr[i+1]-1]$ and $A.col[A.rptr[i],\dots,A.rptr[i+1]-1]$ access the nonzero values of the ith row and the column indices of the ith row, respectively.

Gustavson's algorithm computes the ith row of the output matrix C as the sum of the rows of B scaled by the corresponding nonzeros lying on the ith row of A. In order to sum the resulting scaled rows of B in linear time $\text{nz}(C_{i,:})$, it uses the *sparse accumulator* [24], or expanded real accumulator [53]. The sparse accumulator is an abstract data structure whose typical implementation involves two dense arrays and a list. The SPA array stores the nonzero values for the currently computed row of C. The SPAC array flags whether a nonzero in SPA belongs to the current row or not. Here, instead of using boolean values for SPAC, we use the active row index; this is known as the multiple switch technique, or the phase counter technique [25], As a result, it is more efficient as SPAC does not need to be reset between iterations for computing rows of C. The list allows quickly gathering the nonzeros of ith row of C when it is complete, without the need to scan the whole dense array SPAC of flags. This algorithm is asymptotically optimal in the sense that it has work complexity in $O(\text{flops}(AB))$.

There are usually two phases of a typical SpGEMM algorithm, including Gustavson's. The symbolic phase computes only the number of nonzeros of each row of the output matrix. The numeric phase allocates the memory and carries out the computation. The symbolic phase results in additional memory movement, but helps to avoid reallocation of matrix C at run time.

3.1.3 Hypergraph partitioning

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 $c\in\mathcal{N}$ is a subset of vertices, that is $c\subseteq\mathcal{V}$. Weights can be associated with vertices; for a vertex v, $w(v)$ denotes the weight of the vertex v, which is also extended to a set S of vertices as $w(S)=\sum_{v\in S}w(v)$. The hypergraph partitioning problem [37] is the task of dividing the vertices of a hypergraph into K parts of roughly equal weight, while minimizing an objective (cost) function defined in terms of hyperedges. That is, one seeks a partition $\Pi_{\mathcal{V}}=\{\mathcal{V}_0,\mathcal{V}_1,\dots,\mathcal{V}_{K-1}\}$ of the vertex set of $\mathcal{H}$, where parts $\mathcal{V}_k$ for $k=0,\dots,K-1$ are pairwise disjoint and mutually exhaustive. The vertex partition simultaneously induces a $(K+1)$-way partition on the hyperedges, $\Pi_{\mathcal{N}}=\{\mathcal{N}_0,\mathcal{N}_1,\dots,\mathcal{N}_{K-1};\mathcal{N}_S\}$, where the hyperedges in $\mathcal{N}_i$ for $0\le i<K$ are *internal* as they have vertices only in $\mathcal{V}_i$, while the hyperedges in $\mathcal{N}_S$ have vertices in more than one part and thus are *external*. The connectivity λ_c of a hyperedge c is equal to the number of parts it connects; internal hyperedges hence have connectivity 1, while external hyperedges may have connectivities between 2 and K. A partition $\Pi_{\mathcal{V}}$ is said to be balanced if for each part $\mathcal{V}_k$, $w(\mathcal{V}_k)\le w(\mathcal{V}/K)(1+\varepsilon)$, where ε is a constant defined by a user. There are many ways to express the objective function also known as the *cutsize*. The *cut-net*

metric [11] is one such function which simply counts the external hyperedges,

$$cutsize(\Pi_{\mathcal{V}}) = \sum_{c \in \mathcal{N}_S} 1. \tag{3.2}$$

The most common variants of this problem are NP-complete [37, Ch. 6].

In hypergraph partitioning based data-partitioning for parallelization, the number of parts corresponds to the degree of parallelism, i.e., the number of processing units or threads. Usually, the owner-computes rule is assumed so that a data unit also corresponds to a task. Each vertex corresponds to a portion of data and the corresponding atomic task, which must be stored and executed by a single processing unit. The vertex weights are set in such a way that they are proportional to the amount of work involved in the corresponding atomic task. Each part is associated with a processing unit/thread so that a balanced vertex partition leads to load balance. In most common settings about partitioning matrices, the amount work is defined as the number of nonzeros assigned to a processing unit/thread. The partitioning objective defined in terms of hyperedges straddling the cut corresponds to reducing the communication cost and data movement.

There are a number of hypergraph models and methods used for sparse matrix computations [13, 65]. Among these, the fine-grain hypergraph model [12, 14] is the most general one. The fine-grain hypergraph model associates a hypergraph $\mathcal{H}_A = (\mathcal{V}_A, \mathcal{N}_A)$ to an $m \times n$ sparse matrix A. For each nonzero of A, there is a vertex in $\mathcal{V}_A$, resulting in $\text{nz}(A)$ vertices. For each row and for each column of A, there is a hyperedge in $\mathcal{N}_A$, resulting in $m+n$ hyperedges. In this model, a vertex $v \in \mathcal{V}_A$ corresponding to a nonzero a_{ij} is contained in the two hyperedges corresponding to the ith row and the jth column of A. Under this model, a partitioning is two-dimensional as both rows and columns are represented by hyperedges, which may have connectivity larger than one. As a result, both the nonzeros lying on a single row and the nonzeros lying on a single column of A may be assigned to different threads.

3.1.4 Graph Challenge dataset

We summarize the Graph Challenge dataset in terms of input and the neural networks used. In the Graph Challenge, data consists of both a sparse feature dataset, and a number of example neural networks in the well-known Compressed Row Storage (CRS) format; for the sake of completeness we describe this format in Section 3.1.2. The Graph Challenge's input feature matrix is based on MNIST (Modified National Institute of Standards and Technology), a well-known dataset of handwritten digits [36] and 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 at layers in the neural networks: 1024 (1k), 4096 (4k), 16384 (16k), and 65536 (64k), for which the biases b in the inference function (3.1) are: $-0.3, -0.35, -0.4, -0.45$, respectively. As such large sparse networks are not readily available due to difficulties of obtaining them from real data [32], several deep sparse neural networks are generated using RadiX-Net [2], which uses mixed radices (mixed-radix numeral system) to ensure desired topological properties such as the number of connections per neuron. The sizes of

NN (defined by 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 3.2 – The sizes (in MB, assuming CRS using four bytes for values and indices) of the input feature matrix $X^{(0)}$ and the Graph Challenge neural networks with different numbers of neurons and layers.

these networks are given in Table 3.2. The filtering step requires checking if $C(i,j).v - b$ is in the range $(0, 32)$. If it's below 0, we remove this triple, while if it's greater than 32 we threshold it to 32. The number of neurons, layers, and bytes is given in Table 3.2.

Let us define the *density* of an $m \times n$ matrix as the number of nonzeros divided by mn. Let us also define the *compressed density* as the number of nonzeros divided by $m'n$ where m' is the number of nonempty rows. These two ratios are therefore related to each other by the ratio of the number of nonempty rows to the total number of rows. All weight matrices of a given NN in the Graph Challenge have equal densities. Specifically, the weight matrices of the 1k-, 4k-, 16k-, and 64k-neuron networks have densities of 3.13%, 0.78%, 0.20%, and 0.05% respectively.

We extract some statistics from the 1k-neuron, 120-layer network experiment provided in the challenge to demonstrate important properties of both the feature and weight matrices. Each weight matrix of the 1k-neuron NN occupies exactly 0.75 MB, while all 120 weight matrices occupy 90.47 MB. Table 3.3 shows the density and the compressed density for the first fifteen feature matrices $X^{(k)}$ and the size of the memory touched by the SpGEMM operation processing that feature matrix defined as the combined size of $X^{(k)}$, $X^{(k+1)}$, and $W^{(k)}$ where $0 \leq k \leq 14$. For $X^{(0)}$, all rows are nonempty and hence the compressed density matches the density. Note that for the later layers, the density is computed after the application of the activation function f. While the density is of $X^{(1)}$ is 69.37% without the ReLU, not shown in the table, and decreases to 28.33% with ReLU as reported in the table.

We observe that 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. However, we do not know if the densities would decrease in general.

3.1.5 State of the art

The Graph Challenge is a recent initiative to encourage new developments in graph analytics. The IEEE HPEC Graph Challenge 2019 posed a sparse DNN challenge [33] focused on fast inference for sparse deep neural networks, and described the original

Layer k	Condensed 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
3	23.37	8.98	56.02
4	34.28	6.86	45.98
5	56.39	6.14	36.81
6	62.56	4.26	29.07
7	85.38	3.95	26.21
8	97.23	3.45	23.63
9	99.48	3.22	22.48
10	99.93	3.12	21.93
11	99.95	3.06	21.64
12	99.91	3.04	21.54
13	99.92	3.03	21.50
14	100.00	3.02	21.47

Table 3.3 – The compressed density and the density of feature matrices $X^{(k)}$. The last column reports the total memory (in MB, assuming CRS using four bytes for values and indices) occupied by the three matrices in $X^{(k+1)} = X^{(k)}W^{(k)}$. We assume f has been applied except for the input feature matrix $X^{(0)}$.

problem using the GraphBLAS standard C API [10], which advocates the use of matrix-based algorithms for the sparse computations. The challenge focuses on deep neural networks in particular, which often perform better as a large number of layers allows for more non-linear boundaries. We base our work on the Graph Challenge, as it introduced the problem and data reflective of emerging sparse deep learning systems, and received considerable research attention to compare with.

The parallelization strategies in deep learning typically work by partitioning the input feature matrix (data-parallel), partitioning the neural network (model-parallel), and/or by pipelining [8]. The previous sparse DNN Graph 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 [9, 16, 21, 43, 67, 68] providing performance data for analysis [32]. They evaluate the performance on a common scale by fitting a line using the reported times. 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, such as fusing the bias and the ReLU application within the SpGEMM kernel.

There were two champions and two innovation awards for the sparse deep neural network challenge at the IEEE HPEC Graph Challenge 2020. Ours [48] has received an

Innovation Award thanks to a unique combination of different techniques we transfer from the world of high-performance sparse matrix computations. Another work [44], which also received an Innovation Award, ingeniously uses a different computational kernel known as the sparse matrix–dense matrix kernel (SpMM) by exploiting the structure of the Graph Challenge dataset (see Section 3.1.4). Two other studies were selected as the IEEE HPEC Graph Challenge 2020 Champions as they deliver high performance with the use of GPUs and propose batch parallelism [29] and task-based parallelism [40].

3.2 Sequential sparse inference

Sparse inference may be viewed as a repeated sparse matrix–sparse matrix multiplication $C = AB$ integrated with the filtering of nonzero elements in C using f and the addition of bias, as shown in the inference function (3.1). In Section 3.2.1, we present an SpGEMM-inference kernel which combines these operations and thus computes the sparse inference when applied at each layer. We analyze its flops and data movement requirements. We then perform the same analysis for the full sparse inference using this kernel in Section 3.2.2. Finally, we propose an SpGEMM-inference kernel for the case when the input and output matrices are partitioned in Section 3.2.3. Later in this chapter, we propose parallel sparse inference algorithms which use this kernel.

3.2.1 SpGEMM-inference kernel

Algorithm 3.1 is the SpGEMM-inference kernel which computes sparse inference when applied at each layer. It modifies Gustavson's algorithm to additionally apply the ReLU activation function f and add the bias to the nonzeros of C. As an SpGEMM, it consists of two phases, the symbolic phase (Lines 5-10) and the numeric phase (Lines 15-30). Recall that Gustavson's algorithm computes the ith row of the output matrix C as the sum of the rows of B scaled by the corresponding nonzeros lying on the ith row of A; this may be seen at Lines 16 until 21. To add bias, it suffice to simply initialize each nonzero with the bias value instead of 0 when first writing a value into SPAC for each row (Line 20). The ReLU application simply checks the nonzero value before writing it out to C storage (Line 24). Finally, we threshold the nonzeros of C as required by the Graph Challenge. If a nonzero is greater than 32 we threshold it to 32 (Line 25).

In our implementation of the sparse accumulator, the SPAC array tracks column indices belonging to the active row i (Line 19) which is standard. However, we do not keep the usual list of indices used to iterate over the accumulator in $O(\text{nz}(C_{i,:}))$ time, when writing nonzeros to C. Instead, we scan the complete SPA array in $O(n)$ time to find the nonzeros which belong to the currently active row (Line 22). Avoiding the indirection reduces the run time when rows of the output matrix hold relatively many nonzeros—which, as argued in Section 3.1.4, holds for the neural network inference.

We note that in the sparse inference context, the computed number of nonzeros in the symbolic phase may be an overestimation, as the nonzeros can be removed based on their values, according to the bias and the activation function ReLU. This is unknown

Algorithm 3.1 *SpGEMM*(A, B): The SpGEMM-inference kernel.

Input: An $m \times \ell$ matrix A and a $\ell \times n$ matrix B in the CRS format, the threshold value *thresh*, and the bias value b.
Output: An $m \times n$ matrix $C = AB$ in the CRS format.

```
1: C.rptr[0] ← 0
2: nz_C ← 0
3: for j ← 0 to n − 1 do
4:     SPAC[j] ← −1
5: for i ← 0 to m − 1 do                                          ▶ Symbolic phase
6:     for nz_A ← A.rptr[i] to A.rptr[i + 1] − 1 do
7:         for nz_B ← B.rptr[A.col[nz_A]] to B.rptr[A.col[nz_A] + 1] − 1 do
8:             if SPAC[B.col[nz_B]] ≠ i then
9:                 SPAC[B.col[nz_B]] ← i     ▶ The B.col[nz_B] entry was not yet processed
10:                nz_C ← nz_C + 1
11: Make sure C can store nz_C nonzeros
12: nz_C ← 0
13: for j ← 0 to n − 1 do
14:     SPAC[j] ← −1
15: for i ← 0 to m − 1 do                                          ▶ Numeric phase
16:     for nz_A ← A.rptr[i] to A.rptr[i + 1] − 1 do
17:         for nz_B ← B.rptr[A.col[nz_A]] to B.rptr[A.col[nz_A] + 1] − 1 do
18:             if SPAC[B.col[nz_B]] ≠ i then
19:                 SPAC[B.col[nz_B]] ← i
20:                 SPA[B.col[nz_B]] ← b
21:             SPA[B.col[nz_B]] ← SPA[B.col[nz_B]] + A.v[nz_A] · B.v[nz_B]
22:     for j ← 0 in n − 1 do
23:         if SPAC[j] = i then              ▶ Check that entry belongs to the active row
24:             if SPA[j] > 0 then            ▶ Check that it is a nonnegative value (ReLU)
25:                 if SPA[j] > thresh then          ▶ From the challenge specifications
26:                     SPA[j] ← thresh
27:                 C.col[nz_C] ← j
28:                 C.v[nz_C] ← SPA[j]
29:                 nz_C ← nz_C + 1
30:     C.rptr[i] ← nz_C                                       ▶ Store a row in the CRS
```

until the numeric phase has taken place.

We now provide an analysis of the flops and memory movement requirements. As the symbolic phase requires the same memory movement on A and B as the numeric phase, we consider only the numeric phase.

Assume that the function $\text{flops}(AB) = \sum_{i=0}^{\ell-1} \text{nz}(A_{\{:,i\}})\,\text{nz}(B_{\{i,:\}})$ returns the size of the outer products between each column of A and the corresponding row of B. This is precisely the number of multiplications required to multiply A and B. The complexity of Algorithm 3.1 is $2\,\text{flops}(AB)$ as each multiplication involves an addition (Line 21).

This algorithm accesses the CRS of the matrix B as well as the sparse accumulator structures (SPA and SPAC of size n) $\Theta(\text{nz}(A))$ times. It reads a total of $\text{flops}(AB)(w_{val} + w_{col}) + \text{nz}(A)w_{rptr}$ bytes from the CRS of B, writes to SPA $\text{flops}(AB)$ times, and to SPAC $\text{nz}(C)$ times. On the other hand, the CRS of A and the CRS of C are streamed to the CPU exactly once. Thus, the sequential cost of data movement while ignoring the data movement on the accumulator and the cache effects on B is

$$\left[\Big(\text{nz}(A) + \text{flops}(AB) + \text{nz}(C)\Big)\Big(w_{val} + w_{col}\Big) + \Big(2(2m+2) + 2\,\text{nz}(A)\Big)w_{rptr}\right]g,$$

where factor $2\,\text{nz}(A)$ accounts for reading indices of first and last element of a row of B, from $B.rptr$, for each nonzero of A. This equation yields an upper-bound on the data movement cost of

$$\mathcal{O}\Big(\Big(\text{nz}(A) + \text{flops}(AB) + \text{nz}(C) + m\Big)g\Big). \tag{3.3}$$

While the number of nonzeros of A touched during an SpGEMM is precisely $\text{nz}(A)$, the nonzeros of B that are touched during an SpGEMM are only the nonzeros on rows matching the nonempty columns of A; all nonzeros of B are required only if there are no empty columns in A. Thus, the nonzero pattern of A determines how many times each row is used and thus how many rows of B are required. If these rows may be cached, the bound on the memory movement may be as low as

$$\Omega\Big(\Big(\text{nz}(A) + \text{nz}(B) + \text{nz}(C) + m + \ell\Big)g\Big) \tag{3.4}$$

assuming no columns of A are empty. Thus, which of the two bounds applies depends on whether the required nonzeros of B may be cached as well as the nonzero structure of A.

3.2.2 Sparse inference analysis

The sparse inference (3.1) reduces to a sequence of calls to Algorithm 3.1

$$X^{(d)} = \mathit{SpGEMM}(\ldots\mathit{SpGEMM}(\mathit{SpGEMM}(X^{(0)}, W^{(0)}), W^{(1)}), \ldots, W^{(d-1)}).$$

Thus, a row i of the feature matrix $X^{(k+1)}$ is computed as the sum of the rows of the weight matrix $W^{(k)}$ first scaled by the corresponding nonzeros lying on the row i of $X^{(k)}$. This brings the complexity of sparse inference to

$$\Theta\left(\sum_{k=0}^{d-1} \text{flops}(X^{(k)}W^{(k)})\right).$$

The accumulator structures must be large enough to store $\Theta(\max_k n_{k+1})$ elements, where n_{k+1} is the number of output features of the kth weight matrix $X^{(k)}$. Two buffers suffice to store the input and output matrices $X^{(k)}$ and $X^{(k+1)}$ throughout the inference. Thus, the total storage requirement becomes

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

Based on the upper bound (3.3) on SpGEMM's data movement complexity, the upper-bound on sequential data movement overhead of the sparse inference is

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

Using the lower bound on SpGEMM's data movement complexity (3.4), we obtain the lower bound

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

on the sparse inference's data movement complexity. The lower bound can be attained when the required part of the weight matrix $W^{(k)}$ fits cache for each layer.

3.2.3 SpGEMM-inference kernel for partitioned matrices

As hinted earlier, the model-parallel inference algorithm that we propose later uses partitioned matrices for parallelism. In particular, each thread computes an SpGEMM operation $C = AB$ where matrix B is the thread-local part of the weight matrix $W^{(k)}$. Therein, matrices A and C, which correspond to the parts of the input feature matrix that a thread carrying out $C = AB$ requires and the output feature matrix parts it produces, respectively, are both partitioned and stored as four matrices in CRS. Specifically, matrix A is partitioned into $A_0 \in \mathbb{R}^{m\times \ell_0}$ and $A_1 \in \mathbb{R}^{m\times \ell_1}$ and matrix C is partitioned into $C_0 \in \mathbb{R}^{m\times n_0}$ and $C_1 \in \mathbb{R}^{m\times n_1}$ which induces the following structure on B:

$$\begin{pmatrix} C_0 & C_1 \end{pmatrix} = \begin{pmatrix} A_0 & A_1 \end{pmatrix}\begin{pmatrix} B_{00} & B_{10} \\ B_{01} & B_{11} \end{pmatrix}, \tag{3.5}$$

where $B_{00} \in \mathbb{R}^{\ell_0\times n_0}$, $B_{01} \in \mathbb{R}^{\ell_1\times n_0}$, $B_{10} \in \mathbb{R}^{\ell_0\times n_1}$, and $B_{11} \in \mathbb{R}^{\ell_1\times n_1}$. As an SpGEMM-inference (Algorithm 3.1) takes only two matrices in CRS as input, computing $C = AB$ requires the following four SpGEMMs as A_0 and A_1 each contribute to both C_0 and C_1:

$$\begin{aligned} C_0 &= A_0B_{00} \\ C_0 &= C_0 + A_1B_{01} \\ C_1 &= A_0B_{10} \\ C_1 &= C_1 + A_1B_{11}, \end{aligned} \tag{3.6}$$

Furthermore, it requires that blocks of B are in separate CRS structures.

Instead of a direct implementation of the multiplications and additions in (3.6), we propose in Algorithm 3.2 a fused SpGEMM-inference kernel. This kernel computes $C = AB$ where the matrices are split as shown in (3.5) with less data movement and uses the CRS of matrix B directly. For each row i, the fused SpGEMM-inference multiplies each nonzero in $A_{0\{i,:\}}$ and $A_{1\{i,:\}}$ with the row of B which corresponds to the nonzero column index. Here, the column index k of the nonzeros $A_{1\{i,:\}}$ must be increased by k_0 (Line 17) to match with rows of B. It stores the resulting contributions in a single accumulator of size $n_0 + n_1$. When it writes out the nonzeros in the accumulator to C_1, it decrements the column index by n_0 (Line 33).

While this algorithm performs the same number of flops as the explicit computation (3.6) using Algorithm 3.2, it reduces the data movement by reading A_0 and A_1 and writing C_0 and C_1 only once instead of twice. Note that it applies the activation function ReLU, bias, and the thresholding step to C_0 but not to C_1, which becomes later useful for the parallel algorithm.

3.3 Data-, model- and hybrid-parallel inference

The parallel sparse inference algorithm depends on which of the two inputs, the feature matrix or the neural network, is explicitly partitioned. The former corresponds to the row-wise partitioning of the feature matrix which yields the *data-parallel* inference, the de-facto standard in deep learning. The latter corresponds to the *model-parallel* inference, where each thread stores a part of each weight matrix. In our variant, each thread is assigned some of the nonzeros of each weight matrix according to the two-dimensional nonzero partitioning of the neural network we obtain using the proposed hypergraph model. We later combine it with tiling and a synchronization mechanism. A third parallel variant is the tiling *hybrid-parallel* inference which allows the user to balance between the costs of both parallelization methods, and describe application of the resulting method to the deep neural network inference. Finally, we discuss implementation issues and experiment using our implementations on shared-memory systems.

3.3.1 Data-parallel inference

In this section, we describe and analyze the data-parallel inference algorithm. This algorithm is the state-of-the-art for parallel inference and serves as the baseline later in our experiments (Section 3.4). In general, data-parallel inference uses a row-wise partitioning $\pi_{\mathrm{R}} : \{0, 1, \dots, n-1\} \to \{0, 1, \dots, p-1\}$ of the input feature matrix $X^{(0)} \in \mathbb{R}^{n \times n_0}$ into p parts, where p is the number of threads. In our implementation, we map row i to thread $\lfloor i/\lceil n/p \rceil \rfloor$, which is a block partitioning with block size of $b_{\mathrm{1D}} = \lceil n/p \rceil$. Let s_q denote the set of indices assigned to thread q for all threads q, i.e., ones for which $\pi_{\mathrm{R}}(i) = q$. With this definition, $X^{(0)}_{\{s_q,:\}}$ denotes the rows of the input feature matrix assigned to the thread q. To proceed with inference on the n data inputs, each thread performs sequential inference on its block of input data elements in an embarrassingly parallel fashion. In other words, the thread q executes an SpGEMM (with biases and the activation function

Algorithm 3.2 *fusedSpGEMM*(A_0, A_1, B): The SpGEMM-inference kernel for partitioned matrices.

Input: The matrices $A_0 \in \mathbb{R}^{m\times\ell_0}$, $A_1 \in \mathbb{R}^{m\times\ell_1}$, and $B \in \mathbb{R}^{\ell\times n}$ in CRS format, the threshold value *thresh*, and the bias value b.
Output: The matrices $C_0 \in \mathbb{R}^{m\times n_0}$, $C_1 \in \mathbb{R}^{m\times n_1}$ of $(C_0|C_1) = (A_0|A_1)B$ in CRS.

1: $C_0.rptr[0] \leftarrow 0$, $C_1.rptr[0] \leftarrow 0$
2: $\cdots$ ▶ Symbolic phase not shown
3: Make sure C_0 can store nz_{C_0} nonzeros
4: Make sure C_1 can store nz_{C_1} nonzeros
5: $nz_{C_0} \leftarrow 0$, $nz_{C_1} \leftarrow 0$
6: **for** $j \leftarrow 0$ to $n-1$ **do**
7:     $\mathsf{SPAC}[j] \leftarrow -1$
8: **for** $i \leftarrow 0$ to $m-1$ **do**
9:     **for** $nz_A \leftarrow A_0.rptr[i]$ to $A_0.rptr[i+1]-1$ **do**
10:         $col \leftarrow A_0.col[nz_A]$ ▶ Use the original column index
11:         **for** $nz_B \leftarrow B.rptr[col]$ to $B.rptr[col+1]-1$ **do**
12:             **if** $\mathsf{SPAC}[B.col[nz_B]] \neq i$ **then**
13:                 $\mathsf{SPAC}[B.col[nz_B]] \leftarrow i$
14:                 $\mathsf{SPA}[B.col[nz_B]] \leftarrow b$
15:             $\mathsf{SPA}[B.col[nz_B]] \leftarrow \mathsf{SPA}[B.col[nz_B]] + A_0.v[nz_A] \cdot B.v[nz_B]$
16:     **for** $nz_A \leftarrow A_1.rptr[i]$ to $A_1.rptr[i+1]-1$ **do**
17:         $col \leftarrow A_1.col[nz_A] + k_0$ ▶ Add offset k_0 to the column index
18:         **for** $nz_B \leftarrow B.rptr[col]$ to $B.rptr[col+1]-1$ **do**
19:             **if** $\mathsf{SPAC}[B.col[nz_B]] \neq i$ **then**
20:                 $\mathsf{SPAC}[B.col[nz_B]] \leftarrow i$
21:                 $\mathsf{SPA}[B.col[nz_B]] \leftarrow b$
22:             $\mathsf{SPA}[B.col[nz_B]] \leftarrow \mathsf{SPA}[B.col[nz_B]] + A_1.v[nz_A] \cdot B.v[nz_B]$
23:     **for** $j \leftarrow 0$ to $n-1$ **do**
24:         **if** $\mathsf{SPAC}[j] = i$ **then** ▶ Check that entry belongs to the active row
25:             **if** $j < n_1$ **then** ▶ It belongs to C_0
26:                 **if** $\mathsf{SPA}[j] > 0$ **then** ▶ Check that it is a nonnegative value (ReLU)
27:                     **if** $\mathsf{SPA}[j] > thresh$ **then** ▶ As before
28:                         $\mathsf{SPA}[j] \leftarrow thresh$
29:                     $C_0.col[nz_{C_0}] \leftarrow j$
30:                     $C_0.v[nz_{C_0}] \leftarrow \mathsf{SPA}[j]$
31:                     $nz_{C_0} \leftarrow nz_{C_0} + 1$
32:             **else** ▶ This nonzero belongs to C_1 (no ReLU or thresholding)
33:                 $C_1.col[nz_{C_1}] \leftarrow j - n_0$ ▶ Adjust the index to fit the storage
34:                 $C_1.v[nz_{C_1}] \leftarrow \mathsf{SPA}[j] - b$ ▶ Store without applying ReLU or bias
35:                 $nz_{C_1} \leftarrow nz_{C_1} + 1$
36:     $C_0.rptr[g] \leftarrow nz_{C_0}$ ▶ Store the two rows in the respective CRS structures
37:     $C_1.rptr[g] \leftarrow nz_{C_1}$

ReLU) at each layer k for $k = 0, \ldots, d-1$. Here, at layer $k = 0$, the two input arguments of the SpGEMM are $X^{(0)}_{\{s_q,:\}}$, which is the input matrix assigned to thread q, and the whole weight matrix $W^{(0)}$ respectively. The output of the SpGEMM carried out by the thread q at any layer is always of the same shape as $X^{(0)}_{\{s_q,:\}}$ and becomes the first input argument of the SpGEMM of the next layer. The second input argument of the SpGEMM at any layer k is the whole weight $W^{(k)}$. Each thread q allocates two buffer matrices to store the intermediate local feature matrices $X^{(k)}_{\{s_q,:\}}$ and $X^{(k+1)}_{\{s_q,:\}}$ for computing $X^{(k)}_{\{s_q,:\}}W^{(k)}$ throughout the inference; that is why only two buffers are used. On the other hand, the weight matrices are allocated once globally and shared. The data-parallel algorithm has no work overhead compared to a sequential method.

To proceed with analysis, we recall from Section 1.1.2 that g and h denote the time to transfer a byte from local memory and remote memory, respectively, to a thread, while L denotes the time to complete a barrier. These quantify performance of an abstract machine consisting of p_s sockets, where each socket may run up to p_t threads. The worst-case memory movement occurs when the weight matrices do not fit in cache while only p_t out of p threads may access the weight matrices locally as they are stored at one socket. Then, the worst-case data movement overhead is

$$\mathcal{O}\left(\sum_{k=0}^{d-1} \text{flops}(X^{(k)}W^{(k)})(1 - 1/p_s)(h - g)\right).$$

In fact, it is determined by the partition with the largest workload,

$$\mathcal{O}\left(\sum_{k=0}^{d-1} \left[p \max_q \text{flops}(X^{(k)}_{\{s_q,:\}}W^{(k)})\right] h\right).$$

On the other hand, if elements of each weight matrix fit in cache, the minimum data movement overhead is

$$\Omega\left(\sum_{k=0}^{d-1}\left(\text{nz}\left(W^{(k)}\right) + n_k\right)\left((p_t - 1)g + p_t(p_s - 1)h\right)\right), \tag{3.7}$$

as all threads need to refer to the shared weights at least once. We note that the actual overhead may be closer to the lower or the upper bound as it depends on the size of $W^{(k)}$, which might change between the layers. By replicating the weight matrices on each socket, all threads access the weight matrices locally which further reduces the overhead in the above formula to

$$\Omega\left(\sum_{k=0}^{d-1}(\text{nz}\left(W^{(k)}\right) + n_k)(p - 1)g\right).$$

Finally, since we use CRS to store the disjoint partitions $X^{(k)}_{\{s_q,:\}}$, there is almost no storage overhead except for the extra row pointer array entries, which yield an overhead of $\Theta(p)$ integers.

3.3.2 Model-parallel inference

We assume a mutually disjoint p-way partition $\mathbf{W}^{(k)} = \{W_0^{(k)}, \ldots, W_{p-1}^{(k)}\}$ for each layer k which maps each individual nonzero of the weight matrix $W^{(k)}$ to a unique thread, where $W_q^{(k)}$ is mapped to thread q. Let $\mathbf{R}^{(k)} = \{r_0^{(k)}, \ldots, r_{p-1}^{(k)}; r_S^{(k)}\}$ be an assignment of row indices of $W^{(k)}$ such that all nonzeros on rows in $r_i^{(k)}$ of the weight matrix belong to part $W_i^{(k)}$ only, and are internal, while the nonzeros on each row in $r_S^{(k)}$ belong to multiple parts of $\mathbf{W}^{(k)}$ and thus are external. Let $\mathbf{Z}^{(k)} = \{z_0^{(k)}, \ldots, z_{p-1}^{(k)}; z_S^{(k)}\}$ be a similar assignment of column indices. When the layer number k can be inferred from the context, we shall omit the superscripts and use $\{r_0, \ldots, r_{p-1}; r_S\}$ and $\{z_0, \ldots, z_{p-1}; z_S\}$ to refer to $\mathbf{R}^{(k)}$ and $\mathbf{Z}^{(k)}$, respectively.

Let i be a column index of $W^{(k)}$ and a corresponding row index of $W^{(k+1)}$. In general, it may be that i is internal in $\mathbf{Z}^{(k)}$ and external in $\mathbf{R}^{(k+1)}$, or vice versa. Even if it is internal in both, it may belong to a different part in $\mathbf{Z}^{(k)}$ than in $\mathbf{R}^{(k+1)}$. Handling mismatched assignments and communication between the resulting internal and external parts of the partitionings efficiently during inference is possible using techniques from sparse matrix computations (see, e.g., Yzelman and Roose [72]). However, we save that exercise for future work, and instead inflate the external set to ensure that we do not need to permute columns of $X^{(k)}$ before processing layer $k+1$. For $k \in \{0, \ldots, d-2\}$ and $q \in \{0, \ldots, p-1\}$, let $i_q^{(k)} = z_q^{(k)} \cap r_q^{(k+1)}$ be the new set of indices assigned to thread q, which consists of the indices which are simultaneously assigned to q in $z_q^{(k)}$ and $r_q^{(k+1)}$. Also for $k \in \{0, \ldots, d-2\}$, let

$$i_S^{(k)} = \left(z_S^{(k)} \cup r_S^{(k+1)}\right) \cup \bigcup_{q=0}^{p-1} \left(z_q^{(k)} \cup r_q^{(k+1)} \setminus i_q^{(k)}\right)$$

be the new set of external indices which consist of all indices in $z_S^{(k)}$ and $r_S^{(k+1)}$ and all other indices in $\mathbf{Z}^{(k)}$ and $\mathbf{R}^{(k+1)}$ which are not contained in any $i_q^{(k)}$ for $q \in \{0, \ldots, p-1\}$. We then initialize sets $\mathbf{Z}^{(k)}$ and $\mathbf{R}^{(k+1)}$ to $\{i_0^{(k)}, i_1^{(k)}, \ldots, i_{p-1}^{(k)}; i_S^{(k)}\}$ such that $\mathbf{Z}^{(k)} = \mathbf{R}^{(k+1)}$; the sets $\mathbf{R}^{(0)}$ and $\mathbf{Z}^{(d-1)}$ need not be changed. Thus, the column assignment of $W^{(k)}$ matches to the row assignment of $W^{(k+1)}$ for all $k \in \{0, \ldots, d-2\}$. We note that the local nonzeros of each thread $W_q^{(k)}$ are contained in submatrix $W_{\{r_q \cup r_S, z_q \cup z_S\}}^{(k)}$ despite the change in $\mathbf{Z}^{(k)}$ and $\mathbf{R}^{(k+1)}$.

We use the subscript q to emphasize the locality to show that a matrix is stored at thread q. Let $W_{\{r_q \cup r_S, z_q \cup z_S\}_q}^{(k)}$ for all layers k be the set of d matrices consisting of the nonzeros assigned to thread q in part $W_q^{(k)}$. This is known as the doubly-bordered block-diagonal form [3]. At each layer k of inference, thread q must compute $X_{\{:, z_q \cup z_S\}}^{(k+1)} \leftarrow X_{\{:, r_q \cup r_S\}}^{(k)} W_{\{r_q \cup r_S, z_q \cup z_S\}_q}^{(k)}$. We now discuss the locality of nonzeros in $X^{(k)}$ and $X^{(k+1)}$. The feature matrix parts $X_{\{:, r_S\}}^{(k)}$ and $X_{\{:, z_S\}}^{(k+1)}$ are read and written, respectively, by potentially all threads at each layer k; we refer to $X_{\{:, r_S\}}^{(k)}$ as the *input separator* and $X_{\{:, z_S\}}^{(k+1)}$ as the *output separator*. We distribute these among all threads using the block row-wise partitioning from Section 3.3.1 such that $X_{\{s_r, r_S\}_r}^{(k)} = X_{\{s_r, r_S\}}^{(k)}$ and $X_{\{s_r, z_S\}_r}^{(k+1)} = X_{\{s_r, z_S\}}^{(k+1)}$

for each $0 \leq r < p$. The elements of feature matrix parts $X^{(k)}_{\{:,r_q\}}$ and $X^{(k+1)}_{\{:,z_q\}}$ are used exclusively by thread q. Thus, we store elements $X^{(k)}_{\{:,r_q\}_q} = X^{(k)}_{\{:,r_q\}}$ and $X^{(k+1)}_{\{:,z_q\}_q} = X^{(k+1)}_{\{:,z_q\}}$ locally at thread q.

Instead of threads writing to the output separator in parallel, we have threads write their *partial results* locally and perform a collaborative summation afterwards. Therefore, each thread q may perform all its multiplications at layer k without conflicts. These may be illustrated using the structured multiplication in (3.5)

$$\begin{pmatrix} X^{(k+1)}_{\{s_r,z_q\}_q} & X^{(k+1)}_{\{s_r,z_S\}_q} \end{pmatrix} = \begin{pmatrix} X^{(k)}_{\{s_r,r_q\}_q} & X^{(k)}_{\{s_r,r_S\}_r} \end{pmatrix} \begin{pmatrix} W^{(k)}_{\{r_q,z_q\}_q} & W^{(k)}_{\{r_q,z_S\}_q} \\ W^{(k)}_{\{r_S,z_q\}_q} & W^{(k)}_{\{r_S,z_S\}_q} \end{pmatrix}.$$

As shown earlier, this yields four SpGEMMs (3.6) which we carry out using *fusedSpGEMM* (Algorithm 3.2) to lower the data movement (see Section 3.2.3).

We propose Algorithm 3.3 to carry out the model-parallel inference at layer k. In this algorithm, each thread q proceeds using two-phase reduction:

1. it computes the multiplications using *fusedSpGEMM* on each row-wise partition of the input feature matrix parts (Lines 1-2), and all threads synchronize after the for-loop;

2. it computes the sum of row-wise partition s_q of the partial results and stores the resulting separator locally (Line 4).

Taking L as the machine parameter describing the time in which a barrier completes in seconds (as in Section 1.1.2), such two-phase reduction has a cost of $2L$ seconds for the two barriers. This assumes that the number of rows n is larger than or equal to p. This algorithm has no work overhead in flops as the summations which normally would take place during a sequential SpGEMM are just delayed due to having to reduce them over multiple threads. As each thread is responsible for reducing approximately n/p rows, the resulting sums are spread across all threads allowing a balanced access pattern when they are accessed at the next layer as $\sum_{r=0}^{p-1} X^{(k+1)}_{\{s_r,r_S\}_r}$. Ignoring the symbolic phase and assuming each local weight matrix can be cached, the data movement overhead is due to reading the input separator parts $X^{(k)}_{\{s_r,r_S\}_r}$ and writing and reading the partial results $X^{(k+1)}_{\{s_r,z_S\}_q}$ and $X^{(k+1)}_{\{s_q,z_S\}_r}$, respectively.

Each thread q stores $W^{(k)}_{\{r_q \cup r_S, z_q \cup z_S\}_q}$ in CRS for each layer and uses $4p$ buffer matrices in CRS to store different feature matrix parts between the layers. These are used to store the two-phase reduction parts $X^{(k)}_{\{s_r,r_S\}_q}$ and $X^{(k+1)}_{\{s_r,z_S\}_q}$ and the local parts $X^{(k)}_{\{s_r,r_q\}_q}$ and $X^{(k+1)}_{\{s_r,z_q\}_q}$ such that each part of rows s_r for $r \in \{0, \ldots, p-1\}$ is stored separately. Overall, this yields pd and $4p^2$ matrices in CRS for all threads. The parallel storage overhead is due to the extra nonzeros stored in the partial results matrices and the duplicated index array entries in CRS and thus adds up to

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

Algorithm 3.3 *processLayer*: The model-parallel layer-k inference at thread q.

Input: The local parts $X^{(k)}_{\{s_r,r_q\}_q}$ for $r \in \{0,\ldots,p-1\}$,
the separator parts $X^{(k)}_{\{s_r,r_S\}_r}$ for $r \in \{0,\ldots,p-1\}$,
and the local weight matrix $W^{(k)}_{\{r_q\cup r_S, z_q\cup z_S\}_q}$.
Output: The local parts $X^{(k+1)}_{\{s_r,z_q\}_q}$ for $r \in \{0,\ldots,p-1\}$
and the local separator part $X^{(k+1)}_{\{s_q,z_S\}_q}$.

1: **for** $r \leftarrow 0$ to $p-1$ **do**
2: $\quad X^{(k+1)}_{\{s_r,z_q\}_q}, X^{(k+1)}_{\{s_r,z_S\}_q} \leftarrow \mathit{fusedSpGEMM}\left(X^{(k)}_{\{s_r,r_q\}_q}, X^{(k)}_{\{s_r,r_S\}_r}, W^{(k)}_{\{r_q\cup r_S, z_q\cup z_S\}_q}\right)$
3: **exec** barrier ▶ Executes only for $k \neq 0$
4: $X^{(k+1)}_{\{s_q,z_S\}_q} \leftarrow f(\sum_{r=0}^{p-1} X^{(k+1)}_{\{s_q,z_S\}_r})$
5: **exec** barrier

The synchronization overhead $\Theta(dpL)$ is due to each thread executing a barrier at each layer. The total data movement overhead

$$\Theta\Big(\sum_{k=0}^{d-1}\Big((pn + p|r_s^{(k)}|)g + (\mathrm{nz}(X^{(k)}_{\{:,r_S\}}) + n + p)h + (p(n+p) + \sum_{r=0}^{p-1}\mathrm{nz}(X^{(k+1)}_{\{:,z_S\}_r}))(g+h)\Big)\Big) \tag{3.8}$$

is related to the overhead of reading the CRS indexing arrays of pd weight matrices, the input separators in CRS, and the reading and writing of partial results. Taking account of the load balance of the feature partitions, the data movement overhead is proportional to

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

Thus, ideally the nonzeros in input separators $\sum_{r=0}^{p-1} X^{(k)}_{\{s_r,r_S\}_r}$ as well as the nonzeros in partial results $\sum_{r=0}^{p-1} X^{(k+1)}_{\{s_q,z_S\}_r}$ for each thread q are split equally between p_s processors.

We note that both the parallel storage and the data movement overheads depend on the size of the partial results after reduction. This changes according to the nonzero pattern and is within the range $\mathrm{nz}(X^{(k+1)}_{\{:,z_S\}}) \leq \sum_{r=0}^{p-1}\mathrm{nz}(X^{(k+1)}_{\{:,z_S\}_r}) \leq p\,\mathrm{nz}(X^{(k+1)}_{\{:,z_S\}})$ where f was not applied to $X^{(k+1)}_{\{:,z_S\}}$.

Comparing to the data movement overhead of the data-parallel method (3.7), 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 $W^{(k)}$. If the nonzeros $W^{(k)}_q$ do not fit in cache, an additional data movement $\Theta(\max_q[\mathrm{nz}(W^{(k)}_q) + |r_q \cup r_S|]g)$ occurs. This results in an additional data movement overhead of $\Theta(((p\sum_{k=0}^{d-1}\max_q|r_q| + |r_S| - n_k/p) + \varepsilon_k\sum_k \mathrm{nz}(W^{(k)}))g)$, with ε_k is the imbalance among the number of nonzeros assigned to different threads at layer k.

3.3.2.1 Hypergraph model for model-parallel inference

In this section, we present a hypergraph model we use to obtain the nonzero partitioning of the weight matrices for the model-parallel inference. The standard models and tools may be used to partition the weight matrices independently layer by layer to partition the whole neural network and the associated computations. However, if weight matrices are partitioned independently, it is unlikely that their partitionings will match at any index. In other words, the nonzeros lying on a column of $W^{(k)}$ may be assigned to arbitrarily different thread(s) than the nonzeros lying on a matching row of $W^{(k+1)}$. This requires communication between each two consecutive layers to align the matching row and columns, which is costly.

Recall that sparse inference reduces to a series of d successive SpGEMMs. For layers $k \in \{0, \ldots, d-2\}$, the feature matrix $X^{(k+1)}$ is simultaneously an output of $X^{(k)}W^{(k)}$ and an input to $X^{(k+1)}W^{(k+1)}$. Therefore, the nonzeros lying at column i of $X^{(k+1)}$ (i) contain contributions of the nonzeros lying at column i of $W^{(k)}$ and (ii) are multiplied with the nonzeros lying at row i of $W^{(k+1)}$, for all $i \in \{0, \ldots, n_k - 1\}$ and $k \in \{0, \ldots, d-2\}$. When partitioning a neural network for sparse inference, we want to prevent that these nonzeros lie at different threads. Therefore, we want that the nonzeros in a given column of $W^{(k)}$ belong to the same thread which has the nonzeros at the matching row of $W^{(k+1)}$. To do so, we propose a hypergraph model in which there is only one hyperedge for a pair of matching indices, e.g., a column of $W^{(k)}$ and its matching row of $W^{(k+1)}$. Under such model, memory movement between threads due to assignment mismatches is minimized. To describe this model, we use a matrix in which the weight matrices are aligned at their matching indices to visually form a staircase.

Given a neural network with k layers, we first transpose each weight matrix $W^{(k)}$ for which k is an even number. We then arrange all matrices blockwise within a larger matrix. Particularly, we connect each transposed matrix $W^{(k)}$ with a successive weight matrix $W^{(k+1)}$ horizontally (if $k < d-1$) and a preceding weight matrix $W^{(k-1)}$ vertically (if $k > 0$). Two successive weight matrices are now aligned at the matching dimensions. Figure 3.1 shows an example *staircase matrix* built using this procedure. The *staircase hypergraph model* is then a fine-grain model of the staircase matrix complemented by the following for partitioning purposes:

1. a vector of d weights is assigned to each vertex corresponding to a nonzero of $W^{(k)}$ where the kth weight is 1 while all other weights are 0;
2. the cut-net metric (3.2) should be used.

The former ensures that threads have roughly an equal amount of work when collaboratively processing a layer by load balancing weights of a single layer across threads while the latter ensures that the sizes of the external rows and columns, or the separators, are reduced, as they are directly related to the cut-net metric.

We use the fine-grain model as we are interested in two-dimensional partitionings. Other partitioning models and methods can also be used. We use PaToH [11], a multi-constraint hypergraph partitioner, to obtain a p-way partitioning of the staircase hypergraph. We know which vertex in such partitioning represents a nonzero of which weight

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

Figure 3.1 – The staircase matrix of a neural network consisting of d layers. Here, d is even.

matrix. Thus, we use it to initialize the partitions $\mathbf{W}^{(k)}$ for $k \in \{0, \ldots, d-1\}$ of the weight matrices.

3.3.2.2 Tiling model-parallel inference

Tiling, as described earlier in Section 1.1.1, increases data reuse by prematurely pausing a loop's iteration such that its output may be used by the subsequent iteration immediately, while they are still in cache. However, tiling techniques may incur overheads such as recomputation of factors needed to continue a paused iteration.

Algorithm 3.4 is the tiling model-parallel algorithm in which threads use the partitioned weight matrices (Section 3.3.2) to compute model-parallel inference over a batch of feature matrix at each layer. Assuming batch size b_{size} for which the intermediate feature matrices for $0 < k < d$ remain in cache, each thread q reads from the input feature matrix $X^{(0)}_{\{:,r_q\}_q}$ only b_{size} rows at a time (Line 6), where $n \geq b_{size}$. It also reads $\lceil b_{size}/p \rceil$ rows from the input matrix part $X^{(0)}_{\{:,r_S\}_q}$ (Line 7). As the two-phase reduction is now over b_{size} rows, it requires that $b_m \geq p$. Each thread q uses $2p$ in-cache buffer matrices C and D to store the intermediate feature matrices elements, where C and D are both split row-wise for the two-phase reduction: $\{C_{\{s_0,r_q\}_q}, C_{\{s_1,r_q\}_q}, \ldots, C_{\{s_{p-1},r_q\}_q}\}$ and $\{D_{\{s_0,r_S\}_q}, D_{\{s_1,r_S\}_q}, \ldots, D_{\{s_{p-1},r_S\}_q}\}$. We use functions *checkAvail*, *resetAvail*, and *signalAvail*, which are part of a weak point-to-point synchronization module, to avoid $2dp$ explicit barriers for each of the $t = \lceil n/b_{size} \rceil$ batches. Each thread uses *signalAvail* to set a boolean in a shared memory communicating it has completed one of the phases, which other threads may check for using *checkAvail*; *resetAvail* sets the boolean to false. In the first reduction phase (Lines 10-18), each thread only waits if none of the partial results are ready and may otherwise continue working with any available remote partial results. In the second reduction phase (Lines 20-26), each thread q starts with iteration q of the loop as the local input separator $D_{\{s_q,r_S\}_q}$ must be ready. It then overlaps the barrier with the computation of local results by processing any available contributions of remote threads $D_{\{s_r,r_S\}_{r \neq q}}$ instead of waiting for all threads to complete the previous phase. Thus, the synchronization overhead lies far below $\mathcal{O}(tdpL)$ in practice. When the

inference completes, each thread writes out the computed elements to the large classification matrix $X^{(d)}$.

The data movement overhead can be derived similarly to that of the model-parallel inference (3.3.2), except it is multiplied by the number t of batches 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 the underlying neural networks can be partitioned. This algorithm may be especially effective when the local parts of the neural network fit in cache as well.

3.3.3 Hybrid-parallel inference and deep inference

As the model-parallel variant may be freely mixed with the data-parallel variant, we propose a hybrid approach which combines both the data-parallel and the tiling model-parallel inference.

Assuming $p = p_0 p_1$ threads, the hybrid inference splits n data items into p_0 parts using a block distribution, and groups of p_1 threads process their part using the tiling model-parallel inference. The weight matrices are allocated by the threads of one of the p_0 groups. There is a maximum number of threads we may dedicate to the model-parallel inference due to the fact that normally, the separator size increases with increasing p_1. Therefore, we select p_1 for which the speedup of the tiling model-parallel method is best. We then choose $p_0 \geq \lfloor p/p_1 \rfloor$ which enables the use of all available cores.

The tiling model-parallel method is most beneficial if not only the intermediate feature matrices fit in cache but the weight matrices of the neural network as well. However, inference computations on deep neural networks, where the weight matrices do not fit cache, may also use the tiling model-inference or hybrid-parallel inference. To enable deep inference, we may then cut the network's layers into successive blocks, and apply the tiling method for each block. For example, if $d = 120$, as in one of the neural networks of the Graph Challenge, and we create blocks of 5 layers, inference should invoke the tiling algorithm 24 times. This requires streaming from the main memory not only $X^{(0)}$, but all of $X^{(5k)}$, for $0 \leq k < 24$, which happens each time we start processing a block. An appropriate block size could be selected automatically by greedily growing blocks of layers while they remain cacheable and the separator size remains below a certain threshold. The hybrid method then can deal with the case when $p \gg p_1$. More elaborate schemes may be envisioned; with whatever the secondary details, 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: (i) a block size for which the combined consecutive layers fit cache; and (ii) a tile size for which the intermediate results fit cache as well. In the experiments, we demonstrate that the proposed general approach works for the inputs and neural networks proposed in the Graph Challenge.

Algorithm 3.4 The tiling model-parallel inference at thread q.

Input: The local parts $X^{(0)}_{\{s_r,r_q\}_q}$ for $r \in \{0,\ldots,p-1\}$,
the separator parts $X^{(0)}_{\{s_r,r_S\}_r}$ for $r \in \{0,\ldots,p-1\}$,
the local weight matrices $W^{(k)}_{\{r_q\cup r_S, z_q\cup z_S\}_q}$ for $k \in \{0,\ldots,d-1\}$,
and the tile size b_{size}.
Output: The local parts $X^{(d)}_{\{s_r,z_q\}_q}$ for $r \in \{0,\ldots,p-1\}$
and the local separator part $X^{(d)}_{\{s_q,z_S\}_q}$.

1: **for** $m \leftarrow 0$ to $\lceil n/b_{size}\rceil - 1$ **do**
2: $b_m \leftarrow \{mb_{size}, mb_{size}+1,\ldots,\min\{(m+1)b_{size}, n\}-1\}$ ▶ The current batch indices
3: Let $\{u_0,\ldots,u_{p-1}\}$ be a block partitioning of b_m into p parts.
4: Let D_q be a $b_m \times |r_q \cup r_S|$ in-cache matrix.
5: Let C_q be a $b_m \times |z_q \cup z_S|$ in-cache matrix.
6: $D_{\{b_m,r_q\}_q} \leftarrow X^{(0)}_{\{b_m,r_q\}_q}$ ▶ Read the local part into cache
7: $D_{\{u_q,r_S\}_q} \leftarrow X^{(0)}_{\{u_q,r_S\}_q}$ ▶ Read the separator into cache
8: $resetAvail(C_{\{:,r_S\}_q})$, $resetAvail(D_{\{u_q,r_S\}_q})$
9: **for** $k = 0$ to d **do**
10: **if** $k \neq 0$ **then** ▶ Inter-thread data movement
11: $r = (q+1)$ **mod** p, $done = 1$
12: $checkAvail(D_{\{:,r_S\}_q})$ ▶ Our partial results are added in-place
13: **while** $done < p$ **do**
14: **if** $checkAvail(D_{\{:,r_S\}_r})$ **then**
15: $D_{\{u_q,r_S\}_q} \leftarrow D_{\{u_q,r_S\}_q} + D_{\{u_q,r_S\}_r}$
16: $done \leftarrow done + 1$
17: $r \leftarrow (r+1)$ **mod** p
18: $D_{\{u_q,r_S\}_q} \leftarrow f(D_{\{u_q,r_S\}_q})$ ▶ Delayed ReLU
19: $resetAvail(C_{\{u_q,r_S\}_q})$, $signalAvail(D_{\{u_q,r_S\}_q})$
20: **if** $k \neq d$ **then** ▶ Intra- and inter-thread data movement
21: $r = q$, $done = 0$
22: **while** $done < p$ **do**
23: **if** $checkAvail(D_{\{u_r,r_S\}_r})$ **then**
24: $C_{\{u_r,z_q\}_q}, C_{\{u_r,z_S\}_q} \leftarrow fusedSpGEMM\left(D_{\{u_r,r_q\}_q}, D_{\{u_r,r_S\}_r}, W^{(k)}_{\{r_q\cup r_S,z_q\cup z_S\}_q}\right)$
25: $done \leftarrow done + 1$
26: $r \leftarrow (r+1)$ **mod** p
27: $resetAvail(D_{\{:,r_S\}_q})$, $signalAvail(C_{\{:,r_S\}_q})$
28: Swap C and D
29: $X^{(d)}_{\{b_m,z_q\}} \leftarrow D_{\{b_m,z_q\}_q}$ ▶ Write back the local part into memory
30: $X^{(d)}_{\{u_q,z_S\}} \leftarrow D_{\{u_q,z_S\}_q}$ ▶ Write back the separator into memory

3.3.4 Implementation details

We implement three sparse inference algorithms: the data-parallel, the tiling model-parallel, and the tiling hybrid-parallel inference. Our data-parallel inference implementation uses the SpGEMM-inference kernel (Algorithm 3.1). Recall from Section 1.1.4, explicit allocation means each thread allocates its own data. In all algorithms, we allocate all data explicitly. The input and output buffers are allocated by the thread which writes to it, such that only data reads may be remote. In all algorithms, the weight matrices are allocated by a single thread. As the choice of the data type for nonzero values has a strong effect on performance [16], we optimize it to the Graph Challenge data and use floats to store values ($w_{val} = 4$) and integers for column and nonzero indices ($w_{col}, w_{rptr} = 4$). 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.

We note that at Line 14 of Algorithm 3.4, it is enough to check if $checkAvail(D_{\{s_q,r_S\}_r})$ is true, instead of $checkAvail(D_{\{b_m,r_S\}_r})$, as each thread q depends only on the rows s_q from remote partial results. However, such conditioning with lower granularity requires more complex synchronization structures. We also note that each thread q may perform multiplications $D_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_q\}_q}$ and $D_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_S\}_q}$, for each $r \in \{0, \ldots, p-1\}$, without the need to wait for remote contributions, as these multiplications involve local parts of the feature matrix only. Algorithm 3.5 is a latency-hiding model-parallel algorithm which exploits this and carries out these multiplications (Lines 2-3 in Algorithm 3.5) first. This delays the execution of the barrier and thus hides latency when the workloads or memory accesses are not well-balanced between the threads. However, as the SpGEMMs are now split in two groups, it does not use the *fusedSpGEMM* and touches the output twice. Although this should not hurt performance as the tiling algorithm based on this model-parallel variant keeps the output in cache, our early experiments show that it performs worse than Algorithm 3.3, potentially due to the fact that inference over the Graph Challenge dataset is well-balanced; we do not investigate its performance further.

3.4 Experiments

We first introduce the experimental methodology in Section 3.4.1, including the tile size selection. We then proceed to test our implementations of the data-parallel inference, the tiling model-parallel inference, and the tiling hybrid-parallel inference on the Graph Challenge neural networks. For the purpose of this chapter, we calculate the speedup of our proposed algorithms with respect to our data-parallel inference, as it is the common parallelization in the state-of-the-art; this means the baseline is a parallel algorithm. In all experiments, we use the first five layers of the Graph Challenge networks (i) to confirm the beneficial effects of caching the intermediate results, and (ii) to confirm our understanding of the performance characteristics of the proposed tiling algorithms.

Algorithm 3.5 The latency-hiding model-parallel layer-k inference at thread q.

Input: Local input $X^{(k)}_{\{:,r_q\}_q}$ and weight matrix $W^{(k)}_{\{r_q \cup r_S, z_q \cup z_S\}_q}$.
Input separator parts $\sum_{r=0}^{p-1} X^{(k)}_{\{s_r,r_S\}_r}$
Output: Local output $X^{(k+1)}_{\{:,z_q\}_q}$ and separator part $X^{(k+1)}_{\{s_q,z_S\}_q}$.

1: **for** $r \leftarrow 0$ to $p-1$ **do**
2: $\quad X^{(k+1)}_{\{s_r,z_q\}_q} \leftarrow X^{(k)}_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_q\}_q}$ ▶ Without ReLU
3: $\quad X^{(k+1)}_{\{s_r,z_S\}_q} \leftarrow X^{(k)}_{\{s_r,r_q\}_q} W^{(k)}_{\{r_q,z_S\}_q}$ ▶ Without ReLU
4: **exec** barrier ▶ Executes only for $k \neq 0$
5: **for** $r \leftarrow 0$ to $p-1$ **do**
6: $\quad X^{(k+1)}_{\{s_r,z_q\}_q} \leftarrow f(X^{(k+1)}_{\{s_r,z_q\}_q} + X^{(k)}_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_q\}_q})$ ▶ Fused ReLU
7: $\quad X^{(k+1)}_{\{s_r,z_S\}_q} \leftarrow X^{(k+1)}_{\{s_r,z_S\}_q} + X^{(k)}_{\{s_r,r_S\}_r} W^{(k)}_{\{r_S,z_S\}_q}$ ▶ Without ReLU
8: **exec** barrier
9: $X^{(k+1)}_{\{s_q,z_S\}_q} \leftarrow f(\sum_{r=0}^{p-1} X^{(k+1)}_{\{s_q,z_S\}_r})$ ▶ Apply ReLU

In Section 3.4.2, we first compare the tiling model-parallel approach versus the data-parallel inference for different number of threads. Recall that the hybrid-parallel inference scales up the tiling model-parallel inference. By assuming $p = p_0 p_1$, it runs $p_0 = \lfloor p/p_1 \rfloor$ tiling model-parallel inferences where each such inference uses p_1 threads. The number p_1 is identified in the first set of experiments. In Section 3.4.3 contains experimental results for the hybrid-parallel inference versus the data-parallel inference.

3.4.1 Setup

We use two machines for experiments: an Ivy Bridge node consisting of two sockets each equipped with 10 cores, and a Cascade Lake node with two sockets each with 22 cores. Both machines have 32 KB of L1 data cache size per core. 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 machines run Linux with kernel version 3.10.0 on Ivy Bridge and 5.4.0 on Cascade Lake.

The performance of tiling inference exceeds that of the data-parallel algorithm only when both the input 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. We expect additional performance gains for tiling inference when a tile of intermediate inputs and the local neural network (due to partitioning) fit L2 cache. Despite these cache effects, the gains might still diminish if the separator size is too large. For the tiling methods, when the combined network does not fit L3 cache we take the maximum tile size for which thread-local tiles of $X^{(k)}$ remain in L3 cache. Otherwise, we find the maximum tile size for which both thread-local tiles of $X^{(k)}$ and the combined network fit

Threads	Separator	Tile	Time (in s)	
p	(in %)	size	Data-parallel	Tiling model-parallel (speedup)
2	11.79	11392	2.01	2.22 (0.91)
3	20.61	8067	1.32	1.63 (0.81)
4	16.25	6628	1.02	1.32 (0.77)
5	34.79	4575	0.81	1.55 (0.52)
6	26.67	4278	0.68	1.09 (0.62)
7	40.70	3304	0.58	1.15 (0.50)
8	21.25	3160	0.53	0.79 (0.67)
9	41.81	2601	0.46	0.99 (0.46)
10	42.71	2500	0.42	0.83 (0.51)
20	46.16	1340	0.22	0.53 (0.53)

Table 3.4 – The tiling model-parallel inference results over the first 5 layers of the 1k-neuron network 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.

in L3 cache.

We want to find a tile size b_{size} for which the batches of the intermediate feature matrices $X^{(k)}_{\{b_m,:\}}$ fit in cache between the layers of inference. The size of the intermediate feature matrices $X^{(k)}$ is related to the number of nonzeros which is unknown beforehand, as it changes between layers depending on the computation. We use a formula combining the nonzero density of the first matrix and the size of the separators to find the tile size. We ensure that the tile size $b_{size} > 0$ is divisible by the number of threads used for the tiling model-parallel inference.

3.4.2 The tiling model-parallel inference results

We first test the tiling model-parallel inference and the data-parallel inference on the first five layers of the 1k-, 16k, and 64k-neuron NNs for different number of threads on the Ivy Bridge node. In all tables in this section, we show the run times of the data-parallel inference and the model-parallel inference and the tile size. For Ivy Bridge results, we also show the separator size as a percentage of all column sizes. Table 3.4 shows the tiling model-parallel inference versus the data-parallel baseline results for the 1k-neuron NN on Ivy Bridge node. For this NN, we observe a slowdown of the tiling model-parallel method. This is the case as each of the intermediate $X^{(k)}$ not only fit in cache, but their size also decreases as k grows (see Table 3.2). Therefore, the data-parallel inference effectively performs tiling without the overhead of the algorithmic tiling. We omit the 4k-neuron NN results for which the matrices also fit in cache and the same effect is observed. We conduct the rest of the experiments on the 16k- and 64k-neuron NNs only.

Figure 3.2 shows the run times of the tiling model-parallel and data-parallel inference over the first 5 layers of 16k- and 64k-neuron NN for different number of threads on Ivy Bridge. It summarizes the results in Tables 3.5 and 3.6 with the 16k- and 64k-neuron NN

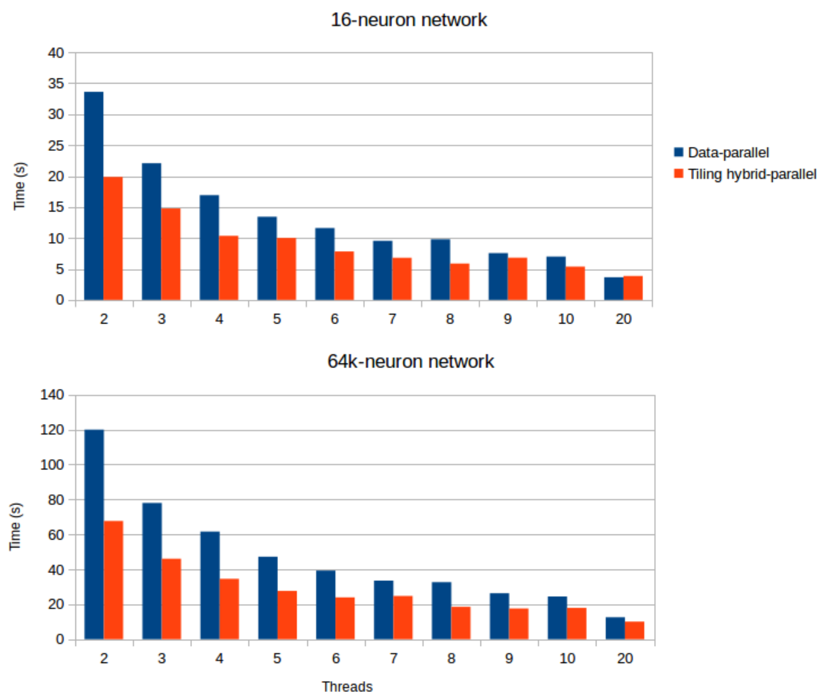

Figure 3.2 – Plot of the run time (in seconds) of data-parallel inference and the tiling model-parallel inference using the first 5 layers of the 16k- and 64k-neuron networks for $\{2, \ldots, 10, 20\}$ threads on Ivy Bridge.

results, respectively. For the 16k-neuron NN we observe up to 1.69 speedup, while for the 64k-neuron NN we observe up to 1.78 speedup against the data-parallel inference. The 64k NN experiment achieves slightly higher speedups as its input feature matrix is larger and therefore tiling saves more data movement. For both NNs, the tiling method achieves the best speedup for 2, 4, and 8 threads on Ivy Bridge. As expected, the separator size grows with p in all experiments so far as the complexity of the partitioning problem increases. Figure 3.3 shows the results for the same experiment repeated on the Cascade Lake node; we omit the separator size as we use the same partitioning as previously. The full experimental results for the 16k- and 64k-neuron NNs are given in Tables 3.7 and 3.8. Here, we achieve higher speedups, with up to 2 times speedup for the 64k-neuron NN, as the combined cache size of Cascade Lake is higher.

To summarize the findings in this section, we note that the efficiency of the tiling model-parallel algorithm decreases with the number of threads. This is so as the parallel overhead grows not only in terms of p, but also in terms of the number of external rows and columns which grows as well with the increasing number p of threads. Therefore, we find with these experiments the number of threads for which the model-parallel inference is most efficient. This number is limited as the problem size is constant while the separator size, which determines the overhead of the tiling model-parallel algorithm, grows with p.

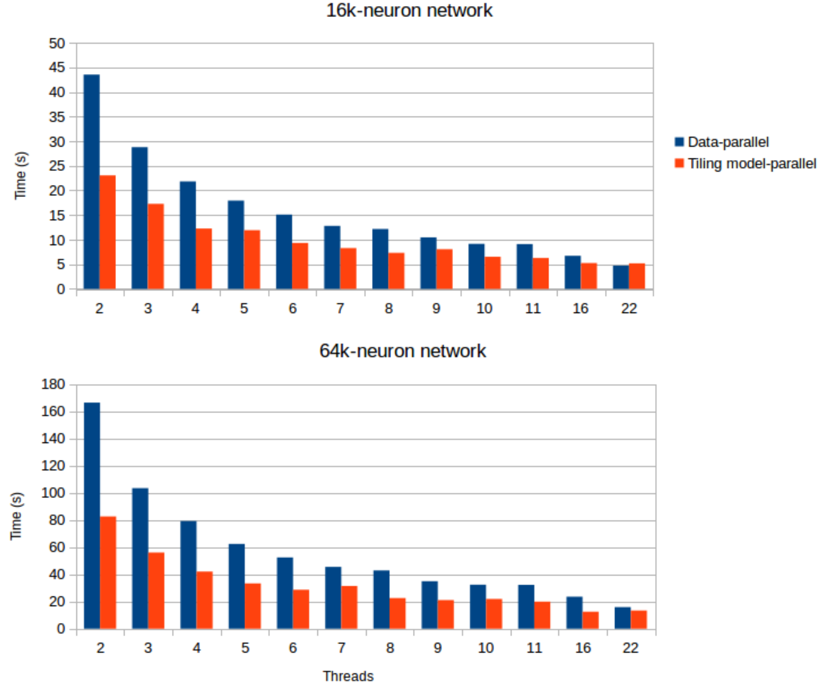

Figure 3.3 – Plot of the run time (in seconds) of data-parallel inference and the tiling model-parallel inference using the first 5 layers of the 16k- and 64k-neuron networks for $\{2, \ldots, 11, 16, 22\}$ threads on Cascade Lake.

Threads p	Separator (in %)	Tile size	Time (in s) Data-parallel	Tiling model-parallel (speedup)
2	0.80	158	33.62	19.86 (1.69)
3	5.37	117	22.07	14.77 (1.49)
4	4.89	116	16.92	10.35 (1.63)
5	6.02	95	13.44	10.01 (1.34)
6	4.24	114	11.60	7.82 (1.48)
7	7.34	63	9.55	6.80 (1.40)
8	7.97	72	9.78	5.87 (1.67)
9	6.77	81	7.56	6.81 (1.11)
10	8.27	90	7.00	5.37 (1.30)
20	16.74	180	3.66	3.87 (0.95)

Table 3.5 – The tiling model-parallel inference results over the first 5 layers of the 16k-neuron network 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.

Threads	Separator	Tile	Time (in s)	
p	(in %)	size	Data-parallel	Tiling model-parallel (speedup)
2	2.08	220	119.90	67.64 (1.77)
3	2.41	210	77.99	46.05 (1.69)
4	4.55	200	61.57	34.57 (1.78)
5	1.69	200	47.22	27.65 (1.71)
6	2.53	180	39.24	23.91 (1.64)
7	7.79	140	33.53	24.75 (1.35)
8	3.68	160	32.64	18.53 (1.76)
9	3.26	180	26.28	17.58 (1.49)
10	5.23	100	24.45	17.93 (1.36)
20	6.17	200	12.58	10.06 (1.25)

Table 3.6 – The tiling model-parallel inference results over the first 5 layers of the 64k-neuron network 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.

Threads	Tile	Time (in s)	
p	size	Data-parallel	Tiling model-parallel (speedup)
2	158	43.53	23.05 (1.89)
3	117	28.80	17.28 (1.67)
4	116	21.84	12.27 (1.78)
5	95	17.94	11.93 (1.50)
6	114	15.10	9.33 (1.62)
7	63	12.80	8.29 (1.54)
8	72	12.17	7.32 (1.66)
9	81	10.47	8.07 (1.30)
10	90	9.16	6.52 (1.40)
11	99	9.10	6.28 (1.45)
12	108	8.38	6.84 (1.23)
16	144	6.73	5.26 (1.28)
20	180	4.96	5.09 (0.97)
22	198	4.73	5.20 (0.91)

Table 3.7 – The tiling model-parallel inference results over the first 5 layers of the 16k-neuron network on Cascade Lake. The separator is given in percentage of the total input and output column size. Speedups are relative to the data-parallel baseline.

Threads	Tile	Time (in s)	
p	size	Data-parallel	Tiling model-parallel (speedup)
2	220	166.34	82.58 (2.01)
3	210	103.39	56.03 (1.85)
4	200	79.19	42.03 (1.88)
5	200	62.33	33.29 (1.87)
6	180	52.40	28.63 (1.83)
7	140	45.48	31.42 (1.45)
8	160	42.87	22.54 (1.90)
9	180	34.88	21.04 (1.66)
10	100	32.36	21.86 (1.48)
11	110	32.24	19.90 (1.62)
12	120	27.66	17.53 (1.58)
16	160	23.55	12.50 (1.88)
20	200	16.69	12.34 (1.35)
22	220	15.85	13.41 (1.18)

Table 3.8 – The tiling model-parallel inference results over the first 5 layers of the 64k-neuron network on Cascade Lake. The separator is given in percentage of the total input and output column size. Speedups are relative to the data-parallel baseline.

3.4.3 The tiling hybrid-parallel inference results

We are now able to proceed with the tiling hybrid-parallel inference. We select suitable p_1 equal to the best performing p from the earlier tiling model-parallel inference experiments, while using the remaining threads to scale up and invoke p_0 such tiling inferences over p_1 collaborating threads.

Table 3.9 shows the results for the tiling hybrid-parallel inference using the 5-layer 16k- and 64k-neuron network on the Ivy Bridge node. Here, we note that we use $p_1 = 5$ for the 64k-neuron network as we prefer p_1 values which are also factors of p. The results in the table confirm that using multiple tiling model-parallel inferences maintains the speedups versus the data-parallel baseline that we observed in Section 3.4.2 with $p = p_1$ threads. The tiling hybrid-parallel is the fastest inference on the first five layers using the full Ivy Bridge machine, 41% faster versus the data-parallel method for the 16k-neuron network, and 50% faster on the 64k-neuron network. From the same table, we additionally conclude that using hyperthreads benefits both the data-parallel and tiling methods, presumably possible in the latter case due to the almost synchronization-free method employed between the (hyper)threads.

Table 3.10 show the tiling hybrid-parallel and data-parallel inference results for the first five layers of the 16k- and 64k-neuron NN on the Cascade Lake node. Here, some of the experiments use less than the available 22 cores per socket due to the indivisibility of p by the other p_1 values with good performance such as 4 and 8. The tiling hybrid-parallel inference is 101% and 94% faster compared to the data-parallel method for the 16k- and 64k-neuron networks, respectively, using all 22 cores with hyperthreads in a $22 \cdot 2$

NN (defined by number of neurons)	p	p_0	p_1	Time (in s) Data-parallel	Time (in s) Tiling hybrid-parallel (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 3.9 – The tiling hybrid-parallel inference results over the first 5 layers of 16k-, and 64k-neuron networks on Ivy Bridge.

NN (defined by number of neurons)	p	p_0	p_1	Time (in s) Data-parallel	Time (in s) Tiling hybrid-parallel (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 3.10 – The tiling hybrid-parallel inference results over the first 5 layers of 16k-, and 64k-neuron networks on Cascade Lake.

configuration. The results for both experiments confirm the same expected behavior, and thus show that the proposed tiling hybrid-parallel inference is better than the state-of-the-art method of data-parallel inference across different architectures.

3.5 Concluding remarks

In this chapter, we consider different parallelizations of the sparse inference. We analyze the data-parallel inference, which is the standard approach based on partitioning the input. We propose efficient tiling model-parallel and tiling hybrid-parallel inference algorithms, which are based on model-parallelism. In our case, this refers to partitioning of the nonzeros of the neural network to reduce the data movement cost.

To effectively partition the network for the tiling variants, we define a matrix called the staircase matrix. This matrix uses a block structure of all weight matrices in a way which accurately captures dependencies and communication between consecutive layers. We propose using a hypergraph model to model the staircase matrix and to partition it to reduce the parallel overheads in the model-parallel algorithms. We also propose a core kernel, called *fusedSpGEMM*, which works on matrices split in a way specific to the tiling algorithms using model-parallelism.

The tiling-based algorithms perform tiling through the layers to increase cache reuse and use weak synchronization to hide potential load imbalance and reduce synchronization costs. The tiling hybrid-parallel algorithm is designed to scale the tiling model-parallel one. It offers best results for inference for 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 demonstrate that this approach works across architectures.

Chapter 4

Conclusions

This thesis designs high performance algorithms for computational kernels on tensors and sparse matrices that arise in data analysis and machine learning applications. The algorithms and techniques focus on and exploit the dependencies in the tensor and sparse matrix computations. Techniques from the high-performance computing and combinatorial scientific computing research areas are adapted to increase the data locality. These techniques include blocking, tiling, space-filling curves, and hypergraph partitioning which enable efficient algorithms for dense tensor and sparse neural network computations. Parallel algorithms are designed by investigating different types of data partitioning and by carefully investigating the parallel overheads in work, data movement within and between CPU sockets, synchronization, and storage. All algorithms are analyzed theoretically using a machine and a cost model to quantify the parallel overheads. Our analyses show that the proposed parallel algorithms have reduced overheads of data movement, thanks to the aforementioned techniques. The quantification of overheads is not only important for designing good algorithms, but also to characterize the limitations different algorithms may incur for different input parameters. We use auto-tuning to initialize parameters such as the block size and the tile size in our codes. All algorithms are implemented and carefully tested on recent machines. The experiments on shared-memory machines confirm that the proposed methods work in practice.

4.1 Summary

In the following, we give summaries of Chapters 2 and 3.

4.1.1 Summary of Chapter 2

Chapter 2 proposes a mode-oblivious data structure for dense tensors, and demonstrates its advantages on one of the most bandwidth-bound tensor operations, the tensor–vector multiplication (TVM). A mode-oblivious algorithm is defined in the thesis as one which has low relative standard deviation between its measured performances across different tensor modes it may be applied to. The proposed mode-oblivious layout, which we called

$\rho_Z\rho_\pi$-layout, stores dense blocks of a tensor, whose elements are ordered lexicographically, ordered according to the Morton order. While storing dense blocks using the standard ordering ensures the applicability of BLAS kernels, storing the blocks using Morton order induces mode-obliviousness by recursively blocking all tensor modes. Therefore, when the proposed $\rho_Z\rho_\pi$-block TVM algorithm performs a TVM with a tensor stored using $\rho_Z\rho_\pi$-layout, the caching occurs on both the input vector and the output tensor. This is so as they are accessed with temporal locality for each of the d modes. The results not only demonstrate superior performance of $\rho_Z\rho_\pi$-block TVM over the state-of-the-art variants by up to 18%, but additionally show that it induces a 71% less sample standard deviation for the TVM across d modes, where d varies from 2 to 10. For the kernels, we use two main BLAS2 libraries as subroutine of the aforementioned algorithms, Intel MKL and LIBXSMM. Results show that the combination of both gives the best results.

The data structure and algorithms for TVM are used as a building block for the higher-order power method (HOPM). The core of HOPM is a tensor-times-a-sequence-of-vectors kernel which computes successive $TVMs$. Thus, in HOPM the cache effects of blocking are magnified, resulting in even more pronounced performance gains for blocked layouts. The experimental results demonstrated up to 38% higher performance with respect to a standard implementation based on the state of the art TVM algorithms. For the HOPM, the blocking itself causes the largest increase in spatial locality, while for the TVM, the spatial locality is mainly induced by the Morton order of the blocks. We note that none of our improvements can be achieved on the level of BLAS libraries, since we require a change in tensor layout.

After developing the storage and sequential algorithms and experimenting with them for parameter tuning, the chapter turns attention to the parallel shared-memory TVM algorithms. By building upon the developed sequential TVM kernels, a number of parallel algorithms are proposed. These algorithms are analyzed for work, memory, intra- and inter-socket data movement, the number of barriers, and mode obliviousness. Two best variants are identified and implemented using OpenMP. These algorithms, called 0-sync and q-sync, deliver close to peak performance on four different systems, with 1, 2, 4, and 8 sockets, and surpass an optimized baseline algorithm based on the state-of-the-art.

4.1.2 Summary of Chapter 3

Chapter 3 tackles the problem of sparse inference as originally proposed by the IEEE HPEC Graph Challenge 2019 [33]. The sparse inference is the task of classifying data items using a sparse neural network. The sparse neural network is represented as a set of sparse matrices known as the weight matrices, one for each layer of the network. The data items to be classified form the input feature matrix, which is also sparse in the problem dataset. This data items are passed through the layers of the neural network to obtain a classification at the end. The key operation in passing the data items is formulated as a matrix computation kernel, the sparse matrix times sparse matrix multiplication (SpGEMM). As both the neural network activation function application and the bias addition may be implemented within the SpGEMM kernel and are less computational intensive, the sparse inference may be effectively seen as an SpGEMM operation repeated

for each layer of the neural network.

The chapter develops our approach gradually, by first presenting the two different parallelization approaches, the data-parallel inference and the model-parallel inference. In a typical use-case scenario, the sparse neural network is trained once, and used for classification of arbitrarily many data items. Thus, while for the data-parallel inference we partition the input feature matrix using block row-wise partitioning without paying attention to the sparsity, for the model-parallel inference we attempt to obtain an improved partitioning of the neural network. We propose a view in which the weight matrices in different layers are stitched together to form a staircase matrix, representing the whole neural network as a single sparse matrix. The structure of this sparse matrix is exploited by representing it as a hypergraph and then using a hypergraph partitioning software as a black-box. In the assumed use-case scenario, this happens during a stage before inference starts where the neural networks are trained to learn the weights and the connections. The cost of hypergraph partitioning is therefore accounted for. We used a two-dimensional nonzero partitioning of the weight matrices of all layers obtained by partitioning the staircase matrix. This induces, in our case, a partitioning on the input and output matrices of an SpGEMM operation, where the successive SpGEMMs have partitions aligned on the common matrix. High-performance SpGEMM-like operations may use tiling. This technique requires pausing an iteration in order to compute a full inference of only a subset of data items such that the intermediate results are cached. We use it to reduce data movement in the model-parallel method and thus propose the tiling model-parallel algorithm, which we also combine with mechanisms to lower synchronization costs.

We implement shared-memory algorithms for the data-parallel inference and tiling model-parallel inference. In the data-parallel inference, the threads execute a series of SpGEMM calls in an embarrassingly parallel fashion. In the tiling model-parallel inference, the threads complete a model-parallel inference on a block of data items before moving on to another block within a tiling scenario. Tiling allows the intermediate feature matrix results to stay in cache throughout the inference. Finally, the tiling hybrid-parallel inference is a combination of both parallelization methods, and may be thought of as running multiple tiling model-parallel inferences each on a separate row-wise block of the input feature matrix and using different threads. Assuming $p = p_0 p_1$ threads, we first experiment on the tiling model-parallel inference to find the most efficient number of threads to partition the network into p_1 parts. We then execute hybrid-parallel inference using p_0 such tiling model-parallel inferences where each uses p_1 threads. Experimental results show that the tiling hybrid-parallel algorithm achieves x2 speedup against the state-of-the-art data-parallel algorithm running on the same number of threads. We also run experiments using hyperthreads which shows that our implementation benefits from it.

4.2 Future work

The two computational problems addressed in this thesis have many exciting prospects. These can be addressed as short-term and long-term goals. Below, we summarize these goals for the two problems separately.

4.2.1 Tensor computations

We believe the mode-oblivious storage is general enough to support the tensor–tensor (*TTM*) and tensor–matrix multiplications (*TMM*). We plan to implement these algorithms while taking advantage of the proposed tensor lay schemes. In particular, blocked algorithms should be designed for the *TMM* products with tall-skinny matrices, which are heavily used in tensor decomposition algorithms. In these cases, we remain in the hard-to-optimize bandwidth-bound regime. If, just as with a *TVM*, the input matrices are skinny enough to fit in cache, the oblivious behavior induced by the Morton order and exploited by our $\rho_Z\rho_\pi$-layout will magnify cache reuse and thus boost performance further. Considering general compute-bound *TMM* and *TTM* products, the use of Morton-ordered blocks is orthogonal to traditional BLAS3 optimizations and will boost cache reuse further [41].

The use of the Hilbert curve instead of the Morton order, even though computationally more expensive to use, will likely even further increase the cache reuse of the bandwidth-bound *TVM* and tall-skinny *TMM* products. The proposed tensor layout and processing method is orthogonal to most parallelization strategies; integration into such parallel schemes [6, 38] is another logical step.

Other future work includes the auto-tuning of non-square block sizes; maintaining a square block is restrictive for the number of choices that may fit in a targeted cache level since the block size grows exponentially with d. Preliminary benchmarks with choosing non-square block sizes indeed showed that the $\rho_Z\rho_\pi$-layout achieves higher *TVM* performance while retaining mode-obliviousness. Other parameters that could be considered for auto-tuning include software prefetch distances and SIMD sizes.

As a long term goal, we plan to extend the proposed algorithms and layout for distributed memory systems. All proposed *TVM* variants should work well, after modifying them to use explicit broadcasts of the input vector and/or explicit reductions on parts of the output tensor. While additional buffer spaces may be required, we expect that the other shared-memory cost analyses will transfer to the distributed-memory case. Additionally, many of the current parallel machines have nodes with accelerators such as GPUs. Nine out of top ten supercomputers on the June 2020 TOP500 list [58] have accelerators, in seven of them these are Nvidia GPUs. Thus, exploiting these accelerators is a must both for better energy use and high performance. Thus, first a distributed-memory hybrid MPI+OpenMP implementations for the *TTM*, *TMM*, and *TVM* will be developed, based on the shared-memory codes developed in the thesis. Then, relevant parts of the codes will be offloaded to the accelerators for the highest performance. How best split the task to multiple GPUs and CPUs available at a single node is a question which we have to elaborate.

4.2.2 Sparse networks

The most immediate work is to efficiently apply the proposed tiling hybrid-parallel algorithm, potentially with the mentioned optimizations, on batches of neural network layers successively. Such blocking approach allows storing in cache not only the tile of intermediate results, but also the weight matrices. This will confirm the applicability and benefits of the proposed approach on a broader set of neural network inference tasks, in particular, the deepest neural networks of the Graph Challenge dataset.

While the fine-grain hypergraph model serves our purposes, other partitioning approaches could also be used. In particular, we want to experiment with the medium grain approach [51] as it generally outperforms the fine-grain model for reducing the communication volume (otherwise known as the connectivity-1 metric). The partitioner method should be able to handle the cut-net and multi-constraint load balancing requirement. 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 such bounds were achieved in earlier work on sparse matrix–vector multiplication [72]. Allowing permutations between layers, which can also be modeled via hypergraphs [59], may allow further reduction of the separator size. The constraints of the partitioning problem also resemble the symmetric partitioning problem for the fine-grain model [60]. We plan to investigate this venue for further optimizing the partitions.

As this thesis demonstrates, adapting the right storage is of crucial importance for high-performance kernels. Thus, applicability of other storages for sparse matrices to the problem of sparse inference should be considered. Optimizations such as the use of algorithms for sparse matrix–dense matrix [1] multiplication may be used when the compressed density reaches 100% during inference as shown in [44].

Many of the neural network applications, including inference, approach the user or the hand-held devices such as smartphones, abiding to the edge computing paradigm and meeting its requirements. Making sense of the world around us by the help of hand held devices' capabilities is the current trend of the technology. Storing the neural networks and carrying out the inference on such devices are very exciting topics that we want to address. While we want to store large and deep models for accuracy, the devices at the edge have smaller memory. As a long-term goal, we plan to investigate methods to compress neural networks at hand, and develop methods to carry out inference efficiently with such compressed networks.

Bibliography

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

[2] S. Alford, R. Robinett, L. Milechin, and J. Kepner. Pruned and structurally sparse neural networks. *CoRR*, abs/1810.00299, 2018.

[3] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 25(6), 12 2002.

[4] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB Tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, December 2006.

[5] B. W. Bader, T. G. Kolda, et al. Matlab tensor toolbox version 2.6. http://www.sandia.gov/~tgkolda/TensorToolbox/, February 2015. visited on 01-30-2019.

[6] G. Ballard, N. Knight, and K. Rouse. Communication lower bounds for matricized tensor times Khatri-Rao product. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 557–567. IEEE, 2018.

[7] C. Bassoy. Design of a high-performance tensor-vector multiplication with BLAS. In J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M.A. Sloot, editors, *Computational Science – ICCS 2019*, pages 32–45, Cham, 2019. Springer International Publishing.

[8] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[9] 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, 2019. IEEE.

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

[11] 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.

[12] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *IPDPS*, page 118, 2001.

[13] Ü. V. Çatalyürek and C. Aykanat. Hypergraph partitioning. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 871–881. Springer, 2011.

[14] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM Journal on Scientific Computing*, 32(2):656–683, 2010.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 3rd edition, 2009.

[16] 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, 2019. IEEE.

[17] L. De Lathauwer, P. Comon, B. De Moor, and J. Vandewalle. Higher-order power method–Application in independent component analysis. In *Proceedings NOLTA'95*, pages 91–96, Las Vegas, USA, 1995.

[18] L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[19] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1):18–32, March 1988.

[20] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.

[21] 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, 2019. IEEE.

[22] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. Association for Computing Machinery.

[23] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[24] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[25] F. Gustavson. Finding the block lower triangular form of a sparse matrix. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 275–289. Academic Press, 1976.

[26] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

[27] K. Hayashi, G. Ballard, Y. Jiang, and M. J. Tobia. Shared-memory parallelization of MTTKRP for dense tensors. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 393–394, New York, NY, USA, 2018. ACM.

[28] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[29] M. Hidayetoglu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W. W. Hwu. At-scale sparse deep neural network inference with efficient GPU implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2020 (to appear). IEEE.

[30] Intel Math Kernel Library Reference Manual. `https://software.intel.com/en-us/articles/mkl-reference-manual`. visited on 30-01-2019.

[31] O. Kaya and B. Uçar. Parallel candecomp/parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018.

[32] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, A. Reuther, R. Robinett, and S. Samsi. Graphchallenge.org sparse deep neural network performance. arXiv e-prints, arXiv:2004.01181, 2020.

[33] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi. Sparse deep neural network Graph Challenge. arXiv e-prints, arXiv:1909.05631, 2019.

[34] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[35] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, September 2009.

[36] Y. LeCun, C. Cortes, and C. J. C. Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

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

[38] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 76:1–76:12. IEEE, 2015.

[39] J. Li, J. Sun, and R. Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'18, pages 19:1–19:15, New York, NY, USA, 2018. ACM.

[40] D. Lin and T. Huang. A novel inference algorithm for large sparse neural network using task graph parallelism. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2020 (to appear). IEEE.

[41] K. P. Lorton and D. S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Comput. Archit. News*, 35(4):6–12, September 2007.

[42] D. Matthews. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

[43] 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.

[44] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud. Studying the effects of hashing of sparse deep neural networks on data and model parallelisms. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2020 (to appear). IEEE.

[45] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[46] 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*, ICPP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[47] numactl(8) - linux man page. `https://linux.die.net/man/8/numactl`. visited on 28-09-2020.

[48] F. Pawłowski, R. H. Bisseling, B. Uçar, and A. N. Yzelman. Combinatorial tiling for sparse neural networks. In *proc. 2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2020 (to appear). IEEE.

[49] F. Pawłowski, B. Uçar, and A. N. Yzelman. High performance tensor–vector multiplication on shared-memory systems. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, volume 12043, pages 38–48, Cham, 2020. Springer International Publishing.

[50] F. Pawłowski, B. Uçar, and A. N. Yzelman. A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations. *Journal of Computational Science*, 33:34–44, 2019.

[51] D. M. Pelt and R. H. Bisseling. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *IEEE 28th International Parallel and Distributed Processing Symposium*, pages 529–539, 2014.

[52] A. H. Phan, P. Tichavský, and A. Cichocki. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing*, 61(19):4834–4846, Oct 2013.

[53] S. Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[54] S. Smith and G. Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 ’15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[55] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[56] P. Springer and P. Bientinesi. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Transactions on Mathematical Software*, 44(3):28:1–28:29, 2018.

[57] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1-2):109–130, 1998.

[58] Top500 list 06/2020. https://www.top500.org/lists/top500/2020/06/. visited on 28-09-2020.

[59] B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 29(4):1683–1709, 2007.

[60] B. Uçar and C. Aykanat. Minimizing communication cost in fine-grain partitioning of sparse matrices. In A. Yazici and C. Şener, editors, *Computer and Information Sciences - ISCIS 2003*, volume 2869 of *Lecture Notes in Computer Science*, pages 926–933. Springer Berlin / Heidelberg, 2003.