

Sparse Computations in Deep Learning Inference

Ioanna Tasou^{†*}, Panagiotis Mpakos^{†*}, Angelos Vlachos[‡], Dionysios Adamopoulos[‡], Georgios Giannakopoulos[‡], Konstantinos Katsikopoulos[‡], Ioannis Karaparis[‡], Maria Lazou[‡], Spyridon Loukovitis[‡], Areti Mei[‡], Anastasia Pouloupoulou[‡], Angeliki Dimitriou[‡], Giorgos Filandrianos[‡], Dimitrios Galanopoulos[‡], Vasileios Karampinis[‡], Ilias Mitsouras[‡], Nikolaos Spanos[‡], Petros Anastasiadis[‡], Ioannis Doudalis[‡], Konstantinos Nikas[‡], George Retsinas[§], Paraskevi Tzouveli[‡], Christina Giannoula[¶], Nectarios Koziris[‡], Nikela Papadopoulou[#], Giorgos Stamou[‡], Athanasios Voulodimos[‡], Georgios Goumas^{†*}

[†] *Computing Systems Laboratory (CSLab), National Technical University of Athens, Greece*

[‡] *Artificial Intelligence and Learning Systems Laboratory (AILS), National Technical University of Athens, Greece*

[§] *Intelligent Robotics and Automation Laboratory (IRAL), National Technical University of Athens, Greece*

[¶] *Max Planck Institute for Software Systems (MPI-SWS), Germany*

[#] *School of Computing Science, University of Glasgow, United Kingdom*

Abstract

The computational demands of modern Deep Neural Networks (DNNs) are immense and constantly growing. While training costs usually capture public attention, inference demands are also contributing in significant computational, energy and environmental footprints. Sparsity stands out as a critical mechanism for drastically reducing these resource demands. However, its potential remains largely untapped and is not yet fully incorporated in production AI systems. To bridge this gap, this work provides the necessary knowledge and insights for performance engineers keen to get involved in deep learning inference optimization. In particular, in this work we: a) discuss the various forms of sparsity that can be utilized in DNN inference, b) explain how the original dense computations translate to sparse kernels, c) provide an extensive bibliographic review of the state-of-the-art in the implementation of these kernels for CPUs and GPUs, d) discuss the availability of sparse datasets in support of sparsity-related research and development, e) explore the current software tools and frameworks that provide robust sparsity support, and f) present evaluation results of different implementations of the key SpMM and SDDMM kernels on CPU and GPU platforms. Ultimately, this paper aims to serve as a resource for performance engineers seeking to develop and deploy highly efficient sparse deep learning models in productions.

1 Introduction

Artificial Intelligence (AI) is revolutionizing modern life, with advances in computer vision and Natural Language Processing (NLP) that enable transformative applications such as autonomous driving, intelligent virtual assistants, automated translation systems, precision medicine, financial anomaly detection, and generative models that produce text, images, and other creative content. These developments demonstrate that AI can considerably augment human capabilities, enhance decision-making, and reshape socio-technical systems across diverse domains. All these technological breakthroughs are based on the rapid evolution of Deep Neural Networks (DNNs), which in turn rely on the architectural enhancement of the networks themselves, the availability of huge amounts of data, and the support of large-scale computational infrastructures to host the unprecedented amounts of computations involved in the training and operation of these models. As of Q4 2025, the largest large language models (LLMs) publicly known (e.g., Qwen3-Max, Ling-1t, Samba-1, etc),

*Corresponding Authors: Ioanna Tasou, Panagiotis Mpakos, Georgios Goumas {itasou,pmpakos,goumas}@cslab.ece.ntua.gr

appear to be approaching or exceeding one trillion parameters, marking a transition beyond the hundreds-of-billions scale. The computational needs to train such models are immense and exponentially increasing [29]. Meta reported that Llama 4 models are trained on a cluster of 100,000 H100 GPUs [34]. Although the computational needs for training capture more publicity, the operation of these models over time in inference tasks is expected to require significantly higher computational capabilities due to its repetitive nature [116]. In terms of environmental footprint, recent studies report that the operation of AI models already consumes electricity equal to that of 17,000 US homes [111].

Sparsification comes as a natural mechanism to tame this colossal demand for computing power. Sparse data structures and computational kernels have been resident in computing for several decades, as they precisely represent data and their relations in scientific computing and graph analytics, to name the most popular examples [8]. In the context of DNNs, several forms of sparsity can be utilized to avoid a large percentage of the operations, therefore speeding up execution. Researchers and practitioners enforce sparsity by pruning layers or neurons from the networks, exploit the outcome of activation functions that lie at the end of each DNN layer and frequently output zeros, or just implement graph neural networks that are inherently sparse. Finally, other highly popular and effective mechanisms such as the Mixture of Experts (MoE) [107; 217; 55] or even quantization [71] can be considered sparsification methods in a wider sense.

In this work, we argue that while sparsification is already a known optimization concept, its potential to further reduce computational requirements in production systems remains largely untapped. We base this claim on three reasons: First, the industry has entered a competition - the ‘AI race’ - to deploy accurate AI systems, leaving aside at this stage the computational concern. This, however, will come forward soon to make the operational model sustainable [235] and profitable. Second, the technological field around sparsity is quite complex, since there exist several interacting potential sources of sparsity. Consequently, the performance gains from sparsity are not straightforward to quantify or guarantee, as they involve complex trade-offs between computational efficiency and problem characteristics. Third, the complexity of the field has, for the moment, limited the involvement of performance engineers in the game, with current solutions in several cases being provided mainly by domain experts. We strongly believe that deeper involvement of performance engineers in sparse DNN optimization will provide much more efficient solutions.

To that end, this survey aims to explore and enable the incorporation of sparsity mechanisms in production AI systems. It is written with performance engineers in mind with limited or no knowledge in DNNs. We find an interesting analogy from the past: sparse computations and in particular, sparse matrix-vector multiplication are ubiquitous in scientific computing, e.g., in the solution of large, sparse linear systems with iterative methods [210]. The involvement of performance engineers and their collaboration with domain experts has increased the understanding of the interaction of the overarching application, the computational kernels involved, and the target execution platform, ultimately leading to highly effective, production-quality systems that rely on sparse data and computations [10; 109; 91].

Since the field is quite complex and fast-evolving, in our initial attempt to capture it, we deliberately limit our focus: we exclude from our discussion the role of sparsity in accelerating training, and discuss only the aspects of training that influence sparse inference. We also restrict our focus on mainstream, commodity hardware platforms like CPUs and GPUs, and leave for future work a large family of research efforts that optimize code for specialized accelerators (e.g., [69]) or even co-design accelerators for this purpose (e.g., [223; 68; 267; 195]).

We organize the survey around the following key questions:

1. In which forms does sparsity arise in DNN inference? (Section 3)
2. How do the original dense computations transform to sparse kernels? (Sections 2.1 and 3)
3. What is the current state-of-the-art in the implementation of these kernels? (Section 4)
4. What sparse datasets can be used to assist research and development in sparse DNNs? (Section 5)
5. What is the current software support to deploy production systems that rely on sparse DNNs? (Section 6)

6. What is the performance behavior of sparse kernels in modern CPUs and GPUs? (Section 7)

We would also like to make a note on the process that led to the production of this survey. We assembled a rather large and diverse team that included last-year undergraduate students, PhD students, postdoctoral researchers, and academics originating from the two disciplines of systems engineering and artificial intelligence. The process was interesting and constructive, revealing the existing gaps between scientists stemming from the two fields and revealing pathways that can bridge these gaps. At the end of the day, the process also served as a learning process for the team itself, and we hope that the community will find the survey useful and enjoyable, as we did with the process that led to it. We envision keeping this as a live document that will constantly capture the evolution in the field. In that sense, if you have comments, have spotted errors, or find that we have not included significant and relevant work, feel free to contact us.

2 Background

2.1 The basics of Deep Neural Networks (DNNs)

This section gives a brief introduction to how inference works within DNNs and to the characteristics of the operations that power them. This material was included to provide the necessary context for the sparsification discussion in Section 3; readers already familiar with neural networks may skip it.

DNNs are computational models composed of multiple **layers** that perform operations towards fulfilling a specific task, typically **classification** (e.g., classifying handwritten digits or objects in an image) or **regression** (e.g., to predict market prices). A typical DNN consists of an input layer, multiple hidden layers, and an output layer, stacked in a pipeline fashion. Conceptually, a neural network layer comprises a set of **neurons**, which perform linear transformations of the layer’s input. These operations depend on a set of meta-parameters, namely the **weights** and **bias**, which are learned during the network training process. The hidden layers of a DNN serve as intermediate steps that extract and combine features from their input. By stacking hidden layers, the network can build abstract representations, capturing complex patterns. Nevertheless, a neural network as described up to this point would be a chain of linear operations. In order to enable the model to express more complex, non-linear relationships, **activation functions** are introduced after the output of each layer, which are consequently non-linear transformations.

Figure 1 presents a typical representation of a simple hidden layer. Input $\mathbf{x} \in \mathbb{R}^m$ linearly transformed through n neurons with weights w_{ij} and biases b_i , and then passes through a non-linear activation function ϕ to produce the output $\mathbf{y} \in \mathbb{R}^n$. Note that, in general, inputs and outputs can be multidimensional, representing, e.g., 2-dimensional images with multiple **features**. In the case of images, features can represent, for example, their RGB content (three features, one per RGB component). In NLP, each **token** (i.e., word or subword) can have various grammar or syntactical features called **embeddings**. In these cases, inputs and output are presented with multi-dimensional **tensors**.

To accomplish their tasks, DNNs go through a training process during which the DNN calculates (learns) its weights and biases by minimizing a loss function, which measures how far its predictions are from the true targets. Using first-order gradient-based optimizers, such as *Stochastic Gradient Descent (SGD)*, the model updates its parameters iteratively: for each small batch of data, it computes the loss, then applies backpropagation to calculate the gradients of the loss with respect to each weight. These gradients are used to adjust the weights in the direction that reduces the loss, gradually improving the network’s performance. Note that sparsification before or during training [172; 57; 151; 48] is beyond the scope of this survey.

Figure 2 provides an overview of a simplified but typical DNN architecture. A DNN is composed of multiple hidden building blocks, where each block typically consists of a specific type of hidden layer followed by an activation function. The most common types of hidden layers and activation functions used in modern deep learning are described in the next paragraphs.

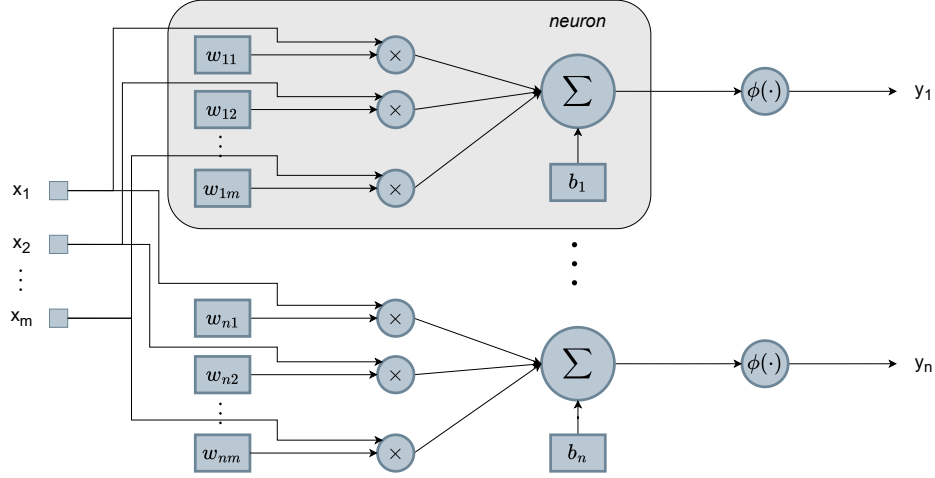


Figure 1: The structure of a basic neural network layer, followed by the activation function (ϕ).

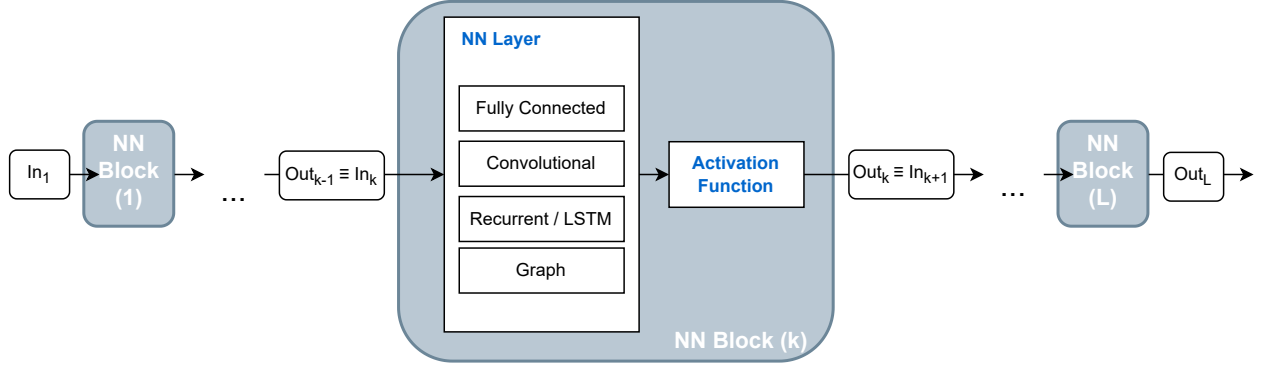


Figure 2: The generic architecture of a DNN.

2.1.1 Activation functions

Activation functions are essential components in neural networks, introducing non-linearity into the model that enables the learning of complex, hierarchical patterns beyond simple linear relationships. This nonlinearity is what fundamentally distinguishes dense neural networks from simple linear models. Without an activation function, stacking multiple linear layers would collapse into a single linear transformation, limiting the network's expressive power to that of basic linear regression. To break this limitation, non-linear activation functions are applied after each linear transformation. Historically, early neural networks primarily relied on the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (1)$$

and the hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2)$$

These functions are smooth and differentiable, which facilitates optimization using gradient-based methods such as backpropagation. However, they suffer from the vanishing gradient problem, where gradients become too small in saturated regions (e.g., very large positive or negative inputs), impeding learning in DNNs.

To address this issue, the Rectified Linear Unit (ReLU) [180] was introduced as a simple, highly effective alternative. It is defined as:

$$\text{ReLU}(x) = \max(0, x) \quad (3)$$

ReLU is non-linear, avoids saturation in the positive domain and allows gradients to flow efficiently through deep architecture. ReLU also promotes sparse activations by activating only neurons with positive input, reducing computation cost and improving generalization. Due to its simplicity and performance, ReLU has become the most frequent activation function in many modern DNNs.

Despite its advantages, ReLU has a limitation called "the dying ReLU problem", where neurons can become inactive and output zero for all inputs. This can happen when the neuron's weights are updated in such a way that it only receives negative inputs. If a neuron dies, it stops contributing to the learning process, which can negatively impact model performance, particularly in deeper networks. To address this, ReLU variants have been introduced, such as Leaky ReLU [162],

$$\text{LeakyReLU}(x) = \max(ax, x), \quad x > 0 \quad (4)$$

and Parametric ReLU (PReLU), where is a learned parameter [85], allowing small gradients for negative inputs. Leaky ReLU allows a small, nonzero gradient for negative inputs, helping to keep more neurons active.

Modern architectures have adopted more sophisticated activation functions. The Gaussian Error Linear Unit (GELU) [90], used in Transformer-based models like BERT and GPT, is defined as:

$$\text{GELU}(x) = x \cdot \Phi(x) \quad (5)$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution. Another popular choice is the Swish function [199], defined as:

$$\text{Swish}(x) = x \cdot \sigma(\beta x) \quad (6)$$

with β typically a learnable or fixed parameter. Swish and GELU offer smoother transitions and better empirical performance in deep and attention-based models. A widely used special case of Swish is the SiLU (Sigmoid Linear Unit), given by $\text{SiLU}(x) = x \cdot \sigma(x)$, which corresponds exactly to Swish with $\beta = 1$.

Both Swish and GELU provide smooth, non-monotonic behavior and have shown strong empirical performance in deep convolutional networks and modern attention-based architectures.

There are also activation functions that operate on vectors. For example, softmax is a function that converts an input vector \mathbf{x} into a probability distribution output vector:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}, \quad \text{for } i = 1, 2, \dots, K \quad (7)$$

where K is the number of classes, and is most commonly used at the output layer.

The choice of activation function is closely tied to both the model architecture and the specific learning task. In convolutional neural networks, ReLU and its variants remain dominant largely because they promote activation sparsity, setting a substantial portion of neuron outputs to zero. This sparsity not only reduces computational load but also enhances representational efficiency by encouraging the network to focus on the most salient features. Softmax, by contrast, is applied where normalized probability distributions are required, such as in attention mechanisms or the output layers of multi-class classifiers. Although hidden layers generally use nonlinear activations, output activations are selected according to the prediction objective: softmax for multi-class classification, sigmoid for binary decisions, and linear outputs for regression.

In modern attention-based and large-scale language models, smoother activation functions such as GELU and Swish are preferred because they allow more nuanced control over neuron responsiveness rather than inducing strict sparsity. While they do not produce the hard zeroing effect of ReLU, their soft gating behavior creates a form of graded sparsity, where small inputs are attenuated rather than eliminated. This smooth modulation supports stable gradient flow, improves optimization dynamics, and ultimately enhances performance in deep neural architectures.

2.1.2 Fully Connected layers

Fully Connected (FC) (dense/linear) layers are a main structural element of DNNs and the central mechanism in *Multi-Layer Perceptrons (MLPs)*. A fully connected layer consists of multiple neurons, where each neuron connects each feature of the input with each feature of the output, and thus performs a linear transformation, which is followed by a non-linear activation function. Stacking multiple such layers creates an MLP that comprises an input layer, one or more hidden layers, and an output layer. The mathematical formulation of a FC layer for a given input $\mathbf{x} \in \mathbb{R}^{d_{in}}$, which is typically a vector representing a single input sample with d_{in} features, which creates an output vector $\mathbf{y} \in \mathbb{R}^{d_{out}}$ with d_{out} features, is as follows:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (8)$$

where, $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ is the weight matrix calculated (learned) at training time, $\mathbf{b} \in \mathbb{R}^{d_{in}}$ is bias vector.

The formula above implies that the network operates on single input \mathbf{x} , and thus the dominant operation is *Matrix-Vector Multiplication*. However, in several cases the network applies inference in batches of input, i.e., $\mathbf{x} \in \mathbb{R}^{d_{in} \times B}$ where B is the batch size, which translates to *Matrix Multiplication*, which in the terminology of software libraries is commonly referenced as **GEMM** (GEneral Matrix Multiplication).

FC layers, though theoretically capable of modeling big data, become inefficient and impractical when applied to high-dimensional inputs such as images. For instance, a single RGB image of size $(224 \times 224 \times 3)$ contains over 150,000 input features; connecting each of these to a dense hidden layer would require millions of parameters, leading to severe risk of overfitting,¹ increased memory usage, and computationally expensive training. Furthermore, this architecture requires flattening the image into a one dimensional vector, which dilutes the inherent spatial locality, (i.e., spatial relationships between neighboring pixels, like edges, textures and local patterns), which is crucial for visual content understanding. These challenges highlight the need for architectures that maintain the spatial arrangement of pixels and leverage hierarchical feature extraction, from simple local patterns (e.g., edges, corners) to complex global structures (e.g., entire objects), for efficient visual understanding.

2.1.3 Convolutional layers

Convolutional layers are the basic building block of Convolutional Neural Networks (CNNs) and offer an efficient alternative that addresses the limitations mentioned above through two key principles: (i) local connectivity, where each neuron is connected only to a local patch of the input, and (ii) weight sharing, where the same filter is applied across different spatial locations. These properties make CNNs both translation-equivariant and highly parameter-efficient, particularly well-suited not only for visual tasks, but also for other structured data domains where spatial or local correlations are important.

The basic operation of a convolutional layer can be described as applying a sliding window over an array, where the window is defined by a learnable **filter** (or **kernel**). In image processing terms, kernels operate as ‘queries’ to the specific part of the image as to whether the image contains a specific edge or corner (in early layers of the DNN), or a specific object (e.g., a letter in later layers of the DNN). Layers typically contain multiple kernels (e.g., 64), each one trained to query a specific feature. To illustrate the application of a single kernel, consider a grayscale image $\mathbf{I} \in \mathbb{R}^{3 \times 3}$ and a kernel $\mathbf{K} \in \mathbb{R}^{2 \times 2}$:

$$\mathbf{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \quad (9)$$

With stride 1 and no padding, the resulting output $\mathbf{O} \in \mathbb{R}^{2 \times 2}$ is:

$$\mathbf{O} = \begin{bmatrix} K_{11}I_{11} + K_{12}I_{12} + K_{21}I_{21} + K_{22}I_{22} & K_{11}I_{12} + K_{12}I_{13} + K_{21}I_{22} + K_{22}I_{23} \\ K_{11}I_{21} + K_{12}I_{22} + K_{21}I_{31} + K_{22}I_{32} & K_{11}I_{22} + K_{12}I_{23} + K_{21}I_{32} + K_{22}I_{33} \end{bmatrix} \quad (10)$$

¹Overfitting happens when a neural network learns the training data too specifically leading to poor performance on new data.

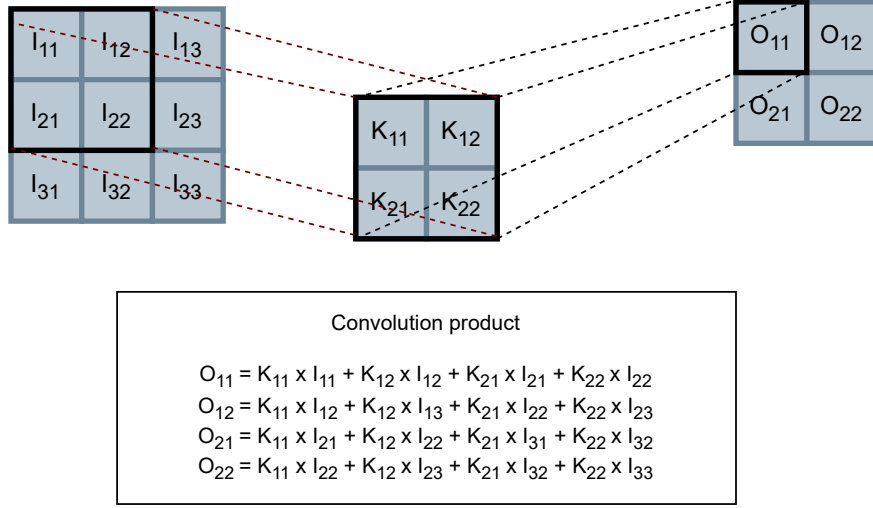


Figure 3: Convolution example: A 3×3 input I is convolved with a 2×2 kernel K using a sliding window to produce the 2×2 output O .

Each entry O_{ij} results from an element-wise product between the kernel and the corresponding 2×2 patch of the input, followed by summation, as shown in Figure 3.

To generalize this operation, consider an input (grayscale) image $I \in \mathbb{R}^{H \times W}$ and a kernel $K \in \mathbb{R}^{m \times n}$ with height m and width n . With stride 1 and no padding, the convolution produces an output $O \in \mathbb{R}^{(H-m+1) \times (W-n+1)}$. Each output entry corresponds to applying the kernel to a local $m \times n$ patch of the input, obtained by placing the *top-left corner* of the kernel at position (i, j) . Formally,

$$O(i, j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} I(i+u, j+v) K(u, v), \quad (11)$$

where the valid output indices are:

$$0 \leq i \leq H - m, \quad 0 \leq j \leq W - n.$$

In practice, several variations of this basic operation are widely used. Padding is applied to preserve spatial resolution when desired, strided convolutions downsample feature maps by skipping positions, and dilated convolutions expand the receptive field without increasing kernel size. These extensions follow the same underlying principles but are beyond the scope of this work.

The convolution operation, as described in Equation 11, involves nested loops and non-contiguous memory access, which is computationally inefficient on modern hardware. A common strategy followed by many deep learning libraries is to implement convolution as a single, large **GEMM** operation using the image-to-column (**im2col**) transformation. This technique converts for example the 2D grayscale input matrix (I) into a 2D matrix by taking every local receptive field (patch) that the kernel will operate on, stretching it into a single column vector, and stacking these columns side-by-side. The K kernel (filter) is also flattened into a row vector. The original convolution is then transformed into a matrix multiplication: $\text{Flattened}(\mathbf{O}) = \text{Flattened}(\mathbf{K}) \times \text{im2col}(\mathbf{I})$ as shown in Equation 12. This transformation maximizes data locality and allows the operation to leverage highly efficient **GEMM** routines.

$$\begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{21} & K_{22} \end{bmatrix} \times \begin{bmatrix} I_{11} & I_{12} & I_{21} & I_{22} \\ I_{12} & I_{13} & I_{22} & I_{23} \\ I_{21} & I_{22} & I_{31} & I_{32} \\ I_{22} & I_{23} & I_{32} & I_{33} \end{bmatrix} \quad (12)$$

Historically, CNNs gained early recognition with the introduction of LeNet-5 [132], which was designed for handwritten digit recognition. Their true breakthrough came with AlexNet [123], which achieved a significant performance leap on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [209], establishing CNNs as a cornerstone of modern computer vision. Subsequent architectures such as VGGNet [220], ResNet [86], and DenseNet [102] further advanced the field by introducing innovations like increased depth, residual connections, and dense connectivity—enhancing training stability and overall accuracy. Today, convolutional layers serve as the backbone for most computer vision systems, forming the basis for advanced tasks, such as object detection (e.g., in YOLO [200], Faster R-CNN [201]) and image segmentation (e.g., in U-Net [205], DeepLab [18]). While attention-based models like Vision Transformers (ViTs) [41] are emerging, CNNs remain fundamental due to their efficiency in spatial data processing.

2.1.4 Sequence modeling: Recurrent Neural Networks and Long Short-Term Memory networks

Sequence models address tasks where the order of inputs and the temporal relationships, i.e., position and time-based dependencies that exist between elements in a sequence, carry essential meaning. These requirements occur in areas such as speech recognition, language processing, and time series prediction. Initially, *Hidden Markov Models (HMMs)* [197] provided a primitive statistical approach to sequence modeling, setting the foundation for more advanced architectures [64] such as the *Recurrent Neural Networks (RNNs)* [45] and the *Long Short-Term Memory (LSTM)* [92] networks which introduced mechanisms to model sequential data and capture temporal dependencies. *Transformers* (see Section 2.1.5) advanced this further by using self attention to process sequences in parallel and enabling more efficient and scalable learning of long-range relationships.

Recurrent Layers Recurrent layers are the core building block of Recurrent Neural Networks (RNNs) [45], introduced in the late 1980s. Recurrent layers handle sequential data (e.g., time series, language, speech) by using internal loops to retain information from previous inputs, effectively allowing the network to maintain a form of memory [70]. This mechanism enables the modeling of sequential dependencies by propagating context through the model’s hidden states. Figure 4 illustrates a recurrent cell, where the hidden state \mathbf{h} receives input \mathbf{x} and is updated over time through recurrent connections to capture temporal dependencies. The following equation describes the operation in an RNN cell:

$$\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \quad (13)$$

At each timestep t , the hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ is updated by combining the previous hidden state \mathbf{h}_{t-1} and the current input $\mathbf{x}_t \in \mathbb{R}^{d_x}$. The learned weight matrices $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ and $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d_x}$, together with the bias $\mathbf{b}_h \in \mathbb{R}^{d_h}$, determine how past and current information are integrated, and the activation function ϕ (commonly tanh or ReLU) introduces nonlinearity. This mechanism allows the RNN to retain information over time by carrying forward contextual knowledge through the hidden state. The hidden state \mathbf{h}_t is then passed to deeper layers (if present) and reused at the next timestep within the same layer. Unfolding the recurrent layer across timesteps reveals how the same cell is reused to process input sequences while maintaining memory through the hidden states.

From the above analysis, we can deduce that RNNs are inherently serial networks, since they need to process one input element at a time (e.g., a word within text). Due to this characteristic, the main computational kernel of RNNs is Matrix-Vector Multiplication (commonly realized through **GEMV**, or **GEMM** when batched) between the weight matrices and the input and hidden-state vectors.

RNNs found significant use in language modeling and speech recognition tasks; however, they faced considerable challenges, most notably the vanishing and exploding gradient problems, which severely limited their ability to capture long-range dependencies effectively [70].

Long Short-Term Memory (LSTM) To overcome the limitations of RNNs in capturing long-range dependencies, Long Short-Term Memory (LSTM) networks emerged in the late 1990s, introducing gating mechanisms to better manage information flow and stabilize gradient propagation [70]. LSTMs are also recurrent layers, extending simple RNNs by introducing a dedicated cell state that acts as long-term memory.

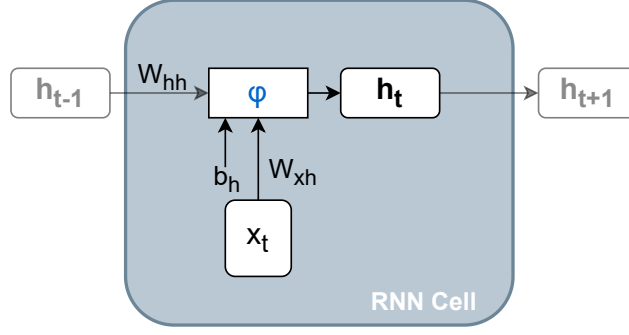


Figure 4: A basic RNN cell, where the hidden state is updated from the previous hidden state and current input, using shared parameters across timesteps.

This cell state flows through the network with minimal modification and is regulated by learned gating mechanisms. The internal structure and operations of an LSTM cell are illustrated in Figure 5.

Mathematically, let the input at timestep t be $\mathbf{x}_t \in \mathbb{R}^{d_x}$ and let $\mathbf{h}_{t-1}, \mathbf{c}_{t-1} \in \mathbb{R}^{d_h}$ denote the previous hidden state and cell state, respectively. The LSTM architecture defines three gates at each timestep: the input gate $\mathbf{i}_t \in \mathbb{R}^{d_h}$, the forget gate $\mathbf{f}_t \in \mathbb{R}^{d_h}$, and the output gate $\mathbf{o}_t \in \mathbb{R}^{d_h}$. Each gate is associated with:

- the forget gate $\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$ determines which parts of the previous cell state \mathbf{c}_{t-1} are retained;
- the input gate $\mathbf{i}_t = \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$ controls how much new information $\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$ enters the cell state;
- the cell state update $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ combines retained memory with candidate values;
- the output gate $\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$ filters the updated cell state, producing the hidden state $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$.

Here, $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_c \in \mathbb{R}^{d_h \times d_x}$ and $\mathbf{U}_f, \mathbf{U}_i, \mathbf{U}_o, \mathbf{U}_c \in \mathbb{R}^{d_h \times d_h}$ are the learned weight matrices, and $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o, \mathbf{b}_c \in \mathbb{R}^{d_h}$ are the bias parameters for their respective gates. The symbol \odot denotes element-wise (Hadamard) multiplication, and the sigmoid function $\sigma(\cdot)$ maps gate values to $(0, 1)$.

The gates allow the LSTM to selectively use, write and forget information, making it highly effective for tasks involving long-term dependencies like language modeling, translation and time-series forecasting.

As with RNNs, LSTMs are inherently serial models whose core computational kernels are matrix–vector multiplications (or matrix–matrix when batched). However, each timestep now requires several such operations, one for each gate, making LSTMs computationally heavier than simple RNNs. Note that more compact variants, such as Gated Recurrent Units (GRUs), reduce the number of gates and thus require fewer computations while still retaining key gating benefits [27].

2.1.5 Transformer networks and the Attention mechanism

While LSTMs improved long-term dependency modeling, the real breakthrough came with the introduction of Transformers in 2017 [236]. Transformers replaced recurrence entirely with a **self-attention** mechanism, which allowed the model to weigh the importance of every token in the input sequence relative to all others simultaneously, significantly enhancing parallel computation and contextual understanding [236].

Transformers are deep architectures composed of repeated **Transformer blocks**, each containing multiple internal sublayers, as shown in Figure 6. The term ‘Transformer layer’ typically refers to one such block, and modern LLMs are built by stacking many of these blocks. This structure is considerably richer than the linear, fully connected, and recurrent layers discussed so far. A standard Transformer architecture

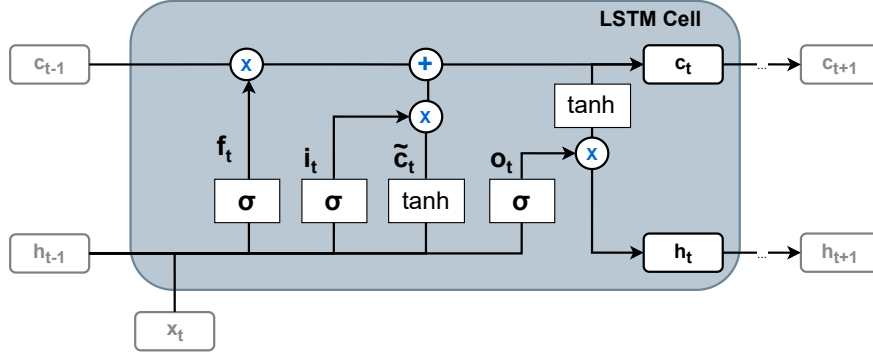


Figure 5: Structure of an LSTM cell with its gating mechanisms and cell state update.

consists of two main components: the **encoder** and the **decoder**, each formed by repeated layers with identical subcomponents. The encoder processes the entire input sequence and produces a contextualized representation, while the decoder generates the output sequence autoregressively, using both the encoder’s representation and the tokens generated so far.

Since Transformers contain no recurrence or convolution, they rely on positional encodings to inject information about token order. These encodings are added to the input embeddings before the first attention layer. A detailed discussion of positional encodings is beyond the scope of this work, but they are included in Figure 6 for completeness.

At the core of the Transformer is the attention mechanism, particularly multi-head self-attention, which applies several attention operations (heads) in parallel. This enables each token to attend to every other token in the sequence, dynamically weighting their importance when forming contextualized representations. Typically, the encoder applies bidirectional self-attention, allowing every token to see the full sequence simultaneously. In contrast, the decoder operates autoregressively: it generates one token at a time, and each prediction depends only on previously generated tokens, enforced through causal masking. To avoid recomputing attention over past tokens during generation, modern implementations employ a key–value (KV) cache mechanism [129].

Attention can be summarized as follows: An input $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ which is a sequence of T tokens with embedding size d_{model} is multiplied with three learned weight matrices, a triplet $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ for each one of the H heads ($d_k = d_{\text{model}}/H$), to compute the projected $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{T \times d_k}$ embeddings² as follows:

$$\begin{aligned} \mathbf{Q} &= \mathbf{XW}_Q \\ \mathbf{K} &= \mathbf{XW}_K \\ \mathbf{V} &= \mathbf{XW}_V \end{aligned} \tag{14}$$

Then attention is computed as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{QK}^T}{\sqrt{d_k}} \right) \mathbf{V} \tag{15}$$

The product of \mathbf{Q} and \mathbf{K}^T gives raw attention scores, which are then scaled by $\sqrt{d_k}$ (the key dimensionality) to stabilize gradients. A softmax is applied row-wise to convert these scores into attention weights, highlighting the most important parts of the input (with respect to the query). These weights are then used to compute a weighted sum of the value matrix \mathbf{V} , allowing the model to focus on relevant information. Matrix-matrix multiplication is the core operation of the Transformer’s computations. These matrix-matrix multiplications

²The attention formulation does not require the values to have the same dimensionality as the queries and keys, although standard Transformers typically choose \mathbf{Q}, \mathbf{K} , and \mathbf{V} to have identical sizes.

become matrix-vector multiplications in the decoder autoregressive stage when the output is processed sequentially, which is typical for local deployments where the batch size is equal to one.

Masked attention shown in Figure 6 is a variant of attention where the model is prevented from considering future tokens in the sequence. In practice, this is done by applying a mask to the attention scores (\mathbf{QK}^T), setting the values corresponding to ‘future’ positions to $-\infty$ before the softmax, ensuring they receive zero weight after the softmax. As a result, each token can only attend to itself and earlier tokens, never to later ones.

This architectural innovation has since revolutionized NLP, powering state-of-the-art models such as GPT and BERT, and extending to applications in computer vision and multimodal tasks [38]. Despite their success, Transformers face challenges with scalability due to their quadratic complexity in sequence length, leading to high computational and memory costs. As a result, recent research has focused on developing more efficient variants, including sparse attention mechanisms [24], and linearized attention [115].

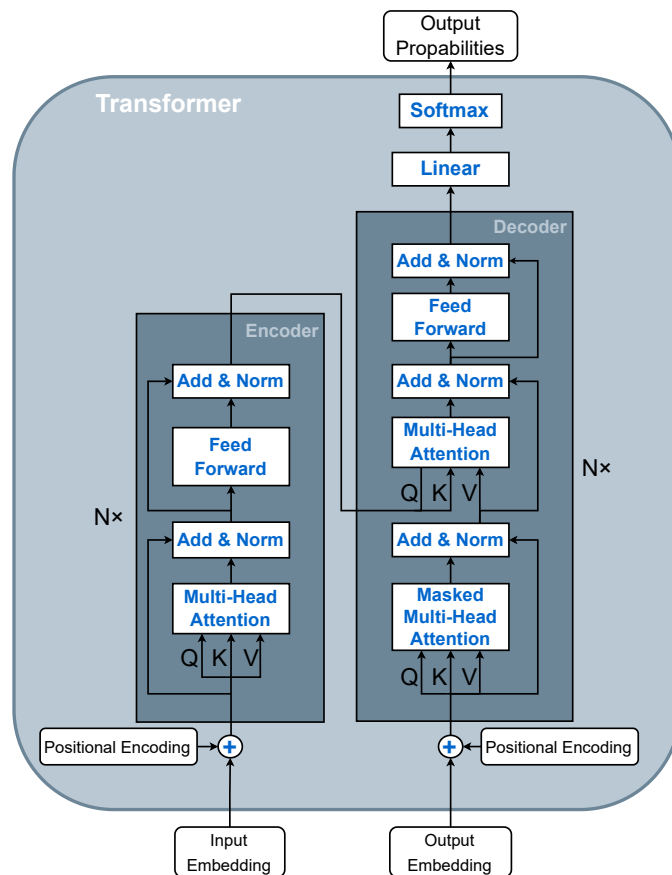


Figure 6: Basic building block of the Transformer model architecture [236]. Dark blue boxes indicate the repeated Transformer blocks in the encoder and decoder. Each block contains the same subcomponents, centered around multi-head attention, and these blocks are stacked to form the full encoder-decoder structure.

2.1.6 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) were developed to learn from graph-structured data, where entities (nodes) and their relationships (edges) must be effectively represented. Traditional machine learning methods, designed for Euclidean data such as tables, sequences, or images, struggle with the irregular structure and complex interdependencies in graphs. GNNs extend deep learning to non-Euclidean domains by leveraging node features and graph connectivity to propagate information. Graphs model structured relationships in

real-world systems such as social networks (users and friendships), molecules (atoms and bonds), and knowledge graphs (entities and relations). Like traditional DNNs, GNNs use multiple layers to aggregate and transform information from neighboring nodes, building increasingly abstract representations. By capturing both the topology and node/edge attributes, GNNs can learn meaningful embeddings that reflect both local and global graph properties.

GNNs take as input graph-structured data with node and edge features. Each node in the graph can have features that describe the entity it represents (e.g., a user’s profile in a social network, or the atoms in a molecule). Each edge may also have features that describe the relationship between the nodes (e.g., friendship strength between users, or chemical bond types between atoms). In a typical use case, the GNN can process either one large graph or multiple smaller graphs. For example, in a single graph, the GNN might process a whole social network. In cases where multiple smaller graphs are involved, such as in graph classification tasks where each graph represents a different molecule, the input might consist of several individual graphs, each with its own set of nodes, edges, and associated features.

The output of a GNN depends on the specific task at hand. For node-level tasks (e.g., node classification), the output is a prediction or label for each node. For example, the GNN could predict whether a user is interested in a particular activity based on their connections and profile features. For edge-level tasks (e.g., link prediction), the output is a prediction about the relationship between nodes, such as whether two users in a social network will become friends. Finally, for graph-level tasks (e.g., graph classification), the GNN outputs a prediction for the entire graph, such as determining if a molecule is toxic based on its chemical structure or classifying a network as ‘healthy’ or ‘risky’. Across all these tasks, the key strength of GNNs is their ability to learn from both the features of the individual nodes and edges as well as their connectivity and structure, allowing them to make predictions based on the overall graph’s configuration and relationships.

A **GNN layer** is typically represented by a combination of:

- **Graph connectivity** ($\mathbf{A} \in \mathbb{R}^{N \times N}$, typically represented as an adjacency matrix or edge list)
- **Feature representation** ($\mathbf{H}^{(l)} \in \mathbb{R}^{N \times d_l}$, the node feature matrix at layer l)
- **Trainable weights** ($\mathbf{W} \in \mathbb{R}^{d_l \times d_{l+1}}$, the learnable weight matrix)

Here, N is the number of nodes in the graph, and d_l is the feature dimensionality at layer l .

The weight matrix is a key component for transforming the input features of nodes and updating their representations through learned transformations. GNNs rely on a **message-passing** (or **neighborhood aggregation**) mechanism: each node iteratively receives and aggregates information from its neighbors, updating its own representation to reflect both local and global graph context [72; 214]. This approach typically involves defining how nodes combine incoming messages (e.g., summing or averaging neighbor features) and how node states are transformed through learnable functions. As a result, GNNs can uncover patterns that arise from the interplay between node features and the structure of the network itself.

Message passing is typically formulated as:

$$\mathbf{H}^{(l+1)} = \phi\left(\mathbf{A}, \mathbf{H}^{(l)}, \mathbf{W}\right) \quad (16)$$

Where l is the layer of the GNN and ϕ a typically non-linear function like ReLU. Standard GNNs like Graph Convolutional Networks (GCNs) [118] use a fixed rule to aggregate neighboring node features. Each node updates its features based on a weighted sum of its own and its neighbors’ features in a uniform manner. This actually means that the matrix \mathbf{A} remains constant, precomputed, and captures static information about the graph, like connectivity and node degrees.

GCNs, as introduced by Kipf and Welling, update node features as:

$$\mathbf{H}^{(l+1)} = \phi\left(\tilde{\mathbf{D}}^{-1/2} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-1/2} \mathbf{H}^{(l)} \mathbf{W}\right) \quad (17)$$

where:

- $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix with self-loops.
- $\tilde{\mathbf{D}}$ is the diagonal degree matrix of $\tilde{\mathbf{A}}$.
- $\mathbf{H}^{(l)}$ represents node features at layer l .
- \mathbf{W} is a learnable weight matrix.
- ϕ is a non-linearity (e.g., ReLU).

The term $\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{H}^{(l)}$ computes the messages through matrix multiplication that involves the normalized adjacency matrix $\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}$.

Advanced GNN architectures like Graph Attention Networks (GAT) [237] introduce mechanisms, such as attention weights, that allow the model to learn the relative importance of neighboring nodes during aggregation. Thus, while the weight matrix helps in feature transformation, the model’s architecture determines how neighboring node relationships and their significance are incorporated into the node embeddings. This means that the matrix \mathbf{A} is computed dynamically at each layer of the GNN and each inference pass. GATs introduce an attention mechanism to weigh the importance of each neighbor’s contribution. For a given node i and the corresponding feature vector \mathbf{h}_i , attention coefficients α_{ij} are computed for each neighbor $j \in \mathcal{N}(i)$ using:

$$e_{ij} = \text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]), \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})}. \quad (18)$$

These coefficients α_{ij} are then used to compute a weighted sum over the transformed neighbor features. This mechanism enables GATs to focus more on relevant neighbors and adapt to heterogeneity in the graph.

2.2 From model architecture to model inference

In this section, we describe how common end-to-end model development and inference pipelines operate. First, we outline a general workflow applicable to most cases and we separate it into three distinct steps: Model training, Model Optimization and Inference. These are depicted in Figure 7 and analyzed in the next paragraphs.

2.2.1 Model training

The process of Model Training refers to the process that makes a DNN learn to perform a specific task, much like a student learns from practice and correction.

Typically, a model is developed using a popular deep learning framework such as PyTorch [192], TensorFlow [1], or JAX [13], and is trained on a collection of training samples called a dataset. The dataset contains thousands or millions of examples of the type of information we want the model to process (e.g., images for a vision task or text for a language task). For many common tasks, data are accompanied by specific supplementary information, known as metadata, and, most importantly, ground truth labels. These labels are the ‘correct answers’ (e.g., the label for an image is ‘cat’ or ‘dog’) and are used exclusively in a process called supervised learning.

The core steps of training are as follows:

1. **Architecture Selection:** Based on the task, a suitable network architecture is chosen (like the ones discussed in the previous section).
2. **The Loss Function:** To guide this adjustment, a crucial mathematical function called the loss function is used. It measures precisely how far the model’s current prediction is from the correct answer (the ground truth label). A high loss value means the model is performing poorly, and a low loss value means it is performing well.

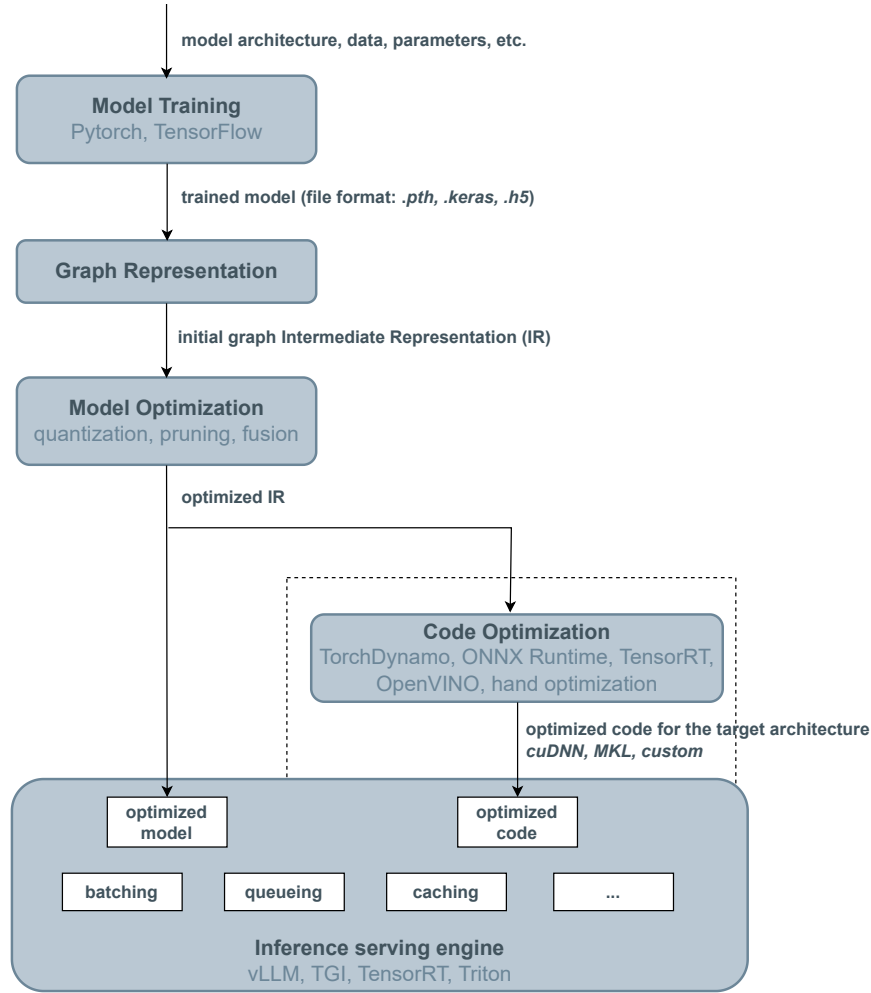


Figure 7: From architecture to model deployment.

3. **Gradient Computation:** The loss function provides feedback to the network on how the weights should change. This feedback takes the mathematical form of gradients—which indicate the direction and magnitude of change needed for every single weight to reduce the loss.
4. **Optimization:** The weights are then updated using an iterative optimization method, such as Gradient Descent, Stochastic Gradient Descent (SGD) [207], or Adam [117]. These methods use the gradients to modify the weights, aiming to minimize the loss.

In this way, the model gradually, over many iterations, learns to capture the underlying patterns in the data. To make the training process computationally efficient, especially with massive datasets, several engineering techniques are employed:

- **Batch Training:** Instead of feeding individual data points one by one, models are trained on small groups of samples called batches. The model calculates the loss and gradients for all samples in a batch simultaneously, and then uses the average gradient to update the weights. This method significantly speeds up the training process by leveraging the parallel processing power of accelerators such as GPUs.
- **Hyperparameters:** Beyond the internal weights, which the model learns, there are external settings called hyperparameters that must be chosen by the engineer. These include the learning rate (how

large each weight adjustment step should be) and the batch size itself. Tuning these parameters is crucial for achieving an effective and stable training run.

- **Epochs:** A single pass through the entire dataset is called an epoch. Training often requires many epochs to fully learn the data's patterns. However, engineers must monitor the training to prevent overfitting, where the model learns the training examples too well, including their noise, and performs poorly on new, unseen data.
- **Scaling and Distribution:** For the largest models and datasets, training must be scaled across multiple GPUs or even multiple servers, a technique known as distributed training. Frameworks like PyTorch and TensorFlow include tools to efficiently manage this distribution, allowing engineers to train models that would otherwise be impossible on a single machine.

Nowadays, the initial training step can often be skipped entirely because many powerful pre-trained models are publicly available online (e.g., via the Hugging Face Hub or PyTorch repositories). These models have already been trained on massive, diverse datasets and can be instantly adapted (fine-tuned) for a specific new task with minimal effort.

2.2.2 Model optimization³

The next goal is to use the trained model to perform inference on new data. There are multiple ways to execute inference, ranging from a simple function call in PyTorch, to a comprehensive pipeline involving exports and hardware-specific optimizations. While the simplest form of inference is common in academic research, our focus here is on identifying how the initial neural network can be enhanced, optimized, and eventually run on the target device.

The most common formats for representing a neural network are PyTorch (.pth files) or TensorFlow (.keras, .h5). These files may contain either the entire model architecture or just the learned weights. From these formats, models can be exported to other formats to enable optimizations. Model optimization is optional but often essential in order to deploy DNNs efficiently and meet specific latency and throughput targets. Typically, the first step in optimization is converting the model into a computational graph, an intermediate representation (IR) that captures the operations and data flow. Various optimizations can then be applied either to the model's weights (e.g., quantization, pruning) or to its computations (e.g., operator fusion).

There are many different model representations that can be the starting point for optimizations as such. One common representation is ONNX, an open standard that facilitates interoperability between frameworks. ONNX encapsulates both the architecture and the weights as a computational graph, where nodes represent operations and edges denote data flow. This format can be used directly for inference, but it can also serve as input for further optimization. Other common representations include TorchScript, which allows PyTorch models to be traced or scripted into an IR that can be deployed outside the Python interpreter; TensorFlow SavedModel / GraphDef, which encode the computation graph and variables for TensorFlow pipelines; TensorFlow Lite, tailored for mobile and edge inference with optimizations such as quantization; Core ML, used for deploying on Apple devices; and hardware- or vendor-specific formats/engines such as TensorRT or OpenVINO. These formats likewise hold both structure and weights, and each provides its own suite of optimizations (operator fusion, quantization, pruning, static graph transformations, etc.), making them attractive starting points for deployment pipelines. Depending on the use case (edge vs cloud, latency vs throughput, resource constraints), one may convert from one representation to another, or optimize directly on these IRs.

³Optimization in this case refers to the process of improving the model towards better performance (e.g., latency of throughput) and resource utilization (e.g., smaller memory footprint). This should not be confused with training-time optimization of the model weights.

2.2.3 Model inference

At this point, the model is in an optimized state and ready for execution. Each backend selects or generates the kernels required for inference based on the optimized graph and available hardware. These kernels may be dynamically compiled at runtime for flexibility or pre-compiled for efficiency, with modern engines often adopting a hybrid strategy. Beyond this, inference workloads can take different forms: **batch inference** prioritizes throughput in offline settings, while online inference minimizes latency for interactive tasks. To meet these demands at scale, models are deployed through **serving systems** or **engines** (e.g., vLLM [129], TGI [50], TensorRT [183]), that manage requests, scaling, caching, versioning, etc. Here, optimization techniques such as quantization or sparsity play a direct role, since they reduce compute and memory overhead, improving both latency and throughput across deployment scenarios.

2.3 Sparsity basics

Sparsity in our case refers to working on data, typically represented as matrices or tensors that are sparse, i.e., a large number of their elements are zero. Sparsity can be coarsely categorized into three main types in DNNs:

Structured sparsity assumes sparse elements in coarse parts of the tensor, as, for example, is the case in weight pruning (see Section 3.1) where one removes components such as neurons, features, filters, or even layers [137; 81], leading to model simplification that is hardware-friendly and easier to accelerate. Structured sparsity typically leads to the creation of smaller, dense neural networks, thus resorting to dense operations [251], and for this reason, we will not focus on structured sparsity in this survey. Moreover, the accuracy of such approaches is not satisfactory in several cases, and as a result, alternative sparsification methods are considered.

Unstructured sparsity assumes the most generic distribution of nonzero elements in the tensor, i.e., they can reside anywhere, without any limitation while their number and position is decided by some optimization process (e.g., in the case of pruning by an algorithm that zeros out specific components that do not contribute substantially to the model’s accuracy [131]). Unstructured pruning can reach much higher compression ratios without compromising model accuracy, but the performance of the kernels that operate on unstructured sparsity can be disappointingly low [62].

Semi-structured sparsity refers to a sparsity pattern where zero elements are not randomly distributed but follow a regular, partially predictable structure. It strikes a balance between structured and unstructured sparsity by imposing some regularity while maintaining flexibility. **N:M sparsity** is a specific and widely adopted subtype of semi-structured sparsity [171], and it will be the main semi-structured paradigm discussed in this survey. In N:M sparsity, exactly n out of every m elements remain nonzero, enabling both substantial compression (typically lower than unstructured sparsity, nevertheless) and more efficient execution on supported hardware. For example, in 2:4 sparsity, every group of four elements contains exactly two nonzero values. **Block sparsity** allows nonzero elements in fixed shaped dense blocks [76] and utilizes data structures like BSR and specialized kernels that expect the nonzeros in this form.

In the following paragraphs, we elaborate more on unstructured and N:M sparsity that are heavily employed in research and practice. In particular, we discuss data structures that store sparse data in each case, computational kernels involving sparse data and, where relevant, hardware support provided by vendors to improve the performance of these kernels.

2.3.1 Unstructured sparsity

Data structures Two dimensional arrays, which we henceforth call matrices, are the core data structure to store sparse data in algebraic operations heavily used in DNNs. A matrix is considered sparse when a substantial fraction of its elements have zero values, and therefore can be stored in a compressed form by omitting them. This creates the need of somehow expressing and storing the structure of the matrix, i.e., the positions of the remaining nonzero elements.

A straightforward approach to describe a sparse matrix is the so-called *Coordinate (COO)* format (Figure 8), the de facto way to store matrices in files. In COO, each nonzero element is stored alongside its indices, i.e., its position inside the matrix (row and column). In several cases, the memory footprint of the sparse matrix is of vital importance, so the state-of-practice approach to store matrices in memory is the *Compressed Sparse Row (CSR)* format (Figure 9) which stores as metadata the number of elements per row (in the form of a prefix sum) and the column index of each nonzero element.

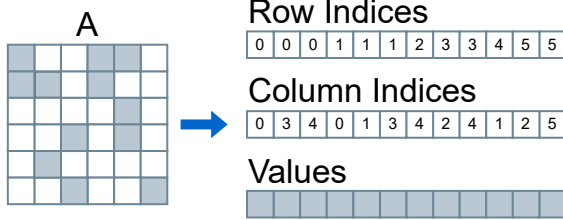


Figure 8: The COO storage format for sparse matrices.

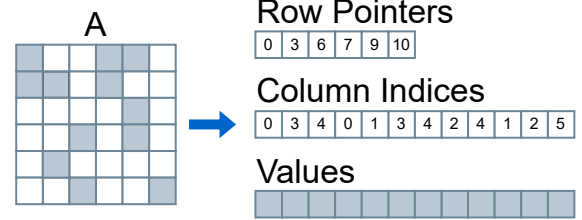


Figure 9: The CSR storage format for sparse matrices.

Furthermore, the optimization process of several sparse kernels brought forward a variety of format families. The most notable families are the ELL-type formats (a variation of CSR in which we assume a constant row length, enforcing it by padding smaller rows with extra zeros, and therefore does not need the RowPtr array), the DIA-type formats (storing the non-empty diagonals, padding smaller ones to the same length as the number of rows), and Block-type formats (variations where we assume that nonzeros are clustered in 2D blocks of constant size, padding with zeros when needed, and storing the positional data of the blocks instead of each element; most commonly encountered in the form of BSR - Blocked Sparse Row, the blocked variant of CSR). Lastly, more complex formats like SparseX [44], CSR5 [152], LCM [23] and DIV [61] specifically target the optimization of the SpMV kernel (see the next paragraph).

Computational kernels Below we provide information on the basics of sparse computational kernels that we revisit in this survey.

Sparse Matrix - Vector Multiplication (SpMV) *Sparse Matrix-Vector Multiplication (SpMV)* is an operation that takes a sparse matrix and a dense vector as input and produces a dense vector as output. SpMV is a fundamental operation in HPC, and in particular a core building block for the solution of large sparse linear systems [210]. In the case of DNNs, SpMV is employed in RNNs, LSTMs (see Section 2.1.4) and in the decoder stage of Transformer networks in cases where input is not processed in batches.

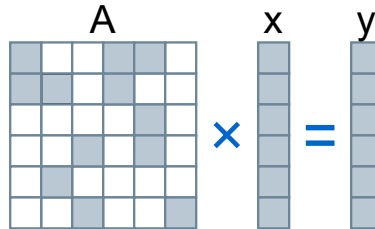


Figure 10: Sparse matrix-vector multiplication (SpMV).

In the typical implementations of SpMV, only the nonzero elements of the matrix and their positions are stored using CSR or COO. This reduces memory usage and computational effort by skipping unnecessary operations with zero values. We next provide pseudocode for these important cases.

```
for (j=0; j<nnz; j++)
    y[RowIdx[j]] += Values[j] * x[ColIdx[j]];
```

Listing 1: SpMV using COO Format

```
for (i=0; i<rows; i++)
    for (j=RowPtr[i]; j<RowPtr[i+1]; j++)
        y[i] += Values[j] * x[ColIdx[j]];
```

Listing 2: SpMV using CSR Format

Sparse Matrix - Dense Matrix Multiplication (SpMM) *Sparse Matrix - Dense Matrix Multiplication* (SpMM) is an operation that takes a sparse matrix and a dense matrix as input and produces a dense matrix as output. Specifically, each element in the resulting matrix is computed by performing a dot product between a row of the sparse matrix and a column of the dense matrix. During this process, only the nonzero elements of the sparse matrix are involved in the multiplication, while the structure of the dense matrix is fully utilized in each computation. SpMM is used in fields such as scientific computing, machine learning, graph processing, data mining, and computational physics. SpMM is a key computational kernel in sparse inference as discussed in Section 4.

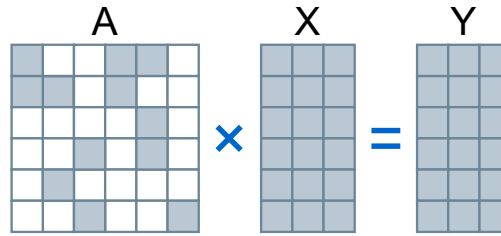


Figure 11: Sparse matrix-matrix multiplication (SpMM).

Sample code:

```
for (i=0; i<rows; i++)
    for (j=RowPtr[i]; j<RowPtr[i+1]; j++)
        for (k=0; k<K; k++)
            Y[i][k] += Values[j] * X[ColIdx[j]][k];
```

Listing 3: SpMM using CSR Format

Sampled Dense-Dense Matrix Multiplication (SDDMM) *Sampled Dense-Dense Matrix Multiplication* (SDDMM) is an operation where two dense matrices are multiplied, but the results are computed only at specific positions defined by a sparse mask. Given dense matrices X1 and X2, and a sparse mask matrix A, indicating the positions of interest, SDDMM computes the dot product of the corresponding row from X1 and column from X2 only at the nonzero locations of A. The output is a sparse matrix with the same sparsity pattern as A, containing the sampled results of the dense matrix multiplication. SDDMM is used in fields like graph neural networks, recommendation systems, natural language processing, and scientific computing.

Sample code:

```
for (i=0; i<rows; i++)
    for (j=RowPtr[i]; j<RowPtr[i+1]; j++)
        for (k=0; k<K; k++)
            Y_Values[j] += X1[i][k] * X2[k][j];
```

Listing 4: SDDMM using CSR Format

Sparse convolution *Sparse convolution* is a variation of standard convolution where the input data or the convolutional filters contain many zero values, and the computation is performed only on the nonzero elements. This reduces the number of operations by skipping computations involving zeros. In practice, sparse convolution applies a convolutional kernel to an input feature map while leveraging the sparsity pattern to optimize performance.

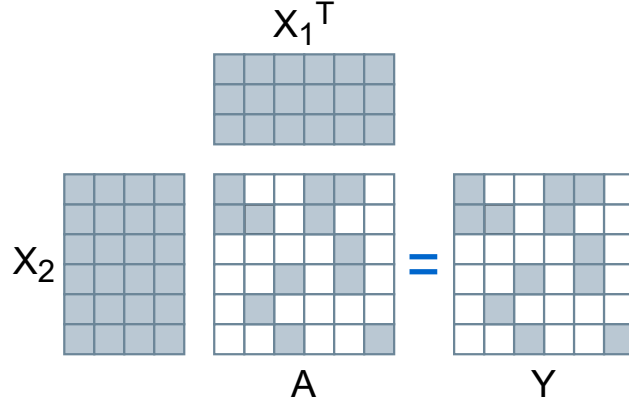


Figure 12: Sampled Dense-Dense Matrix Multiplication (SDDMM).

Sample code (Dense input, sparse convolution kernel):

```
for (i=0; i<H-K+1; i++)
  for (j=0; j<W-K+1; j++)
    for (n=0; n<nnz; n++)
      output[i][j] += k_val[n] * input[i+k_row[n]][j+k_col[n]];
```

Listing 5: Sparse Convolution using COO Format

Sparse convolutions can be further categorized depending on where their sparsity originates, leading to three main types with each type requiring specialized optimization strategies [253]. *Filter Sparse Convolution* specifically addresses sparsity within the convolution kernels themselves, converting the operation into an SpMM kernel via `im2col` transformation where applicable. *Activation Sparse Convolution* focuses on the common scenario where activations after non-linearities (like ReLU) are sparse, designing efficient vectorized routines that only process nonzero activation (input) values. *Masked Sparse Convolution* is the same as a standard convolution kernel with the addition of an element-wise multiplication with a sparse binary mask that determines the sparsity pattern of the output. *Sparse Convolution for multidimensional input* or *submanifold convolution* (see Appendix B) is a specific case of masked convolutions that strictly maintains the input sparsity pattern, ensuring that output nonzeros only appear where input nonzeros existed, which is critical for applications involving intrinsically sparse data like 3D point clouds.

Other sparse kernels Other sparse kernels like *Sparse Matrix - Sparse Matrix Multiplication* (SpGEMM) [14] or *Sparse Matrix - Sparse Vector Multiplication* (SpMSpV) [9] have been discussed and implemented for applications in other domains. To our knowledge, they are not heavily used in DNN inference and thus not further analyzed in this survey.

2.3.2 Semi-structured N:M sparsity

Data structures Existing widely used sparse matrix formats (CSR, COO etc.) are not well-suited for representing matrices with moderate sparsity levels, such as those resulting from n:m structured pruning. To better exploit these regular sparsity patterns, alternative representations have been introduced [171; 143], consisting of the data and the index array. The data array stores only the nonzero values, while the index array encodes the positions (ranging from 0 to m-1) of the nonzero elements within each group of m elements. For instance, in 2:4 structured sparsity, which is natively supported by NVIDIA’s Sparse Tensor Cores and presented in Figure 13, only 2 out of every 4 weights are kept. Their positions within the group are encoded using 2 bits each, resulting in 4 bits of metadata per group in the index array. Note that, computational kernels for N:M sparsity that operate on such data structures involve state-of-the-art, recent implementations that are discussed in Section 4.

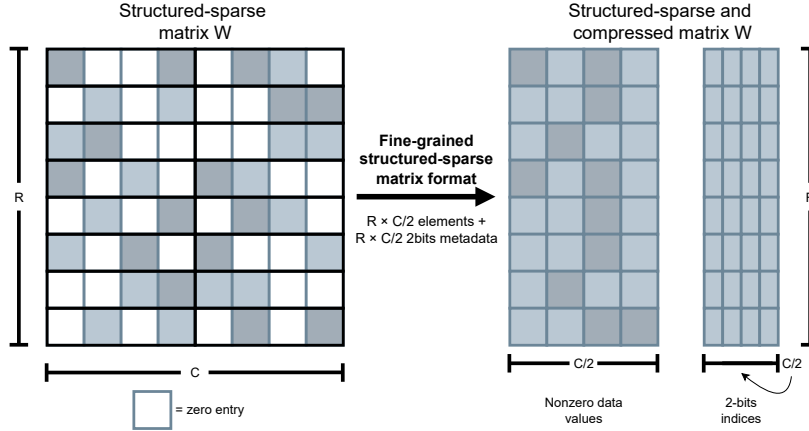


Figure 13: Data structure for semi-structured sparsity [171].

Hardware support NVIDIA’s Sparse Tensor Cores, introduced with the Ampere GPU architecture, are an extension of the Tensor Core architecture, for accelerating matrix operations under structured sparsity. They support the 2:4 sparsity pattern, where exactly two nonzero elements are retained in every group of four consecutive weights. The cores are optimized to perform sparse matrix–dense matrix multiplication (SpMM) efficiently by skipping unnecessary multiplications with zero, effectively halving the number of compute operations and achieving nearly a $2\times$ speedup. The sparse operand is stored in a compressed format, using 2-bit metadata per group to encode the nonzero positions, for fast decoding and minimal storage overhead. Sparse Tensor Cores support a range of input formats, FP16, BF16, INT8, and TF32, and nearly double the arithmetic throughput compared to their dense counterparts.

3 Sparsity in DNNs

In this section we aim to elaborate on all forms of sparsity that are discussed in the context of deep neural networks. Our main focus, however, is to further analyze those forms of sparsity that lead actually to representation of sparse data and invocation of sparse computational kernels. As we clarify in this section, some of the cases that use terms relevant to sparsity, do not ultimately resort to sparse computations.

In general, there are three sources of sparsity in the inference of deep learning tasks: *enforced sparsification*, *natural sparsity* and *ephemeral sparsification*.

Enforced Sparsification refers to the process of deliberately sparsifying the inference process in order to save computation overhead and data representation. Enforced sparsification can take form in two ways: **model (weight) pruning** and **attention sparsification**. The main elements that can be sparsified (pruned) are: weights, neurons and filters in convolutional layers [167]. Attention sparsification, on the other hand, uses properly selected masks to avoid some tokens in the attention computation.

Natural sparsity is inherent to the problem at hand and mainly refers to GNNs where the problem is represented with a graph, a naturally sparse data structure. We also discuss in Appendix B the special case of sparse input in 3D point clouds [254; 230; 97; 264], but as we discuss there, although they carry the term ‘sparse’, the computations involved are dense.

Ephemeral Sparsification is applied during the computation of each example individually in an inference step and it is only relevant for this example. This case refers to the activation functions like ReLU and SoftMax that lead to sparsified outputs of DNN layers. Ephemeral sparsification is dynamically updated for each example and configured during inference and training.

Finally, we also consider the special cases of the **Mixture of Experts (MoE)** [107; 217; 55] and **Quantization** [71]. A MoE is a DNN architecture that divides a large model into multiple smaller specialized components, called experts. For each input, only a subset of these experts is activated, allowing the model to

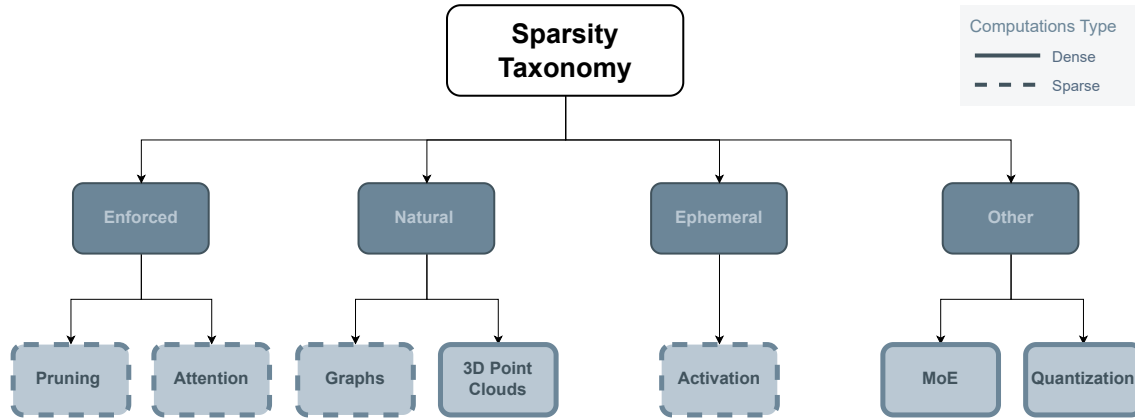


Figure 14: Forms of sparsity in DNNs.

focus on the most relevant knowledge while reducing computation costs. This approach enables high model capacity with improved efficiency compared to using a single large model for all tasks. Since in MoE models only a small subset of experts (and thus parameters) is activated per input, this approach is considered to apply ‘conditional sparsity’. However, MoE models do not use sparse data structures and computations and as such, we do not consider them further and provide more information on this popular approach in Appendix C.

Quantization is a technique that reduces the precision of numerical values (such as model weights and activations) from high-precision formats like 32-bit floating point to lower-precision ones like 8-bit integers. This decreases the memory footprint and computational requirements of a model, enabling faster inference and lower energy consumption. While it can introduce small accuracy losses, quantization is often used to make large models more efficient for deployment on limited hardware. Clearly, quantization cannot be considered a form of sparsity, as it relies on dense computations with lower precision. However, since it is frequently discussed in par with sparsification methods, we provide some information on quantization in Appendix D.

Figure 14 provides an summary of the various forms of sparsification together with the computations involved. In the following paragraphs we further discuss those forms that involve sparse data and computations.

3.1 Weight pruning

Weight pruning is the process of removing weights from the layers of a DNN with a primary goal to improve model performance, such as inference time, required flops and latency, along with reducing memory footprint. Secondly, the process can have a positive impact on other goals like generalization, since sparsity can act as a form of regularization [93]. Three major types of pruning have been extensively discussed: **structured pruning**, which removes whole neurons, channels, or filters; **semi-structured pruning**, which removes weights in regular blocks or patterns; and **unstructured pruning**, which removes individual weights. Model sparsification can be applied at different stages of the training pipeline, including before training (e.g., sparse initialization), during training (gradual pruning), or after training (post-hoc pruning) [93; 62]. In practice, different layers and architectures exhibit different sensitivities to pruning, and pruning methods therefore assign sparsity levels per layer using heuristic or empirically driven criteria rather than a universal rule [98; 48; 154]. Some layers are typically pruned more conservatively (for example, downsampling or projection layers in residual networks), while others can sustain higher sparsity without substantial accuracy degradation [174; 88]. It is also important to note that sparsifying different layers can lead to very different computational savings. In CNNs, the FLOP contribution of a layer depends on both its spatial resolution and its channel dimensionality, which vary inversely across the network: early layers operate on large spatial maps but with fewer channels, while deeper layers have many more channels but operate at much smaller spatial sizes. As a result, the FLOP reduction achieved by pruning a given layer is not uniform across the network, and

choosing how to distribute sparsity—given an overall sparsity or FLOP-reduction target—requires balancing accuracy considerations with practical efficiency. Therefore the sparsification process and the related design choices drastically affect the level and pattern of sparsity created, with a high subsequent impact on the efficiency of the relevant operations at inference time. More details about methods for model sparsification like random pruning, magnitude pruning and regularization can be found in Appendix A.

Sparsity has been applied to a wide range of architectures and layer types, including fully connected layers, recurrent networks, convolutional networks, and transformer-based models. Fully connected layers have been extensively explored in pruning research, with both structured (e.g., neuron or channel removal) and unstructured weight pruning investigated in early and recent works [131; 81]. LSTM and RNN layers can be sparsified similarly to fully connected layers, supporting both structured and unstructured pruning, as demonstrated in methods like ESE (block-balanced sparsity) [82] and MASR (bitmap scheme with moderate sparsity) [77]. Convolutional layers have also been widely studied, with approaches covering unstructured, structured, and channel/filter-level pruning [174; 87]. More recently, transformer-based models have become an important target for pruning, with sparsification applied to attention projections, feed-forward layers, and embeddings [125; 58]. In contrast to earlier deep architectures, which were often trained in a parameter-rich but data-limited regime, modern large language models (LLMs) are typically trained on hundreds of billions to trillions of tokens, resulting in multiple tokens per parameter [94]. This higher data-to-parameter ratio alters the model’s redundancy profile, and consequently, simpler sparsification techniques that proved effective in previous settings tend to be less reliable for today’s LLMs.

Following the analysis on the operations involved for each one of the layers in Section 2.1, the following apply:

- When a Fully Connected layer is pruned, the weight matrix \mathbf{W} from Equation 8 becomes sparse. Thus, if the operation is applied on single input, the computation involved is transformed from matrix-vector to sparse matrix-vector multiplication (**SpMV**). If the operation is applied on batches of input, the relevant matrix multiplication is transformed to sparse matrix - matrix vector multiplication (**SpMM**).
- When a Convolutional layer is pruned, then the convolution operation is transformed into *sparse convolution*. In the cases where the `im2col` and matrix multiplication is employed, the process is transformed again to `im2col` and sparse matrix - dense matrix multiplication (**SpMM**).
- When a Recurrent or LSTM layer is pruned, the relevant weight matrices (see Section 2.1.4) become sparse. The operations involved are thus transformed from matrix-vector to sparse matrix - dense vector multiplication (**SpMV**). Again, if the input is batched, a sparse matrix - matrix vector multiplication is used (**SpMM**).
- In Transformer and LLM architectures, most components (including attention projections, feed-forward layers, and embeddings) are implemented as linear layers. When these linear layers are pruned (e.g., in the computation of \mathbf{Q} , \mathbf{K} , and \mathbf{V} , the weight matrices \mathbf{W}_Q , \mathbf{W}_K , and \mathbf{W}_V in Equation 14 are becoming sparse), their associated dense matrix multiplications are transformed into sparse matrix-dense matrix multiplications (**SpMM**).

3.2 GNNs

Sparsity may arise in GNNs at multiple levels of the model and data pipeline. We can attest that sparsity in GNNs arises in all three forms. In this paragraph, we comment on the most frequently discussed form: the natural sparsity of GNNs. In Section 3.5 we make a brief discussion on other potential forms that may be relevant to GNNs.

GNNs inherently operate on graph-structured data, where natural sparsity arises from the topology of real-world graphs. These include social networks, biological systems, knowledge graphs, and citation networks, which typically show low average node degrees (in some cases lower than 10) compared to the total number of nodes. As such, the adjacency matrix representing the graph is highly sparse (in several cases less than 1% of the elements are nonzeros), and this structural sparsity is central to GNN design. Additionally, in

some domains (e.g., natural language processing or biology), node and edge features can also be sparse - for instance, multi-hot encoded attributes or bag-of-words representations. Because of this, graph connectivity is almost always represented using sparse formats like edge lists or sparse matrices.

This form of sparsity is not only widespread but foundational. Nearly all GNN architectures are designed to exploit this input sparsity by performing computations only along nonzero entries in the adjacency matrix. Message passing frameworks and libraries such as Deep Graph Library (DGL) [244] and PyTorch [56] are structured around this assumption, using sparse matrix operations as their core computational primitives.

As discussed in Section 2.1.6, the typical message-passing operation in GNNs is of the form:

$$\mathbf{H}^{(l+1)} = \phi\left(\mathbf{A}, \mathbf{H}^{(l)}, \mathbf{W}\right) \quad (19)$$

where \mathbf{A} is a sparse matrix following the pattern of the adjacency matrix of the graph, $\mathbf{H}^{(l+1)}$, $\mathbf{H}^{(l)}$ the node features at layers $l + 1$ and l respectively and \mathbf{W} is a learnable weight matrix.

Message-passing in Graph Convolutional Networks (see Equation 17) utilizes a static matrix \mathbf{A} and the computation involves the computation of $\mathbf{H}^{(l+1)}$ using \mathbf{A}, \mathbf{D} (a degree matrix – also static) and $\mathbf{H}^{(l)} \mathbf{W}$. The computational kernel involved is **SpMM**. Other, more advanced networks like GraphSAGE [79], Relational Graph Convolutional Networks (R-GCN) [215] and Graph Isomorphism Networks (GIN) [259] use more elaborate functions for message passing, but in terms of sparse computations they fall into the same category as GCNs, relying on the **SpMM** operation.

On the other hand, Graph Attention Networks (see Equation 18) need to recalculate matrix \mathbf{A} at each layer dynamically introducing essentially an attention mechanism. Matrix \mathbf{A} retains the same structure as the adjacency matrix, but its values need to be recomputed at each step. The computation of this attention matrix \mathbf{A} is based on a Sampled Dense-Dense Matrix Multiplication (**SDDMM**) using the adjacency matrix of the graph as mask on the product. This operation is then followed by **SpMM** to compute the final matrix $\mathbf{H}^{(l+1)}$.

3.3 Sparse activations

Inference can be accelerated by utilizing ephemeral sparsity, i.e., the type of sparsity that occurs when sparse output is produced by the activation operation of a DNN layer. Modern DNNs use an activation function like ReLU or alternatives (see Section 2.1.1), which inherently produce sparse activations by clipping a large number of layer outputs to zero. We frame the sparse activation landscape in the two following groups, **inherent** and **explicitly induced sparsity**, which are discussed next.

In the inherent activation sparsity zeros emerge automatically from the activation itself. To be more elaborate, classic ReLU yields sparsity by deterministically mapping all negative pre-activations to zero. Thresholded variants, in the manner of FATReLU [127], generalize this mechanism by introducing a positive threshold, suppressing small but non-informative responses. In practice, this can be deployed in a “static” method that simply replaces the activation in a pre-trained model without retraining, or a “dynamic” thresholding approach, more of which will be discussed in the following group.

Although ReLU-based techniques remain the most widely adopted ones, mainly due to their plug-and-play simplicity and already very high zero yield, several significant alternative approaches achieve comparable or even superior sparsity control and efficiency. Ranking-based gating functions such as k-WTA [257] apply a global top- k filter to an N -dimensional activation vector \mathbf{y} : only the k largest entries are preserved and all others are set to zero. To adapt across layers of differing width, k is typically chosen via a fixed sparsity ratio $\gamma \in (0, 1)$, so that $k = \lfloor \gamma \times N \rfloor$ in each layer. The runtime cost of selecting the top- k elements is $O(N)$, equivalent to a standard ReLU pass.

In all the cases mentioned, the zero pattern is a deterministic function of the current pre-activations alone:

$$\begin{aligned} S_{\text{ReLU}} &= \{i \mid z_i \leq 0\}, \\ S_{\text{FATReLU}} &= \{i \mid z_i \leq \tau\}, \\ S_{k\text{-WTA}} &= \{i \mid \text{rank}(z_i) > k\} \end{aligned} \quad (20)$$

with τ and k set as hyperparameters and no auxiliary sparsity-inducing loss, since zero values arise inherently during the forward pass from the activation itself.

Consistent with this view, Mirzadeh et al. [170] demonstrate that swapping GELU with plain ReLU in billion-scale language models produces extremely sparse feed-forward layers, which can be rapidly fine-tuned to recover any lost accuracy. Again here, sparsity is a direct by-product of the activation form.

While inherent activation functions deterministically produce zeros from the current pre-activations and are governed by architectural hyperparameters (τ, k) , their density may not align with the designated task or hardware-level targets. To obtain finer, layer-adaptive and data-adaptive control, one can explicitly induce sparsity during training or adaptation. In explicitly induced activation sparsity the network keeps its usual non-linearity, but auxiliary mechanisms, like density-penalizing loss terms, curriculum schedules or quantization deliberately drive intermediate activations toward zero. A good example of this category is L1 Regularization, which adds a penalty $\lambda \|\alpha\|_1$ on the intermediate activations a , shrinking small responses toward zero and increasing at exactly zero without changing the activation function itself. A closely related but often stronger alternative is the square Hoyer regularizer, proportional to $(\|\alpha\|_1)^2 / \|\alpha\|_2^2$, which is scale-invariant and concentrates energy into fewer coefficients, typically yielding higher sparsity at similar accuracy [127]. Building on these induced distributions, Activation Map Compression [66] augments training with an L1 term and then quantizes activations to 8-bit followed by entropy coding, exploiting the elevated fractions of zeros and near-zeros to cut bandwidth and memory traffic for end-to-end acceleration.

Building on penalty-based and compression approaches, DEFT (Density-aware parameter-Efficient Fine Tuning) [208] treats activation density as an explicit training signal during fine-tuning of Transformer MLPs: a lightweight density loss is added to steer intermediate outputs toward zero, typically reducing activation density substantially with minimal impact on accuracy. Extending this explicitly induced principle to generative settings, SQ-DM [52] targets diffusion models via temporal sparsity, coupling aggressive 4-bit activation quantization with time-step-aware gating/routing; sparse and dense channels are then directed to specialized compute units, translating induced zeros and near-zeros into end-to-end speed and energy gains. Collectively, these methods are classified as explicitly induced because the zero pattern is shaped by external objectives, losses, schedules, or quantization policies rather than being a fixed consequence of the activation function itself.

Returning to the previously mentioned FATReLU, while its ‘static’ approach is in the inherent sparsity category, the FATReLU in its ‘dynamic’ form is better understood as a hybrid of the two categories. It consists of an inherent component, which is the thresholded activation that zeroes all responses below a per-layer level T_ℓ and an explicitly induced component, wherein the threshold set is not fixed but rather scheduled or learned under external objectives and accompanied by brief fine-tuning, along with explicit regularization during adaptation (L1, square-Hoyer) so that weights reorganize around progressively stricter gates. Such external objectives are target activation density, accuracy-loss tolerance or latency/energy budgets. Operationally, thresholds are raised per layer via a simple curriculum or a sensitivity analysis that selects the largest T_ℓ consistent with a small accuracy-degradation bound; short recovery phases at each step preserve performance, while the activation distribution shifts toward higher sparsity. Because the attained density is governed by this optimization loop rather than by the nonlinearity alone, dynamic FATReLU is a hybrid mechanism: an inherent thresholded gate operated by an induced controller.

Extending this hybrid pattern, ReLUfication pipelines [222] and SAFS exemplify mixed methods that couple inherent and induced sparsity. In ReLUfication, GELU/Swish are replaced with ReLU to establish an inherently sparse baseline, then a staged L1 schedule and FATReLU-style dynamic thresholds are applied to progressively prune weaker activations—achieving very high sparsity in large language models without performance loss. In parallel, SAFS [157] performs AutoML-driven activation search, jointly evolving piecewise-linear activation functions and training hyperparameters to maximize sparsity in already weight-pruned networks.

This dichotomy distinguishes sparsity arising from the activation rule from sparsity treated as an explicit optimization objective. In practice, the two are frequently combined (e.g., sparse activations followed by density-aware fine-tuning) to achieve higher, better-controlled sparsity that guides methodological choices and hardware co-design.

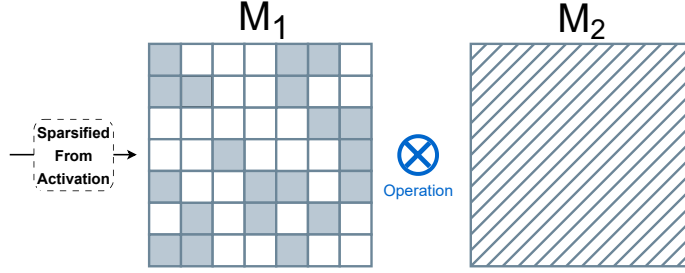


Figure 15: Operations with sparsified activations from the previous layer.

Interestingly, the sparsification opportunities at the activation stage can be much more significant than those in weight pruning. This contrast is readily quantified by a layer-level analysis. Consider the second 3×3 convolution in the Inception-V3 initial block (stem), the illustrative case used by Georgiadis [66]. That layer consumes a $149 \times 149 \times 32$ activation tensor and emits a $147 \times 147 \times 32$ one, summing up to an amount of 1.401.920 total input-plus output activations. In reality, its kernel holds only

$$32 \times 32 \times 3 \times 3 = 9216$$

parameters. Hence the activation volume exceeds the parameter volume by a factor of

$$\frac{1401920}{9216} \approx 1.52 \times 10^2 (\approx 150\times).$$

Even in single-precision,

$$\frac{1401920 \times 4\text{B}}{1048576} \approx 5.6\text{MB}, \quad \frac{9216 \times 4\text{B}}{1024} \approx 36\text{KB}.$$

This means $\sim 5.6\text{MB}$ of activations versus merely $\sim 36\text{kB}$ of weights. Such a two-orders-of-magnitude disparity confirms that aggressively sparsifying activations can yield far greater memory bandwidth and compute saving than further pruning the already compact weight tensor for this, and many other, CNN layers.

In an analogous way as in the case of weight pruning, a sparse tensor is entering the layer of a DNN because of the sparsification that took place at the output of the previous layer. In this case, the operation involved is determined by the following layer. If the layer is Fully Connected, then the operation is again **SpMM**. If the layer is a convolutional layer, the operation is a sparse convolution with the sparsity resident at the input rather than the weights. Figure 15 depicts the process. The \mathbf{M}_1 matrix represents a sparse activation tensor resulting from a previous layer’s non-linearity (e.g., ReLU, FATReLU, or k -WTA), where zero entries are omitted. In both cases, the sparsity resides in the input (\mathbf{M}_1) rather than the weights (\mathbf{M}_2), which remain dense. Note that the operation for computing the weight gradients for layers with sparse inputs, during the backward pass, can also be considered as a sparse operation, leading to potential benefits during training.

Due to the ephemeral nature of sparsity in this case, sparse tensors need to be generated in each inference example (as opposed to weight pruning, where sparsified weights are inserted in the inference engine at deployment time). This is a very important feature of this type of sparsity that greatly influences the optimization opportunities and process. Performance engineers need to consider the efficacy of the process to manage this form of sparsity, i.e., there is no luxury in heavy-weight approaches to store the sparse tensor. On the contrary, on the fly, effective and lightweight schemes need to be employed.

3.4 Sparse Attention

Sparse attention modifies the dense attention operation shown in Equation 15, by introducing a binary mask $\mathbf{M} \in \{0, 1\}^{N \times N}$, which selectively zeroes out certain entries in the attention matrix.

The sparse attention operation is then implemented as:

$$\text{Attention}_{\text{sparse}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}_{\text{nz}} \left(\frac{\mathbf{M} \odot \mathbf{Q}\mathbf{K}^T}{\sqrt{d}} \right) \mathbf{V} \quad (21)$$

where \odot denotes element-wise multiplication that is implemented by the Sampled Dense-Dense Matrix Multiplication (SDDMM). softmax_{nz} is the softmax operation defined in Section 2.1.1 applied only to the nonzero elements that arise from SDDMM. The outcome is then multiplied by \mathbf{V} using an SpMM kernel. Note that, formally, this can be defined similarly to the masked attention mechanism discussed in Section 2.1.5, with $-\infty$ being the outcome of the operation in the zero mask elements. Then, we can apply the standard softmax function to the entire outcome. Sparse attention is a special case of masked attention, however, the typical masked attention relies on dense operations.

In sparse attention, some tokens do not interact directly, which can affect gradient propagation. This should be taken into account when designing the mask pattern. Obviously there is the usual performance vs accuracy trade-off, so caution is needed for choosing the proper mask pattern (usually a combination of more than one pattern is used). To systematically understand and compare alternative approaches to produce the sparse attention mask, we identify a first distinction, as to whether the mask is dependent on the model it will be applied to. We distinguish between **predetermined** and **learnable** masks. Predetermined masks are designed manually and remain unchanged regardless of the input or training process. This means that the mask is defined before training and remains fixed. In contrast, learnable masks are flexible and change based on training data or input content. This category is further divided into two main approaches: those where the mask is learned during training (train-time learned), and those where the mask is generated at inference time based on the input (input-dependent). This taxonomy is shown in Figure 16.

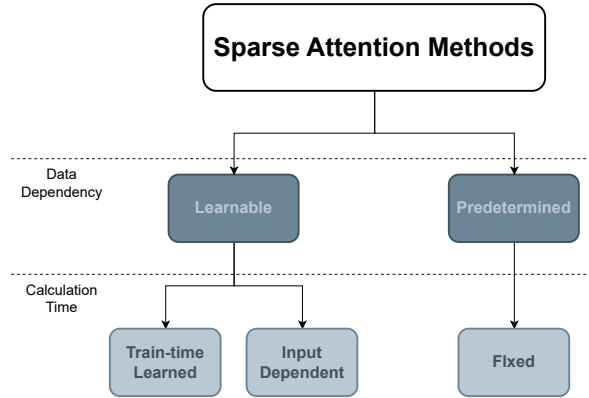


Figure 16: Taxonomy of sparse attention methods, organized by whether the sparsity pattern is fixed or dynamic, and when it is determined (before training, during training, or at inference).

The most common predetermined patterns include structured approaches such as local attention, which confines attention computations to tokens within a fixed-size window. This significantly reduces complexity from $O(N^2)$ to $O(\text{window_size}^2)$, where $\text{window_size} \ll N$. Blockwise patterns [196; 191] represent a practical example of fixed patterns by dividing the input sequence into fixed-size chunks and computing attention only within these local blocks [24]. Another fixed pattern method uses strided (or dilated) attention [24; 12], where tokens attend to other tokens at fixed intervals, capturing a broader context efficiently. Combined patterns [150], utilized by architectures such as Longformer and BigBird [265], integrate local attention with strategically placed global tokens to blend local context with selective global information. Additionally, some methods introduce random attention patterns, where each token is allowed to attend to a randomly selected subset of other tokens. While these patterns are technically produced for each instance, they are considered predetermined because they do not depend on the data or model dynamics during training or inference. This randomness serves to increase connectivity and mitigate information bottlenecks, but the sampling itself is independent of token content. Random attention is classified within the predetermined category due to its fixed structural pattern and data-agnostic design. Nevertheless, implementation-wise, the ad hoc computation of the mask hinders any code optimization approaches that could be applied considering the structure of the mask. Figure 17 illustrates examples of the most common predetermined attention patterns and their combinations, including sliding window, global, dilated, and random attention.

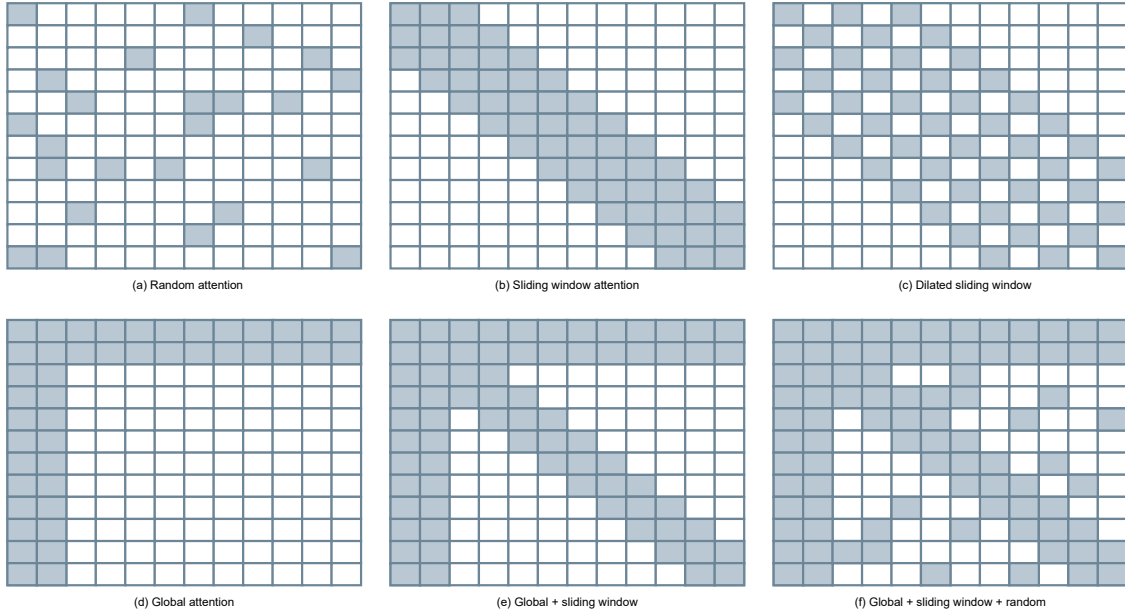


Figure 17: Examples of fixed sparse attention patterns. Top row: (a) Random, (b) Sliding Window, (c) Dilated Sliding Window. Bottom row: (d) Global, (e) Global + Sliding Window, (f) Global + Sliding Window + Random attention.

Learnable sparse attention [238; 119; 158] patterns adapt during training or inference, allowing the model to focus on the most relevant parts of the sequence. These masks are not predetermined, they instead emerge from training dynamics or are constructed on-the-fly during inference. This category can be further divided into two subtypes, based on when the mask is calculated.

The first subtype includes methods that learn the sparse attention mask during training, after which the mask remains fixed during inference. Examples of such approaches include the Routing Transformers [206], which clusters tokens into groups based on learned relationships to determine sparse attention patterns, and Layer-wise Attention Pruning [218], which learns which attention heads are redundant and removes them permanently. Some adaptive sparse attention models also fall into this category, learning token-level importance scores during training and applying the resulting mask uniformly at inference. Hybrid models, such as Longformer, incorporate both fixed and learnable components—combining local attention, global tokens, and learnable sparse patterns to balance efficiency with contextual awareness.

The second category of learnable sparse attention methods involves computing the mask separately for each instance during inference. This includes methods like SAAP [168] and SeerAttention-R [65]. The first method uses learned indexing networks to generate top- k sparse attention patterns per query, with attention computed only over the selected keys using a standard softmax whereas the second dynamically constructs sparse attention masks by aligning with the attention structure of a dense teacher model for each input.

Another approach to achieving sparse attention goes beyond explicitly calculating a sparse matrix \mathbf{M} and instead focuses on approximating the attention matrix using various techniques, such as low-rank approximations or kernel functions. For example methods such as Linformer [247], and Performer [25], project keys and queries into lower-dimensional representations, reducing computational complexity to linear order. Performer and Random Feature Attention [193] specifically employ kernel methods to approximate the softmax operation, drastically reducing both computational and memory requirements. These approaches collectively enable efficient scaling of transformer models and allow handling significantly longer sequences without substantial computational overhead.

Sparse attention methods also include recurrence-based models, such as Transformer-XL [31]. These models process input sequences by splitting them into smaller segments and incorporating recurrent mechanisms,

which maintain memory of previous segments, allowing models to retain information over longer distances. Additionally, downsampling techniques like Perceiver [108] and Funnel Transformers [32] reduce the sequence length progressively by applying pooling operations, resulting in significant computational savings. Adaptive sparsity models, such as Switch Transformers [55], dynamically activate subsets of parameters based on the specific input context, further optimizing computational efficiency.

3.5 Combined sparsity

One natural question that arises is whether combining the various forms of sparsity discussed in this section would be of additional benefit. For example, one could prune the trainable weights of a GNN [194; 78] or utilize both activation and graph sparsity [178; 175]. Although this is a clearly interesting field that can potentially lead to challenging accuracy-performance tradeoffs, and bring in the game additional sparse kernels like SpGEMM (sparse matrix by sparse matrix multiplication) [267], we find that the research in the field is currently limited, and thus leave further investigation for future study.

4 Sparse computational kernels for DNN inference

The performance of sparse computational kernels is decisive for the performance of DNN inference itself, and it is no surprise that their optimization has attracted vivid interest from vendors and researchers. In this section we review the state of the art of the sparse kernels involved in DNN inference, namely Sparse Matrix-Matrix Multiplication (SpMM), Sampled Matrix-Matrix Multiplication (SDDMM), Sparse Matrix-Vector Multiplication (SpMV) and Sparse Convolution. Common performance features of these kernels is their irregularity that leads to imbalance and memory bottlenecks. However, different and subtle instantiations of these generic performance bottlenecks may come forward depending on the kernel, matrix and architecture. The vast majority of proposed kernels are optimized for GPU execution, reflecting the dominant role of GPUs in high-throughput inference workloads. In the following paragraphs we present relevant work for each kernel.

4.1 SpMM and SDDMM

As analyzed in Section 2.3, SpMM is a ubiquitous operation in sparse inference resident in all sources of sparsity. SDDMM on the other hand is used together with SpMM in GAT GNNs and sparse attention. Since researchers that work on SDDMM also work with SpMM, we include both kernels together in our analysis. The main performance challenges addressed for these kernels is the irregularity in the access of the dense vector(s) that prohibits the utilization of existing data locality, and load balance, again an artifact of sparsity. In Section 7 we provide evaluation results for these kernels in GPUs and CPUs where we validate that the performance of these kernels in modern compute devices is very low.

4.1.1 GPU implementations

Several GPU-based SpMM and SDDMM approaches employ code-level optimizations such as tiling, row splitting or merging, and other low-level transformations to improve memory access patterns in line with GPU optimization guidelines (e.g., effective use of shared memory and memory coalescing). These techniques enhance memory locality and workload balance during execution. In addition, reordering the sparse matrix during preprocessing is a commonly used strategy to further improve locality. ASpT [96] applies 2D tiling and column reordering to expose dense tiles and improve memory coalescing in both SpMM and SDDMM. ASpT-RR [113] extends this by grouping rows based on access patterns and applying reordering to further enhance data locality. RoDe [189] handles big rows in SpMM and SDDMM by splitting rows into regular and residual parts, enabling pipelined execution across GPU sub-blocks to improve load balance. RS-SpMM [95] partitions matrices into heavy and light segments and applies thread coarsening (i.e., increases the load per thread) and shared memory optimizations to each segment. GNNAdvisor [248] leverages neighbor partitioning, dimension partitioning for large embedding sizes, and warp-aware memory customizations to optimize workload balance, embedding-dimension parallelism, and memory access. Merge-SpMM [263] addresses SpMM through

a merge-based row-splitting strategy guided by simple heuristics, combining thread- and instruction-level parallelism to hide latency and maintain balanced execution.

Many works opt for low-level GPU optimizations, such as memory alignment, warp scheduling, and caching, to improve workload balance. Sputnik [63] introduced **SpMM** and **SDDMM** kernel designs based on 1D subwarp tiling, incorporating reverse offset memory alignment (ROMA) to align sparse rows with vector memory instructions and a row swizzle technique to enhance load balancing across threads. GE-SpMM [103] proposed a coalesced row caching mechanism along with coarse-grained warp merging, which improves memory access patterns and data reuse among warps by caching contiguous row segments in shared memory and grouping warps to reduce control divergence in CSR-based implementations. More recently, GE-SpMM was incorporated into the dgSPARSE library [39] and was extended to provide an **SDDMM** implementation, with similar optimizations. The **SpMM** implementation evaluated in Section 7 under the name dgSPARSE is based on the GE-SpMM kernel. MergePath-SpMM [216] aims in load balancing the sparse computation through nonzero-splitting and proposes a merge-path algorithm to explicitly track assigned matrix rows and avoid expensive atomic operations for complete row updates.

Existing approaches also utilize NVIDIA GPU Tensor Cores – specialized hardware units designed to accelerate matrix operations using high-throughput, mixed-precision fused multiply-add instructions. These approaches adopt custom sparse formats and data restructuring techniques to expose denser submatrices, enabling efficient mapping of sparse workloads onto Tensor Cores and maximizing hardware utilization. Acc-SpMM [274] introduces the BitTCF compression format and data-affinity-based reordering in order to enhance the density of the blocks and pairs that with a high-throughput pipeline aiming to hide memory latency by overlapping memory access and sparsity-aware load balancing. HC-SpMM [140] uses hybrid core selection to dispatch dense submatrices to Tensor Cores and sparse ones to CUDA cores, employs kernel fusion to reduce kernel launch overhead and improve data reuse, and applies layout reformatting to expose denser segments suitable for Tensor Core processing. Using the same logic, HR-SpMM [246] splits sparse computation by assigning short rows on regular CUDA cores and long rows on Tensor cores, combining load balancing and good hardware utilization. Magicube [138] targets **SpMM** and **SDDMM** with the SR-BCRS format for 1D block sparsity and performs register-level online transposition of dense tiles to support low-precision Tensor Core execution. VectorSparse [22] also proposes 1D-blocking with column vector sparse encoding and supports single and half precision implementations for structured and unstructured sparsity. DTC-SpMM [51] proposes the ME-TCF format and integrates general matrix reordering, automatic code generation, and low-level kernel tuning. Tiled-CSR [261] presents a new compression format designed for unstructured sparsity, complemented by row shuffling, adaptive memory access, and 3D-grid tiling to enable fine-grained scheduling on Tensor Cores. SMaT [185] applies matrix permutation to create dense blocks for TCs and offers a BCSR implementation that overlaps data movement with computation. FastSpMM [241] proposes a new Compressed Sparse Tile (CST) format, virtual-row-based merging, and load-aware mapping, combined with runtime kernel optimizations like software pipelining and data prefetching, to overcome format conversion overheads, load imbalance, and irregular memory access. Unlike the previous approaches, Flash-LLM [256] introduces a Load-as-Sparse, Compute-as-Dense approach and employs a tiled-CSL sparse format to mitigate memory bandwidth bottlenecks which is then unpacked to a dense matrix to fully utilize Tensor Cores.

Another line of work utilizes the NVIDIA Sparse Tensor Cores, which support a fixed 2:4 sparsity pattern – requiring at least two zeros in every group of four elements. Several approaches focus on encoding sparse data into 2:4-compatible submatrices so that semi-structured sparsity can be efficiently executed on these specialized hardware units. NM-SpMM [161] supports arbitrary vector-wise N:M sparsity without relying on hardware-specific features, using hierarchical blocking to improve locality across memory hierarchies and applying latency-hiding strategies. nmSPARSE [143] extends N:M sparsity to vector-wise and block-wise formats, eliminating shared memory bank conflicts through computation reorganization. Jigsaw [270] reorganizes sparse matrices into 2:4 patterns using a hierarchical storage format and column reordering, allowing efficient mapping to Sparse Tensor Cores. VENOM [15] generalizes N:M sparsity with a flexible V:N:M format, combining column-wise pruning and block sparsity, and maps these patterns onto Sparse Tensor Cores. Marlin [60] is also implemented for sparse tensor cores whilst introducing quantization and mixed precision support.

Other works address the problem of tuning the sparse kernels over the large optimization space and propose code generation, heuristic tuning, and runtime adaptation to handle the diversity of sparsity patterns and hardware targets. EC-SpMM [144] and LO-SpMM [145] adopt code generation approaches that apply 2D tiling and matrix reordering to optimize for pruned deep learning models. LO-SpMM extends this by introducing hierarchical tiling, constraint-based search space reduction to decrease the tuning time, and proxy-based performance evaluation. SparseRT [249] uses an inspector-executor framework where sparse matrix data are stored in instruction cache and embedded directly in instruction operands, enabling efficient load balancing through compile-time specialization. DA-SpMM [30] proposes a heuristic-driven design that selects appropriate kernel strategies based on input characteristics, using orthogonal design principles to adapt to input dynamics at runtime.

Several works integrate fused kernel designs and communication-aware scheduling to optimize sparse operations especially in GNN pipelines. fuseGNN [21] introduces a high-level message passing framework with fused SDDMM-SpMM execution, structured into five stages with register blocking to minimize kernel launch overhead and intermediate memory usage. GNNOne [73] combines fused computation with inter-thread communication and scheduling strategies by organizing threads into groups within a warp and optimizing host-to-device data transfer, aiming to reduce synchronization and improve throughput during attention computation. GNNPilot [101] enhances performance by combining neighbor packing for load balancing in sparse graphs, bin packing (BIN_CSR format) for improved data locality in dense graphs, dynamic parallelization strategies, and a row-panel-based kernel fusion technique. FeatGraph [100] extends the TVM compiler [19] with GNN-specific optimizations, including feature-dimension tiling, hybrid graph partitioning and user-defined scheduling templates, to efficiently fuse and optimize graph traversal with feature computation. Fused3s [141] proposes one kernel that combines the three basic operations of sparse attention (SDDMM, Softmax, SpMM) in one, while utilizing CUDA Tensor Cores and minimizing data movement.

Finally, regarding **vendor** provided libraries, NVIDIA’s cuSPARSE offers optimized implementations for many sparse operations including SpMM and SDDMM, supporting different formats and access patterns. rocSPARSE [203] is a HIP-based library in the AMD-ROCm ecosystem that provides optimized sparse linear algebra routines for AMD GPUs.

4.1.2 CPU implementations

To our knowledge there are very few works that propose sparse kernels needed in deep learning inference for CPU architectures. TACO [120] is a library that provides optimized compiler generated kernels for sparse linear algebra including SpMM and SDDMM. Intel’s MKL [240] is a widely used vendor library that provides optimized implementations for sparse kernels like SpMM, SpMV, SpGEMM but does not support SDDMM. Similarly, AMD’s AOCL-Sparse [3] provides SpMV and SpMM implementations; however, it lacks SDDMM support. From the aforementioned papers, ASpT and FeatGraph also provide CPU support. SparseDNN [250] introduces an x86 code generator that combines row reordering and tiling to optimize direct convolution on CPUs, tuned for cache utilization and SIMD execution. FusedMM [198] fuses SDDMM and SpMM into a single multi-stage kernel with register blocking to minimize memory traffic and improve locality. J-stream [124] proposes a tiling strategy based on matrix signatures and a streaming scheme along the output dimension to minimize memory movement.

4.1.3 General observations

Our analysis reveals that optimization strategies often align closely with the target domain of each paper. Methods targeting graph attention, including GNNOne, FeatGraph, fuseGNN, fused3s and GNNPilot, typically propose fused execution pipelines that combine SDDMM and SpMM into a single optimized workflow with tailored dataflow and scheduling strategies. A distinct category of works – such as ASpT, ASpT-RR, RS-SpMM, DA-SpMM, SMaT, FusedMM, and J-stream – do not focus on a specific machine learning domain or sparsity type. Instead, they use generic matrices from the SuiteSparse collection and employ classical optimizations like 2D tiling, reordering, and shuffling, which are well-suited for the large, irregular structures commonly found in these datasets. In contrast, works addressing attention sparsity – such as Magicube, Sputnik, VectorSparse, and Tiled-CSR – primarily rely on low-level GPU optimizations, including memory

alignment, instruction-level tuning, and register-level transpositions, to improve throughput under structured or unstructured sparsity.

4.2 SpMV

The Sparse Matrix Vector Multiplication (SpMV) kernel has been widely studied in the HPC field. SpMV is a memory bound kernel and at the same time dominates the execution time of large system solvers that cover a wide range of applications like graph processing and scientific computing. Many different approaches [121; 152; 44] have addressed the bottlenecks of SpMV for both CPU and GPU architectures. These approaches are best suited for large matrices with a high unstructured sparsity of 99% and therefore do not suit the smaller, denser and more structured matrices that appear in LLMs for example. In deep learning, SpMV is typically found when the network layers process the input sequentially, one token at a time, i.e., SpMV is the primary computational kernel when running sparse RNNs and LSTMs. As observed by Hoefer et al. in [93], most approaches targeting SpMV for recurrent neural network inference rely on specialized hardware accelerators that provide explicit support for sparse deep learning computations. In more modern network architectures SpMV can appear in the sequential, autoregressive decoding stage of Transformers when the batch size is one. To accelerate LLM inference, EC-SpMV [146] proposes a novel approach that uses hierarchical block extraction and a compressed sparse format EC-CSR with delta indexing to enhance data locality and reduce storage overhead in GPUs. From the aforementioned papers, RS-SpMM and MergePath-SpMM also support an SpMV kernel implementation.

4.3 Sparse Convolution

As discussed in Section 2.3.1, sparse convolution is a key computational kernel in CNNs, that are particularly important in computer vision tasks where sparsity arises from structured inputs (e.g., 3D point clouds) or pruning techniques used to accelerate inference. Many approaches have been proposed that cover the different sparse convolutions in both CPU and GPU architectures. TACO-UCF [253] is an extension of the TACO library, offering a generalized approach that tackles all the different types of sparse convolutions. Escoin [20] proposes a direct approach for filter sparse convolution that avoids the im2col lowering of the kernel into SpMM and tries to maximize data reuse in GPUs by exploiting the inherent spatial locality of convolution operations. XNNPACK [46] and SparseDNN also provide im2col sparse convolution implementations. In [127] Kurtz et al. propose a CPU-tailored sparse convolution algorithm that exploits activation sparsity by efficiently integrating run-time SIMD compression of sparse input with low-level kernel optimizations, including vectorization, memory access pipelining, and data blocking. Lastly, several papers propose optimized implementations of submanifold sparse convolution and are discussed in detail in Appendix B.

Table 1: Summary of SpMM, SDDMM and SpConv implementations discussed in Section 4

Name	Platform	Kernels	Dataset	Sparsity Type	Optimization	Domain
cuSPARSE [181]	GPU	SpMM, SDDMM	-	unstructured	Closed-source NVIDIA library	-
rocSPARSE [203]	GPU (AMD)	SpMM, SDDMM	-	unstructured	Open-source AMD library (GPU)	-
ASPT [96]	GPU, CPU	SpMM, SDDMM	SuiteSparse	unstructured	2D tiling, column reordering	HPC
ASPT-RR [113]	GPU	SpMM, SDDMM	SuiteSparse	unstructured	Clustering-based row reordering	HPC
deSPARSE [103; 39]	GPU	SpMM, SDDMM	SuiteSparse	unstructured	CSR-based, coalesced row caching, coarse-grained warp merging	GNN
Sputnik [63]	GPU	SpMM, SDDMM	DLMC	unstructured	1D subwarp tiling, ROMA, row swizzle	CNN, Transformers
EC-SpMM [144]	GPU	SpMM	custom pruned weights	unstructured	Codegen, 2D tiling, matrix reordering	DNN
LO-SpMM [145]	GPU	SpMM	custom pruned weights	unstructured	Codegen, hierarchical tiling, reordering, prefetching	DNN
SparsRT [249]	GPU	SpMM, SpConv	DLMC	unstructured	Codegen, inspector-executor framework, load balancing	CNN, Transformers
RoDe [189]	GPU	SpMM, SDDMM	SuiteSparse, DLMC	unstructured	Row splitting (regular/residual), pipelined execution, ROMA	HPC, Transformers, CNN
DTG-SpMM [51]	GPU (TC)	SpMM	custom pruned weights, SuiteSparse	unstructured	ME-TCF format, reordering, codegen	GNN
Acc-SpMM [274]	GPU (TC)	SpMM	custom pruned weights, SuiteSparse	unstructured	BitTCF format, data-affinity reordering, load balancing	GNN
RS-SpMM [95]	GPU	SpMM, SpMV	SuiteSparse	unstructured	Split matrix (heavy/light segments), streaming, thread coarsening	GNN
VectorSparse [22]	GPU (TC)	SpMM, SDDMM	DLMC, ResNet	(un)structured	1D Octet Tiling, column-vector sparse encoding	HPC
Escin [20]	GPU	SpConv	custom pruned weights, SuiteSparse	unstructured	Direct sparse convolution (avoids in2col)	Transformers, CNN
GNNOne [73]	GPU	SpMM, SDDMM	custom pruned weights, SuiteSparse	unstructured	Fused computations, inter-thread communication, scheduling	GAT
HC-SpMM [140]	GPU	SpMM	SNAP, TUDataset, KONECT	unstructured	Hybrid core selection (TC/CUDA), kernel fusion	GNN
DA-SpMM [30]	GPU	SpMM	SuiteSparse	unstructured	Heuristic-driven algorithm selection, orthogonal design principles	HPC
Merge-SpMM [263]	GPU	SpMM	SuiteSparse	unstructured	Merge-based row-splitting, load balancing	HPC
Magcube [138]	GPU	SpMM	DLMC, LRA transformer benchmark	structured	SR-BCRS format for 1D block sparsity, online transposition	Transformers
fuseGNN [21]	GPU	SpMM, SDDMM	graph dataset	unstructured	Kernel fusion, kernel launch overhead minimization	GAT
FastSpMM [241]	GPU (TC)	SpMM	SuiteSparse	unstructured	CST format, virtual-row-based merging, runtime optimizations	GNN
Flash-LLM [256]	GPU (TC)	SpMM	LLM models	unstructured	Tiled-CSL format, load-as-sparse/compute-as-dense approach	Transformers
SMaT [185]	GPU (TC)	SpMM	SuiteSparse, synthetic banded matrices	unstructured	BCSR format, matrix permutation, computation/communication overlapping	Transformers
Tiled-CSR [261]	GPU (TC)	SpMM	ImageNet	unstructured	SIMD compression, vectorization, pipelining, blocking	CNN
Activation Spconv [127]	CPU	SpConv	SuiteSparse	unstructured	Neighbor/Bin packing (BIN_CSR), dynamic parallelization, kernel fusion	GAT
GNNPilot [101]	GPU	SpMM, SDDMM	graph dataset	unstructured	TVM extension, feature-dimension tiling, hybrid partitioning	HPC
FeatGraph [100]	GPU, CPU	SpMM, SDDMM	Llama (N:M-pruned weight matrices)	semi-structured	Hierarchical blocking, sparsity-aware latency hiding	GNN
NM-SpMM [161]	GPU (STC)	SpMM	graph dataset	unstructured	Neighbor/dimension partitioning, warp-aware memory customization	Transformers, HPC
GNNAdvisor [248]	GPU	SpMM, SDDMM	graph dataset	unstructured	Vector-Wise/Block-Wise N:M formats, bank conflict elimination	Transformers, DNN
nnSPARSE [143]	GPU (STC)	SpMM, SpMV	BERT-Large, OPT, synthetic	semi-structured	V:N:M format, column-wise pruning, Sparse TC	Transformers, DNN
Jigsaw [270]	GPU (STC)	SpMM	DLMC	semi-structured	Hierarchical reorder-aware format, column reordering	GNN
VENOM [15]	GPU (STC)	SpMM	HuggingFace	unstructured	Kernel fusion, memory traffic minimization, locality awareness	-
FusedMM [108]	GPU (STC)	SpMM, SDDMM	SuiteSparse, NetworkRepository	unstructured	Compiler generated kernels, OpenMP directives	CNN
TACO [120]	CPU	SpMM, SDDMM	-	(un)structured	TACO extension for SpConv	-
TACO-UCF [253]	CPU, GPU	SpConv	DLMC, S3DIS [7], SemanticKITTI [11]	(un)structured	Closed-source Intel library	-
MKL [240]	CPU	SpMM	-	unstructured	Open-source AMD library (CPU)	-
AOCL-Sparse [3]	CPU	SpMM	custom pruned matrices	unstructured	x86 codegen, row reordering, z-curve tiling	CNN
SparseDNN [230]	CPU	SpMM, SpConv	ImageNet	unstructured	Im2col sparse convolution, cache prefetching	CNN
XNNPACK [46]	CPU	SpMM, SDDMM	SuiteSparse	unstructured	Tiling based on matrix signatures, memory traffic minimization	HPC
J-stream [124]	CPU	SpMM, SpMV	graph dataset	unstructured	Merge-path algorithm, nonzero-splitting load balancing	GNN
MergePath-SpMM [216]	GPU	SpMM, SpMV	graph dataset	unstructured	MET-TCF based format, Kernel fusion (SDDMM-Softmax-SpMM)	GAT
fused3S [141]	GPU (TC)	SpMM, SDDMM	graph dataset	unstructured	Row splitting (short rows on CCs, long on TCs)	Transformers, HPC
HR-SpMM [246]	GPU (CC, TC)	SpMM	DLMC, SuiteSparse	unstructured	Sparse TC, 4-bit weight quantization, mixed precision	Transformers
Marlin [60]	GPU (STC)	SpMM, SDDMM	integrated in vLLM	semi-structured	EC-CSR format, hierarchical blocking, delta indexing	Transformers
EC-SpMMV [146]	GPU	SpMV	SpaseGPT	unstructured	-	-

5 Datasets for sparse inference research

In the previous section we presented numerous works that involve designing new hand-crafted or generated kernels to speedup sparse neural network inference. To evaluate their results, these works either incorporate their kernels into an existing framework like PyTorch and run a model end-to-end, or isolate parts of the model—for example, a single layer or operation—and test their kernel’s effectiveness specifically on that part of the model. Applying this to the most time consuming parts of the model allows them to generalize their findings to the full network. A critical issue in this process is to identify appropriate data for conducting experiments. Works that evaluate end-to-end performance over a model need to test their work on fully trained sparse models with acceptable accuracy to ensure that the shape and characteristics of the model’s sparsity are representative of real-world scenarios. On the other hand, works that focus on the performance of isolated kernels require internal data from the neural network, such as the sparse weight matrices and activations of particular layers, to test their approach.

In this section we examine the availability and characteristics of data that can support research and experimental assessment of performance engineering methods that target sparse DNNs. We present the characteristics of the DLMC (Deep Learning Matrix Collection) [63] which serves as the standard matrix suite of pruned model layers, custom approaches for model pruning, the availability of sparse models in public repositories, sparsification approaches for semi-structured pruning, graphs that represent DNNs, attention masks and datasets relevant to sparse activation.

5.1 Deep Learning Matrix Collection (DLMC)

The Deep Learning Matrix Collection (DLMC) is a curated dataset of sparse matrices designed specifically for benchmarking and evaluating sparse kernels in deep learning workloads introduced in [63]. Unlike general-purpose collections such as SuiteSparse [35], DLMC consists of matrices extracted directly from the weight tensors of deep neural networks, resulting in sparsity patterns that closely reflect those found in real-world models. The matrices are generated using a variety of unstructured pruning techniques, including magnitude pruning, random pruning, L_0 regularization [159], and variational dropout [173]. The latter two are examples of training-time pruning methods.

The models used are a classic Transformer [236], the foundational architecture for most Natural Language Processing tasks, and a ResNet-50, a widely adopted convolutional neural network in the field of Computer Vision. These two baseline models are well-established and typically sufficient for conducting basic experiments on DNN kernels. Additionally, the dataset creators provide essential metadata and statistics for each matrix, allowing researchers to analyze the underlying data characteristics. DLMC has become the staple benchmark for research on unstructured sparsity [63; 189; 22; 138; 261; 270], serving as a reference point for evaluating sparse kernel performance as it is currently the only structured and publicly available dataset that can be readily used with custom kernels—eliminating the need for researchers to generate weight matrices and perform pruning themselves.

However, DLMC’s utility for modern kernel experiments could be considered more limited due to its age and scope. The models in the collection, such as the original Transformer and ResNet-50, date back to 2019 and 2020, failing to represent the exponential growth in scale seen in contemporary architectures. For instance, the original ‘Attention Is All You Need’ Transformer featured a model dimension of 512, a feed-forward dimension of 2,048, and 8 attention heads—yielding weight matrices with sizes in the order of 106 parameters. In contrast, modern large language models (LLMs) such as LLaMA 3 405B employ a model dimension of 16,384 and a feed-forward dimension of 53,248 across 126 layers, resulting in individual feed-forward matrices containing in the order of 109 parameters. These models therefore operate at scales several orders of magnitude larger, with total parameter counts extending into the hundreds of billions, far beyond the representational range captured in DLMC.

As mentioned above, modern LLMs like Llama 3 [42], Qwen3 [262] and DeepSeek V3 [148] operate on projection and FFN matrices that dwarf those in the original Transformer. Apart from the different size, modern models also integrate techniques like Grouped-Query Attention (GQA) [2], Multi-Head Latent Attention (MLA) [169], MoE routing [217], FlashAttention’s fused I/O-aware kernels [33], rotary/relative embeddings

[225], and speculative decoding [136]. These innovations alter both the shape and execution flow—for example, GQA changes KV matrix dimensions and cache patterns, MLA applies low-rank compressions and runtime up-sampling, and FlashAttention fuses QK^\top , softmax, and matrix multiplication into streaming kernels. This means that we cannot treat inference as a vanilla sequence of dense GEMMs. Current DLMC benchmarks, built around small, sparsified Transformer and ResNet matrices, miss these real-world inference characteristics. If DLMC is to remain relevant for kernel designers and researchers, it must evolve; including sparse, pruned matrices sampled from these mega-scale LLMs and reflecting their grouped attention, compressed latent KV, fused execution, and dynamic sparsity patterns during inference—so that benchmarks match the reality of how state-of-the-art models are actually computed. Furthermore, the dataset lacks matrices from key modern architectures like Diffusion Models, State-Space Models and MLP-Mixers and GNNs.

5.2 Customly pruned layers

Apart from the DLMC dataset, many studies rely on their own custom benchmarks and matrix collections. The methods used to create these diverse datasets can be broadly categorized into three approaches:

1. **Pruned weights from real models:** This is a popular trend, as it provides the most realistic sparsity patterns. Researchers take well-known models like ResNet-50, Transformer, BERT, MobileNet, EfficientNet, and various LLMs (e.g., OPT-30B) and apply specific pruning algorithms to their weight layers. The key benefit of this approach is that the resulting sparse matrices reflect the characteristics of a real-world neural network layer, including the distribution of nonzero values and the overall matrix dimensions. Papers like SparseRT [249], Faster CNNs [190], Escoin [20], and Flash-LLM [256] all follow this strategy. These datasets are not just matrices; they are direct outputs of a model compression process, making them ideal for testing the efficiency of sparse kernels on realistic workloads.
2. **Synthetic matrices:** This approach involves generating matrices from scratch with controlled parameters like size and sparsity level. While not derived from a pruned model, these matrices are often designed to mimic the structural characteristics of DNN layers. For example, EC-SpMM [144] and LO-SpMM [145] create synthetic matrices that are ‘DNN-inspired’, meaning they have shapes and sparsity patterns that are representative of what you would find in deep learning models. This method offers the advantage of reproducibility and control, allowing researchers to isolate variables like matrix dimension, sparsity, and block size to understand their impact on kernel performance. SparseDNN [250] is a clear example of this trend, where matrices are generated with specific sizes and sparsity percentages for systematic testing.
3. **Hybrid and specialized approaches:** Some research combines these methods or focuses on highly specific model types. For instance, SparTA [276] applies structured sparsity to models like BERT and HuBERT, which is a form of pruning that results in a more regular sparsity pattern than the unstructured variety found in DLMC. Similarly, Minuet [264] focuses on sparse 3D CNNs and voxelized data, which introduces a different type of sparsity pattern relevant to 3D vision tasks. This shows a trend toward increasingly specialized datasets tailored to niche applications.

In essence, the DLMC serves as a robust and standardized starting point, but the trend of creating custom datasets allows researchers to push the boundaries of sparse kernel research by exploring new models, algorithms, and sparsity patterns that are not yet covered by public collections. However, the lack of a standardized benchmarking approach may pose a challenge for progress in the field, as it complicates fair and consistent comparisons between sparse kernel implementations. Furthermore, many of these custom benchmarks fail to accurately recreate weight matrices or models that reflect real-world deep learning workloads, thereby limiting the practical relevance and generalizability of their results.

5.3 Sparse models in public repositories

Public repositories such as Neural Magic’s SparseZoo [164] and the Hugging Face Hub [49] can serve as an ideal source of sparse datasets. However, it is worth noting that finding already sparse networks in model repositories is not very common. SparseZoo has historically been the most prominent resource, but as of 2025, it is no longer being actively updated, with the most recent state-of-the-art contribution being Llama3-8B-2:4. By contrast, platforms like Hugging Face Hub host a vast number of pre-trained models, but these are typically dense.

In contrast, the Hugging Face Hub contains a much larger variety of pre-trained models, but dense models are the default: few models explicitly enforce sparsity. Notable exceptions include cases like: `Intel/bert-base-uncased-sparse-1_2`, which employs structured 1:2 sparsity. Also, the DeepSparse Sparse LLMs collection—hosted via Neural Magic’s presence on Hugging Face—includes some models where at least 50% of the weights have been pruned.

One reason for the rarity of sparse models is that many model hubs prioritize broad compatibility and ease of use. Dense models are heavily supported by existing frameworks, hardware, and inference pipelines. Sparse inference runtimes like DeepSparse exist and can leverage pruned models, but their adoption is still limited by compatibility, performance trade-offs, and toolchain complexity.

5.4 Data for semi-structured sparsity

N:M pruning has become a key technique for optimizing large-scale deep learning models, particularly where matrix multiplication operations dominate the computational time. Transformer-based architectures, especially in NLP, are the most common targets due to their heavy reliance on linear layers, with weight matrices from models such as [38], GPT [166], OPT [271], and LLaMA [233] frequently used to construct N:M sparse datasets. Beyond these real-world models, researchers also consider broader model suites, including CNNs like ResNet [86] and VGG [220], object detection models such as MaskRCNN [40], and even GANs [74], to demonstrate the general applicability of pruning methods.

The standard approach for producing N:M sparse models follows a train, prune, finetune workflow: a dense model is first trained to convergence on the target task, its weights are then pruned to satisfy a specified N:M sparsity pattern, and finally the pruned model is fine-tuned or fully retrained to recover any lost accuracy. Research has also focused on moving beyond NVIDIA’s hardware-native 2:4 sparsity pattern (50%) to explore more aggressive, arbitrary N:M ratios, including 2:8 (75%), 2:16 (87.5%), and even greater than 95% sparsity, as investigated in [143], VENOM [15], and NM-SpMM [161]. Additionally, different pruning granularities are being explored, from element-wise to vector-wise and block-wise N:M sparsity, with coarser granularities often yielding more regular memory access and higher performance. To support these experiments, frameworks such as NVIDIA’s Automatic Sparse Pruning (ASP) [171] are frequently extended to handle arbitrary N:M patterns and granularities, enabling systematic evaluation of both sparsity and computational efficiency.

For controlled micro-benchmarking, synthetically generated matrices are often employed, allowing evaluation of kernel efficiency independently of any specific model architecture, as seen in studies like [143] and [15]. We consider this a solid approach, since the actual distribution of nonzero elements in semi-structured sparsity does not influence performance in this case. What actually matters is the size of the matrix and the N, M factors.

5.5 Graphs for GNNs

Evaluating performance on GNNs is rather straightforward. Since both GNN models and their input data are inherently sparse, there is no need for additional modifications such as pruning or retraining/fine-tuning. Moreover, most studies rely on well-established benchmark graph datasets that are already widely used, such as SuiteSparse [35], SNAP [135], OGB [99], TUD [176], allowing models to be run end-to-end and their performance measured directly.

The sizes and densities of the graphs in the most popular datasets vary a lot. Measuring these two quantities is important since different sparse matrix techniques have additional costs for the sparsification of matrices

and show different performance for different input densities. The sizes range from a few tens of nodes for graph classification (in molecular sciences), to a few thousands of nodes for node classification and link prediction (in citation datasets), to hundreds of thousands of nodes in knowledge graphs. In all datasets, the average number of edges per node is rarely above 10, resulting in densities of less than 1% in most cases.

An important factor in graph-based learning tasks is whether the task involves processing a single large graph or many smaller, disjoint graphs:

1. Single large graph scenarios (e.g., citation networks, social networks, web graphs): the dataset is represented as one connected or partially connected graph, with message passing occurring globally across the node space.
2. Many small graph scenarios (e.g., bioinformatics, molecular graphs, protein structures): consist of multiple graphs that serve as independent connected components. Each graph corresponds to a distinct entity (e.g., a molecule), and message passing is restricted within each graph. During inference, these graphs are processed in batches, allowing the model to handle many graphs in parallel. Vectorized computations are still employed, with indexing ensuring message passing remains confined to each respective graph. In benchmark reporting, node and edge counts are typically averaged per graph.

Another key feature in real-world datasets is the distribution of nonzero elements. There are two main cases: a) power-law distribution, where arbitrarily long ‘evil’ rows are present, leading to workload imbalance and irregular memory accesses and b) structured distribution, where the number of nonzero elements per row does not vary significantly.

Some graph-related parameters here are the average and maximum node degree. When their values have small variance (i.e., max degree is relatively small), this indicates that the nonzeros are evenly distributed. For example, from the table of datasets (Table 2), Cora, PubMed, Citeseer, artist, and amazon0505 are considered type 1 (with artist ranking first), while OVCA8 and PROTEINS are considered type 2.

While the input feature size provides a useful overview of the aggregation workload, it does not directly impact the actual computations involved. This is because very large initial feature sizes in a GNN are computationally expensive, so a projection layer (also called a combiner, feature projector, or encoder) is typically used to reduce the input feature dimension to a manageable hidden size before computation. Alternatively, some GNN architectures perform this projection implicitly within the first GNN layer, where the learnable weight matrix simultaneously handles dimensionality reduction and feature transformation. In both cases, the dimensionality relevant for **SpMM** is determined by the projected hidden size, rather than the raw input feature size. Typical hidden dimensions for GNN layers range from 16, 32, 64, 128, 512, and occasionally up to 1024, depending on model design, computational resources, and task complexity. The hidden dimension (embedding size) is treated as a hyperparameter during optimization, often selected through empirical tuning on validation performance, and is then used in the central Sparse Matrix-Matrix multiplication (**SpMM**) operation. This dimension determines the output size of intermediate GNN layers, balancing computational cost with model capacity, and serves as the key parameter for optimizing the **SpMM** kernel. The direct use of sparse, well-defined matrices along with their associated feature dimensions simplifies both performance evaluation and kernel design.

5.6 Attention masks

As discussed in Section 3.4, masks are either predetermined or learnable. In general, the scientific community has increasingly shifted its focus towards learnable and dynamic attention masks rather than predetermined ones. This trend reflects the growing demand for models that can adaptively modulate attention based on input context, achieving better trade-offs between efficiency and accuracy. Despite this growing interest, relatively little research has systematically investigated how these masks impact kernel performance, and consequently, there are few publicly available datasets or ready-to-use masks for testing sparse attention kernels.

Table 2: Graph datasets used in bibliography

Title	Structure	Nodes	Edges	Feature size(initial)	Category	Source
Cora	Single graph	2,708	5,278	1,433 (node)	citation	Planetoid
PubMed	Single graph	19,717	88,648	500 (node)	bioinformatics	Planetoid
Reddit	Single graph	232,965	114,615,892	602 (node)	internet-forum	-
Reddit Binary	Multiple subgraphs	~429.6	~995.5	-	internet-forum	TUDortmund
AmazonProducts	Single graph	1,569,960	264,339,468	200	e-shop	GraphSAINT
ogbn-proteins	Multiple subgraphs	132,534	39,561,252	8 (edge)	bioinformatics	OGB
ogbn-mag	Single graph	1,939,743	21,111,007	128 (node)	citation	OGB
ogbn-arxiv	Single graph	169,343	1,166,243	128 (node)	citation	OGB
com-Youtube	Single graph	1,138,499	2,990,443	-	web-network	SNAP
com-Orkut	Single graph	3,072,441	117,185,083	-	web-network	SNAP
com-Amazon	Single graph	334,863	925,872	-	e-shop	SNAP
Flickr	Single graph	8,925	899,765	-	web-network	GraphSAINT
Harvard	Multiple subgraphs	15,126	824,617	-	citation	-
Citeseer	Single graph	3,327	9,103	3,703 (node)	citation	SNAP
sx-stackoverflow	Single graph	2,601,977	63,497,050	-	internet-forum	SNAP
roadNet-CA	Single graph	1,965,206	2,766,607	-	road-network	SNAP
web-BerkStan	Single graph	685,230	7,600,595	-	citation	SNAP
artist	Single graph	50,515	1,638,396	-	web-network (?)	Kaggle
amazon0505	Single graph	410,236	3,356,824	-	e-shop	SNAP
OVCAR-8H	Multiple subgraphs (200)	46.67	48.70	-	bioinformatics	TUDortmund
PROTEINS	Multiple subgraphs (1,113)	39.1	145.6	3	bioinformatics	TUDortmund
PPI	Multiple subgraphs	2,245.3	61,318.4	-	-	-
ogbl-collab	Single graph	235,868	1,285,465	128 (node)	citation	OGB
ogbl-ppa	Single graph	576,289	30,326,273	58 (node)	bioinformatics	OGB
ogbl-ddi	Single graph	4,267	1,334,889	-	bioinformatics	OGB
ogbn-products	Single graph	2,449,029	61,859,140	100 (node PCA)	e-shop	OGB

Several prominent repositories accompany leading papers, offering either ready-to-use libraries or reference implementations that support dynamic sparse attention during both training and inference. Researchers typically rely on these tools to apply sparse attention schemes and measure model performance. These implementations are often integrated with popular deep learning frameworks such as PyTorch or TensorFlow, facilitating experimentation and model adaptation. For example, SpargeAttn [269] provides a modular framework for dynamically generating sparse masks during inference, using heuristics like cosine similarity and mean pooling. This toolkit allows plug-and-play integration with pre-trained models, enabling sparsity without retraining from scratch. A primary limitation is that achieving competitive accuracy generally requires relatively low sparsity ratios (around 50%). Similarly, MInference [112] integrates dynamic sparse attention into fine-tuning pipelines for large language models, reflecting a trend toward scalable, resource-efficient adaptation of foundation models.

Finally, it is important to note that these activations are typically applied in groups—semi-structured or structured—rather than in an unstructured, element-wise manner. As a result, the exact computations performed can vary depending on the chosen sparsity pattern.

5.7 Sparse Activations

Recent work showcases that sparse activations tend to compress mostly into binary masks that extract which neurons are active (i.e., nonzero) at inference. To the best of our knowledge, no generic tool specifically crafted for revealing activation masks from numerous model architectures and numerous methods that create sparsity exists. That said, workarounds to extract sparse activations are possible. A universal technique is to intercept the outputs of particular layers during the forward pass and apply a threshold to the intermediate activations to generate binary masks. This technique is most suitable for sparsity that is induced by activation functions like ReLU [170] or by learned gate control [222].

In cases where sparsity is spatial or structure-induced at the level of convolutional layers, masks are revealed by finding regions with zero-valued activations. If sparsity is implicitly exposed, such as with the ‘lazy neuron’ effect with transformers [142], the practical solution is profiling activation statistics across different inputs and thresholding their mean or variance. Likewise, in approaches employing regularization or noise injection

[127], one could estimate sparse patterns by looking at activations across different batches and finding stably inactive components. These methods, however, depend on specific prerequisites, namely, access to the pre-trained model or to the training pipeline, including the source code, so that instrumentation during either training or inference can be implemented.

6 State of practice

Deploying a sparse neural network is a complex engineering task that requires careful consideration of performance goals, target hardware, and model architecture. The process of model sparsification—including the selection of appropriate pruning techniques and activation schemes—is intrinsically linked to the deployment strategy. The primary objectives of introducing sparsity are almost always to reduce latency, memory footprint, and energy consumption. Towards these objectives, this section attempts to outline a selection of tools, libraries, and frameworks that support sparsity in DNN inference. It is important to acknowledge that the domain is extremely broad and fast evolving and therefore, this cannot be considered a complete or fixed list. Following Section 3 we divide our analysis into four subsections based on the source of sparsity, in order to determine which frameworks offer support for each category.

6.1 Utilizing sparsity in weights

6.1.1 Unstructured sparsity

Pruning a DNN model is quite straightforward with the support provided by popular frameworks like PyTorch or TensorFlow. Once an unstructured sparse model has been produced, deployment needs to incorporate the proper sparse data structures and optimized kernels. Unlike the case of semi-structured sparsity discussed next, unstructured sparsity has no predictable pattern as the zeros are scattered across the weight tensors. These irregular unstructured sparsity patterns require specifically designed sparse kernels to realize the potential performance gains. However, the specialized kernels provided by researchers are typically highly complex and not easily integrated into the current production-ready deep learning frameworks, which are primarily optimized for dense or highly regular data formats. Therefore the support for unstructured sparsity in deep learning inference is currently limited. In the following paragraphs we split our analysis into three sections: we first cover the mature, optimized kernel implementations in both CPUs and GPUs (see Section 4 for detailed analysis regarding research implementations), and then examine the high-level, end-to-end frameworks that offer comprehensive support for deploying sparsity.

Support for unstructured sparsity in CPUs The nature of unstructured sparsity is best suited for CPUs, yet due to the current dominance of GPUs and their overall performance benefits, the available support for sparsity in CPUs is limited. Key production and research tools in this area include the following:

- While often outperformed by more specialized kernels, vendor libraries provide an easily integrated solution that still offers decent performance gains. Intel’s MKL [240] is a widely used vendor library that provides optimized implementations for sparse kernels like **SpMM**, **SpMV**, **SpGEMM**. The library provides highly optimized implementations that take into account the input matrix’ structure using inspector-executor logic and are based on the CSR storage format, thus making them easily adjustable to any framework. Similarly, ARM performance libraries [6] and AMD’s AOCL-Sparse [3] provide sparse linear algebra support for ARM and AMD CPUs respectively.
- LIBXSMM [89] is an optimized, Just-In-Time (JIT) code generation library for specialized dense and sparse matrix operations, including small matrix multiplications and deep learning primitives like convolutions (with fusions like GEMM + activation).
- Specialized research inference engines (e.g., SparseDNN) propose advanced techniques, such as code generation and optimized scheduling, specifically to accelerate unstructured sparse inference on CPUs. While generally not yet production-ready, these engines serve as valuable resources for studying new sparse algorithms and often provide highly improved prototype kernels.

Support for unstructured sparsity in GPUs Unstructured-sparse GPU acceleration presents a greater challenge because GPUs fundamentally favor massively parallel applications with structured and coalesced memory access. Thus, support is often more research-driven, though several industry-grade options exist:

- NVIDIA cuSPARSE [181] is the standard vendor library for sparse linear algebra on NVIDIA GPUs, offering optimized baseline implementations that are essential for many inference pipelines that rely on unstructured sparsity.
- ROCm stack (AMD) provides rocSPARSE [203] for sparse linear algebra on AMD GPUs.
- Triton (OpenAI) [187] provides a standard for writing custom sparse kernels, allowing developers to produce highly performant CUDA kernels for unstructured sparsity without the need for extensive knowledge of CUDA programming or specific hardware details.

Support for unstructured sparsity in high level frameworks While hardware support for unstructured sparsity is provided by specialized libraries and low-level kernels, high-level frameworks need to offer APIs and runtimes to deploy these techniques effectively:

- PyTorch [192] offers torch.sparse APIs and some semi-structured support, but general-purpose, high-performance unstructured sparse acceleration (especially on GPUs) is limited and often requires custom kernels or external runtimes.
- TensorFlow / Keras [232] provides tf.sparse.SparseTensor and sparse operations, but GPU inference acceleration for unstructured sparse neural network weights is limited in practice. Most TensorFlow production sparse workflows use structured patterns or CPU-specific optimizations.
- The OpenVINO Toolkit [105] offers an end-to-end approach for training, pruning and deploying deep learning models. It supports quantization and handles sparse weights with the Neural Network Compression Framework (NNCF), leveraging Sparse Weight Decompression in its runtime to reduce memory bandwidth requirements on compatible Intel hardware.
- ONNX [186] defines the SparseTensor prototype, but practical, widely-supported unstructured-sparse kernels are limited. The ONNX Runtime has started adding block-sparse kernels and special-case support, but general unstructured-sparse acceleration often requires custom operators or engine-specific extensions.
- DeepSparse (Neural Magic) [163] is an industrial approach for executing unstructured sparse models. DeepSparse implements a proprietary runtime specifically designed to skip zero-computations effectively on x86 architectures (AVX-512/AVX2). It executes unstructured pruned models significantly faster than dense baselines by keeping the active dataset entirely within the CPU caches hierarchy, effectively turning memory-bound workloads into compute-bound ones on commodity hardware.
- Hugging Face Transformers [252] focus primarily on dense computations. Sparse acceleration is enabled via integration with external toolchains (TVM [19], DeepSparse, OpenVINO, or vendor runtimes) rather than native unstructured-sparse kernels.

6.1.2 Semi-structured sparsity

N:M Sparsity Once a model has been pruned to an N:M pattern, deployment becomes primarily a matter of aligning the model’s format and runtime execution with the capabilities of the target hardware. While training and fine-tuning determine the quality and stability of the sparse model, deployment determines whether that sparsity actually translates into measurable efficiency gains.

Modern GPUs, including NVIDIA’s Ampere and Hopper architectures as well as AMD’s RDNA 4 generation, are currently the main execution targets for N:M sparse models. NVIDIA platforms provide native support for 2:4 sparsity through Sparse Tensor Cores, while AMD supports 4:2 structured sparsity. The approaches

are practically the same. On such hardware, the runtime recognition of fixed patterns allows execution units to skip zero multiplications efficiently, achieving up to a $2\times$ improvement in throughput for matrix multiplications and roughly a 50% reduction in memory footprint without additional software overhead.

In practice, deployment requires exporting the model with its block masks or compressed sparse representation intact, often using framework-level support such as PyTorch’s `torch.sparse` utilities or NVIDIA’s Automatic Sparse Pruning (ASP) extensions. The key step is ensuring that the sparse format used during training matches the runtime format expected by the inference engine, typically a compressed layout optimized for the specific 2:4 or 4:2 pattern supported by the target GPU as discussed in Section 2.3.2. For research-oriented or custom accelerators, alternative ratios like 2:8 or 2:16 can be used, provided that corresponding sparse kernels are implemented in CUDA, Triton, or specialized runtime backends.

Standard CPUs lack specific support for this kind of sparsity. Consequently, when an N:M model is deployed on a CPU, the runtime must handle it either as unstructured sparsity with the generic representations and kernels of this case, or as dense computations that can utilize the resident SIMD support. Therefore, outside of aforementioned GPUs environments, N:M sparsity currently serves primarily as a storage and memory bandwidth optimization rather than a computational accelerator.

Support for N:M sparsity in GPUs

- TensorRT [182] (and TensorRT-LLM [183]) is NVIDIA’s high-performance inference optimizer. The TensorRT Model Optimizer includes modules like `modelopt.torch.sparsity` that enable sparsification of PyTorch models into the 2:4 N:M format and other patterns. The resulting optimized model is then run by TensorRT-LLM which uses highly-fused CUDA kernels to leverage the Sparse Tensor Cores for peak throughput.
- Core PyTorch APIs lack native N:M kernel acceleration. However, high-performance sparse workflows are enabled through extensions like NVIDIA’s TensorRT Model Optimizer (which works as a PyTorch extension) or `torch.ao` (PyTorch’s Acceleration Optimizer), which is building towards a unified sparse inference framework. The common workflow involves training and pruning in PyTorch, then converting and optimizing with TensorRT tools.
- vLLM [129] is a highly efficient LLM serving engine. It is not natively built around N:M sparsity like TensorRT-LLM. However, there exist specialized versions or integrations. For instance, Neural Magic’s `nm-vllm` package [165] extends vLLM with custom CUDA kernels to support sparse models (including N:M pruned models) and deliver acceleration, combining vLLM’s efficient attention mechanisms (PagedAttention) with sparse computation.

Block sparsity Block sparsity splits weight matrices into fixed-size blocks and prunes whole blocks, preserving local structure and improving memory locality compared to unstructured sparsity. A common workflow is to apply a block mask, and convert the masked weights to a Block Sparse Row (BSR) layout such as PyTorch’s `Tensor.to_sparse_bsr()`, which stores only nonzero blocks and their indices. During inference, frameworks with BSR-aware kernels can execute these weights efficiently, either via specialized block-sparse kernels (i.e., DeepSpeed’s Sparse Attention Kernels [36]) or by breaking the work into many small dense matrix multiplications.

6.1.3 Structured sparsity

Structured sparsity offers the simplest deployment pathway. While the pruning procedure itself may be complex, deployment is trivial: after pruning, the model is restructured into a smaller dense network. For example, pruning 128 out of 512 convolutional filters produces a new layer with 384 filters that operates like any standard dense layer. From an engineering standpoint, structured sparsity is universally compatible. The resulting model can be deployed on any hardware platform, CPUs, GPUs, TPUs, mobile, or edge devices, and runs without modification in standard inference frameworks such as PyTorch, TensorFlow, or ONNX runtime. Structured sparsity is widely used when deploying models on resource-constrained devices or whenever a straightforward, low-effort performance improvement is needed without altering the inference pipeline.

6.2 Activation sparsity

Since the zero elements in activation sparsity are input-dependent, skipping zero operations must occur dynamically at runtime. This typically requires hardware or kernels capable of detecting and bypassing inactive neurons on the fly, a capability available mainly in specialized ASICs and FPGA accelerators rather than general-purpose CPUs or GPUs. Nevertheless, recent research has shown that dynamic activation sparsity can still be partially exploited on standard hardware. Prototype systems demonstrate that early CPU-oriented sparse-convolution libraries have achieved measurable speedups by skipping zero activations [127], while recent work on large transformers (e.g., ReLU-based LLM variants [170], activation-predictive pruning, and lightweight sparsity predictors [156; 219; 273]) has shown that dynamic sparsity patterns can be estimated or enforced well enough to accelerate inference with negligible accuracy impact. Although these approaches are not yet mainstream or universally supported, they indicate a growing trajectory toward practical activation-sparsity-aware inference on commodity hardware.

6.3 Attention sparsity

In Transformer models, sparsity can be introduced into the attention mechanism to reduce its quadratic complexity. Architectures like Longformer [12] and BigBird [266] use structured attention patterns (e.g., sliding-window, global tokens, random blocks) to approximate dense attention efficiently. These models are natively supported in Hugging Face Transformers [252] and can be deployed with minimal changes. For high-performance inference, tools like DeepSpeed [4] (with Triton-based block-sparse kernels [36]), xFormers [134] and vLLM offer optimized support for various sparsity patterns. These libraries reduce memory usage and latency, enabling scalable deployment of sparse attention models on modern hardware.

6.4 GNN sparsity

In practice, deploying GNNs is heavily dependent on specialized frameworks such as PyTorch Geometric (PyG) [56] and Deep Graph Library (DGL) [244], which provide optimized CUDA and CPU kernels tailored for graph operations. From an engineering perspective, the key responsibilities are ensuring that the production system uses these optimized kernels and that graph data is represented efficiently in a sparse format. However, there are trade-offs: not all GNN layers have fused kernels, and for very large graphs or latency-critical systems, you may need custom tuning (e.g., JIT compilation, multi-core runtime systems, or distributed GNN frameworks).

7 Evaluation

In this section, we evaluate the performance of the previously discussed **SpMM** and **SDDMM** kernels, focusing exclusively on those that are both published and open-source.⁴ We focus on those kernels as they currently dominate efforts as discussed in Section 4. The interested reader can find a recent evaluation for **SpMV** in HPC setups in [177]. The primary goal of this evaluation is to benchmark all kernels within a unified experimental framework, enabling a fair and consistent comparison of their performance. Beyond that, we aim to explore the computational capabilities of both GPU and CPU platforms by examining the extent to which each kernel approaches peak performance on its respective architecture. Another objective is to highlight the performance variability across different types of matrix datasets; particularly comparing graph-based matrices from the SuiteSparse collection against dense-like matrices from the DLME repository. We also investigate how structural properties of the matrices influence kernel performance, seeking to identify any strong correlations between matrix characteristics and computational efficiency. Lastly, this section reflects on the practical aspects of the evaluation process itself. We assess the setup complexity involved in running each kernel, noting that some implementations require minimal effort, while others demand more extensive preparation and tuning.

⁴The repository, including all kernel implementations and benchmarking scripts, is available at https://github.com/pmpakos/sparse_survey_evaluation.

Table 3: Experimental setup

Testbed	NVIDIA A100 (GPU)	AMD EPYC 7402 (CPU)
Cores	6912 CUDA cores	24 cores
Memory	40 GB HBM2 (BW 1555 GB/s)	128 MB LLC (L3) 256 GB DDR4
Measured BW	1350 GB/s	DRAM 77 GB/s
Compiler	cuda-12.5	gcc 12.2.0

7.1 Experimental setup

In our experiments, we use server-class CPU and GPU; a 24-core AMD EPYC 7402 CPU and an NVIDIA A100 GPU. A detailed presentation of each platform is presented in Table 3. The 24-core EPYC processor is composed of multiple chiplets, referred to as Core Complex Dies (CCDs), with each CCD containing 3 cores that share the L3 cache. Additionally, the CPU is configured with a single NUMA domain (NPS1). Threads are pinned to cores using OpenMP environment variables, and matrices are initialized in parallel using the Linux first-touch policy, ensuring that data is allocated in the NUMA node local to the thread accessing it. For each testbed, we calculate the memory bandwidth using the STREAM benchmark and report it alongside the nominal bandwidth. For each configuration (testbed, matrix, and format), we execute 128 iterations of **SpMM** and **SDDMM**, and record the average performance in GFLOPs. Subsequently, we calculate the arithmetic mean across three independent experiments to ensure robustness.

7.2 Datasets

The first dataset comprises the graph collection described in Section 5.5, Table 2. This dataset spans a wide range of graph sizes, from a few thousand to several hundred thousand nodes, and from approximately 9K up to 40M edges. Due to hardware limitations, not all graphs could be included in the experiments, as some exceeded the memory capacity of a single device. Ultimately, 18 representative graphs were selected, as listed in Table 4. These graphs originate from diverse domains such as bioinformatics, social and communication networks, and citation graphs. The majority are extremely sparse (99.9% sparsity), with only one exception exhibiting 94% sparsity. We utilize these graphs to assess the performance of the available **SpMM** and **SDDMM** kernels.

The second dataset is drawn from the DLMC benchmark suite, presented in Section 5.1 and originally introduced in [63]. These matrices are derived from deep learning workloads, specifically from the weight tensors of neural networks subjected to unstructured pruning. The pruning strategies include magnitude pruning, random pruning, L0 regularization, and variational dropout, all of which induce sparsity patterns representative of modern deep learning models. The characteristics of these matrices are summarized in Table 5, with a total of 2680 matrices being available. For the **SpMM** kernels, we directly employ the DLMC matrices. For the **SDDMM** kernels, however, we adopt a different strategy. Instead of weight-derived sparsity, we generate attention masks following the methodology described in [265] and Section 3.4. Specifically, we construct random, windowed, and global attention masks, as well as the BigBird mask that combines these three patterns. The defining features of these masks, such as the number of rows and target sparsity levels, are used as input to the generator and correspond to the values reported in the DLMC dataset, leading to the generation of 84 masks.

7.3 Formats and algorithms examined

Our performance evaluation contains publicly available implementations provided both by hardware vendors (Intel, AMD, and NVIDIA) and by the broader research community.

Table 4: Graph matrices used for evaluation

Matrix	Rows	Nonzeros	Sparsity (%)	Size (MB)
citeseer	3327	9340	99.9156	0.08
cora	2708	10858	99.8519	0.09
pubmed	19717	88673	99.9772	0.75
PROTEINS	43471	162088	99.9914	1.40
ogbl-ddi	4267	1067911	94.1347	8.16
ogbl-collab	235868	967632	99.9983	8.28
ogbn-arxiv	169343	1166243	99.9959	9.54
harvard	15126	1649234	99.2792	12.64
com-Amazon	334863	1851744	99.9983	15.41
REDDIT-BINARY	859254	1991016	99.9997	18.47
amazon0505	410236	3356824	99.9980	27.18
OVCAR-8H	1890931	3946402	99.9999	37.32
wiki-Talk	2394385	5021410	99.9999	47.44
roadNet-CA	1971281	5533214	99.9999	49.73
com-Youtube	1134890	5975248	99.9995	49.92
web-BerkStan	685230	7600595	99.9984	60.60
sx-stackoverflow	2601977	36233450	99.9995	286.36
ogbn-proteins	132534	39561252	99.7748	302.33

Table 5: DLMC dataset used for evaluation

Features	Values
Rows	512, 2048
Nonzeros	250 - 673K
Sparsity (%)	50, 60, 70, 80, 90, 95, 98

All experiments are conducted using single-precision floating point (FP32), as this is the most universally supported precision level across the selected formats. Formats operating on alternative precisions—such as bfloat16, FP16, or INT8—are excluded from this study due to limited and inconsistent support across platforms and implementations.

To ensure a fair comparison of computational performance, we deliberately exclude any preprocessing overheads associated with specific formats. This decision allows us to focus purely on the runtime performance of **SpMM** and **SDDMM** operations, without skewing results due to format-specific setup or conversion stages.

In terms of kernel support across platforms, we observe a significant imbalance in format availability as discussed in Section 4. On GPUs, a wide variety of **SpMM** implementations are available, offering flexibility in both format and optimization strategy. In contrast, CPU-based **SpMM** support is more limited, with only four available. We note that for three GPU **SpMM** kernels (dgSPARSE, GNNPilot, DTC) multiple versions are available. After testing with both datasets, we select and present results for the best performing version of each format.

The disparity in kernel support is even more pronounced for **SDDMM** kernels. While several GPU-based implementations exist, CPU support is extremely limited. In fact, only one usable **SDDMM** kernel is available on CPU, the implementation from ASpT [96]. Although the FusedMM library [198] includes an **SDDMM** operator, it is tightly integrated with its **SpMM** routine, preventing us from isolating and benchmarking the

Table 6: Formats and implementations tested on each testbed

NVIDIA A100 (GPU)		AMD EPYC 7402 (CPU)	
SpMM	SDDMM	SpMM	SDDMM
cuSPARSE [181]	cuSPARSE [181]	Intel-MKL [240]	ASpT [96]
ASpT [96]	ASpT [96]	AOCL-Sparse [3]	
RoDe [189]	dgSPARSE [39]	ASpT [96]	
dgSPARSE [103; 39]	GNNPilot [101]	FusedMM [198]	
GNNPilot [101]			
DTC-SpMM [51]			
Sputnik [63]			

SDDMM kernel independently. As a result, FusedMM’s SDDMM component is excluded from our evaluation. The complete mapping of supported formats for each kernel and platform is summarized in Table 6.

Our evaluation is carried out across multiple inner dimensions of the matrices involved in the computations, specifically the sequence length, with three representative values examined: 32, 128, and 1024.

7.4 Results and analysis

Before proceeding with the analysis of the evaluation results, we note how the results are presented and interpreted. All figures in this section are shown as boxplots, which aggregate the performance measurements of all matrices belonging to each category shown on the x-axis. These plots provide an overview of the performance trends, while also illustrating the median performance and the variability across different matrices within each category.

In addition, it is important to highlight a key consideration regarding the CPU measurements. The performance of the CPU kernels is strongly influenced by the total problem size in relation to the capacity of the Last-Level Cache (LLC). By problem size, we refer to the combined memory footprint of the sparse matrix and the dense matrices involved in the computations. When the working set fits within the LLC, data reuse is maximized and memory access overheads are minimized, enabling higher performance. However, once the working set exceeds the LLC capacity, frequent accesses to main memory dominate execution, leading to a significant drop in performance. This effect is particularly relevant for the graph dataset, where the total problem size may fall either below or above the LLC threshold depending on the embedding dimension (k). In contrast, the DLMC matrices are relatively small, resulting in problem sizes that consistently remain within the LLC across all cases.

7.5 SpMM

Figure 18 presents the performance of the SpMM kernel across the two datasets. On the GPU, we observe comparable average performance for both graph and DLMC matrices, with only a small fraction of DLMC cases (fewer than 5%) achieving noticeably higher performance. It is worth emphasizing that the theoretical peak performance of the GEMM kernel on the evaluated GPU is 18 TFLOPs/s. In comparison, the SpMM kernel attains on average only about 2.5% of this peak, with rare cases reaching up to 20%.

The CPU exhibits a different trend. Here, SpMM performance on the DLMC matrices is, on average, $1.5\times$ higher than on graph matrices, for problem sizes smaller than the LLC. Regarding larger than LLC problem sizes, which are observed in graph matrices only, we observe a large drop in performance, $6\times$ on average, which shows the large effect that the LLC size has on the attainable performance. The theoretical peak performance of the GEMM kernel on the CPU is 1.4 TFLOPs/s. Relative to this bound, the DLMC matrices achieve approximately 20% of peak performance on average, with some matrices approaching the upper limit. In contrast, the graph matrices remain far behind, reaching about 13% of peak performance on average.

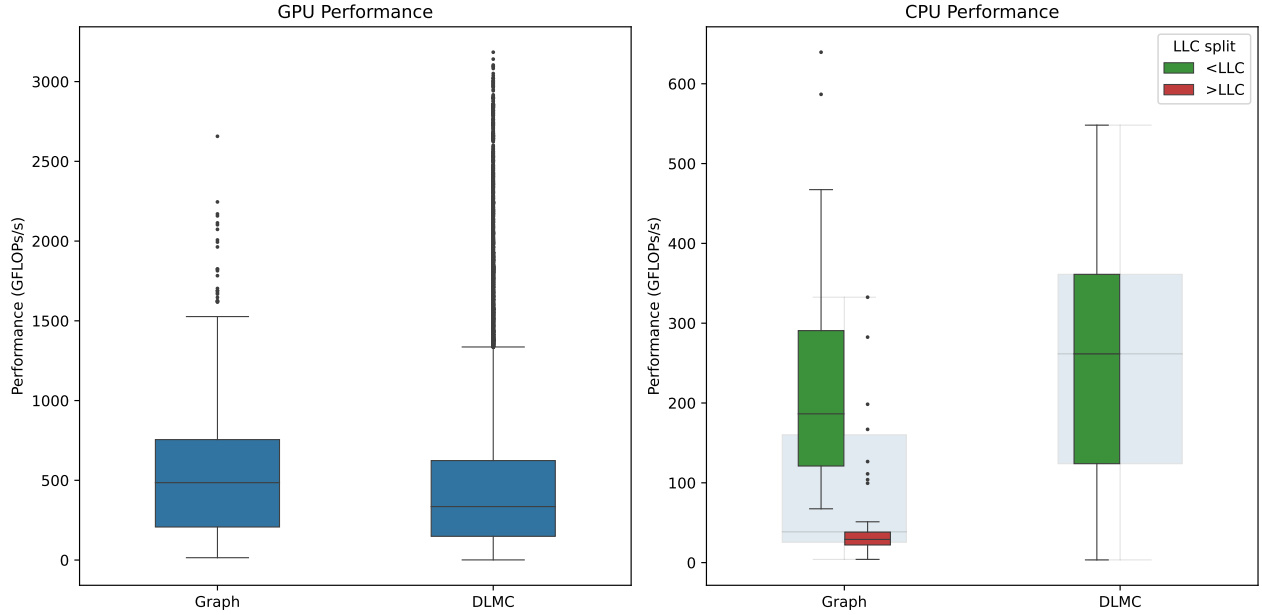


Figure 18: Performance of the SpMM kernel for the two different datasets on GPU and CPU (embedding dimension $k=128$). The CPU results are split in two boxplots, for smaller- and larger-than-LLC problem sizes, as LLC affects results of CPU performance by a great factor.

7.5.1 Format evaluation

Figure 19 presents the performance of the various SpMM formats and implementations evaluated in this study. Across both matrix datasets, similar performance trends are observed. Vendor-provided formats, such as cuSPARSE (NVIDIA), MKL (Intel), and AOCL (AMD), deliver stable and consistent performance across all cases. However, these implementations do not achieve the highest attainable performance, exhibiting relatively limited performance ceilings compared to other evaluated formats. All formats exhibit considerable performance variability, with RoDe, dgSPARSE, DTC, and Sputnik for the GPU showing particularly large fluctuations on the DLMC matrices. We also observe that no format is particularly dominant in performance compared to the others, with the exception of the RoDe format for the graph dataset.

7.5.2 Sparsity effect on performance

The impact of matrix sparsity can be analyzed using the DLMC dataset, where sparsity levels range from 50% to 99%. The graph matrices, in contrast, are predominantly extremely sparse, with roughly 99% sparsity in most cases. Figure 20 presents the effect of sparsity on SpMM performance for both GPU and CPU. As expected, performance decreases substantially on both devices as sparsity increases from 50% toward 100%, approximately a $12\times$ drop on the GPU and a $5.5\times$ drop on the CPU. On the GPU, the decline in performance is nearly linear with increasing sparsity, because GPUs rely on a large number of concurrent computations to maintain high throughput; when the number of computations decreases due to higher sparsity, the GPU is underutilized and performance drops sharply. On the CPU, performance remains relatively stable for sparsity levels up to 80%, since CPUs can still exploit vectorized operations and efficiently schedule the smaller number of computations. Beyond this point, however, the reduced workload leads to a steep drop in performance, as vectorization and instruction-level parallelism can no longer be efficiently utilized. We need to note, however, that the performance reported here expresses throughput, i.e., operations per time unit. The overall latency of the entire kernel is affected by both throughput and number of operations, with sparsity having a direct and opposite impact on both. This constitutes a significant tradeoff when tuning an inference engine, especially when one needs to take into account the impact of sparsity on accuracy as a third important parameter.

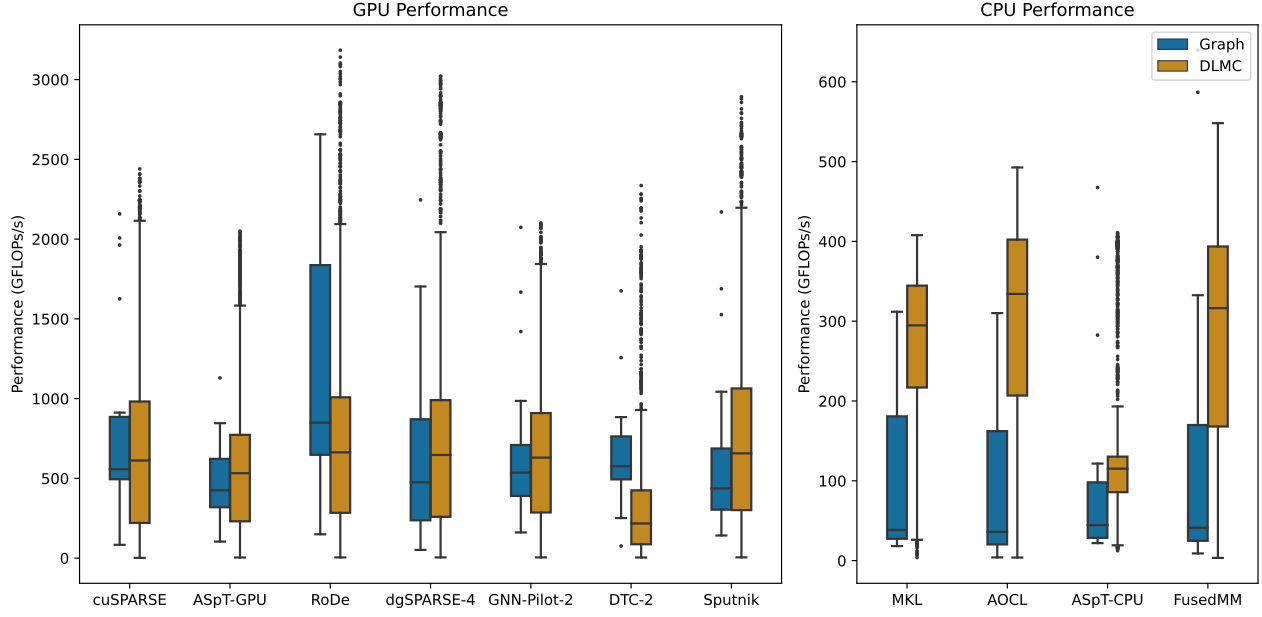


Figure 19: Performance of the evaluated SpMM formats for the two different datasets on GPU and CPU (embedding dimension $k=128$).

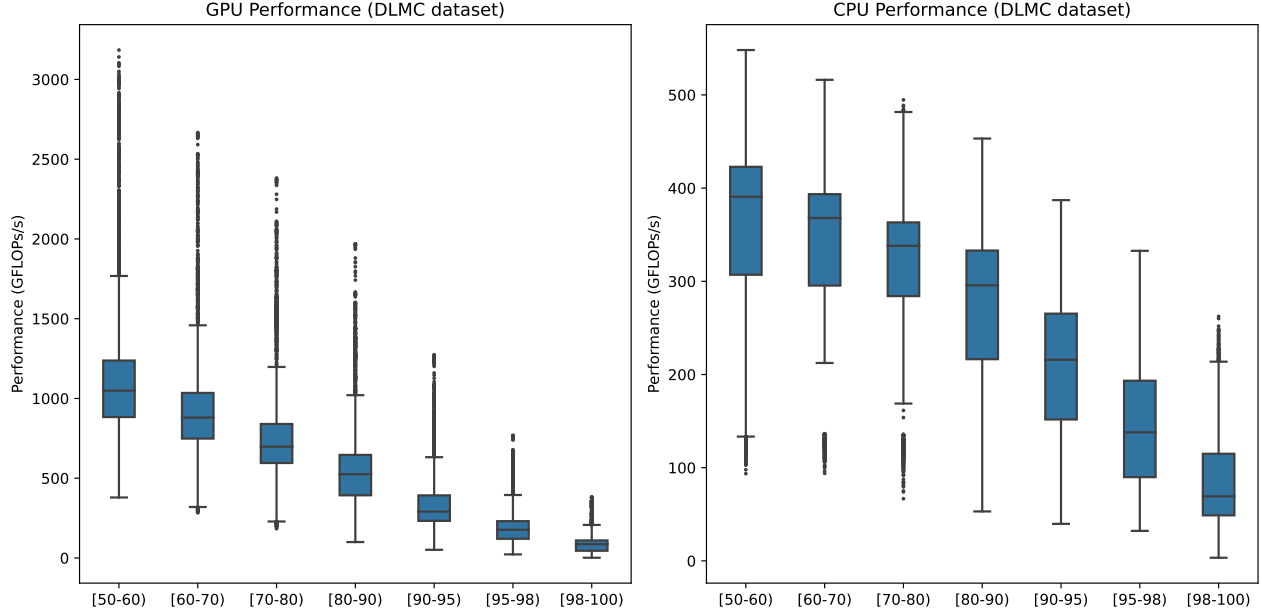


Figure 20: Effect of sparsity levels of the DLMC matrices on the performance of SpMM on the GPU and the CPU (embedding dimension $k=128$).

7.5.3 Embedding dimension (k) effect on performance

Figure 21 presents the effect of the different k values on the performance of the SpMM kernel. We observe distinct patterns for the graph and DLMC datasets. On the GPU, changing the embedding dimension does not significantly affect performance on graph matrices. In contrast, DLMC matrices, which exhibit higher density and therefore greater computational requirements, show performance scaling by up to $10\times$ as k increases from 32 to 1024. This behavior can be explained by the interaction between computational intensity and memory access patterns on each platform. The higher density of the DLMC matrices provides sufficient

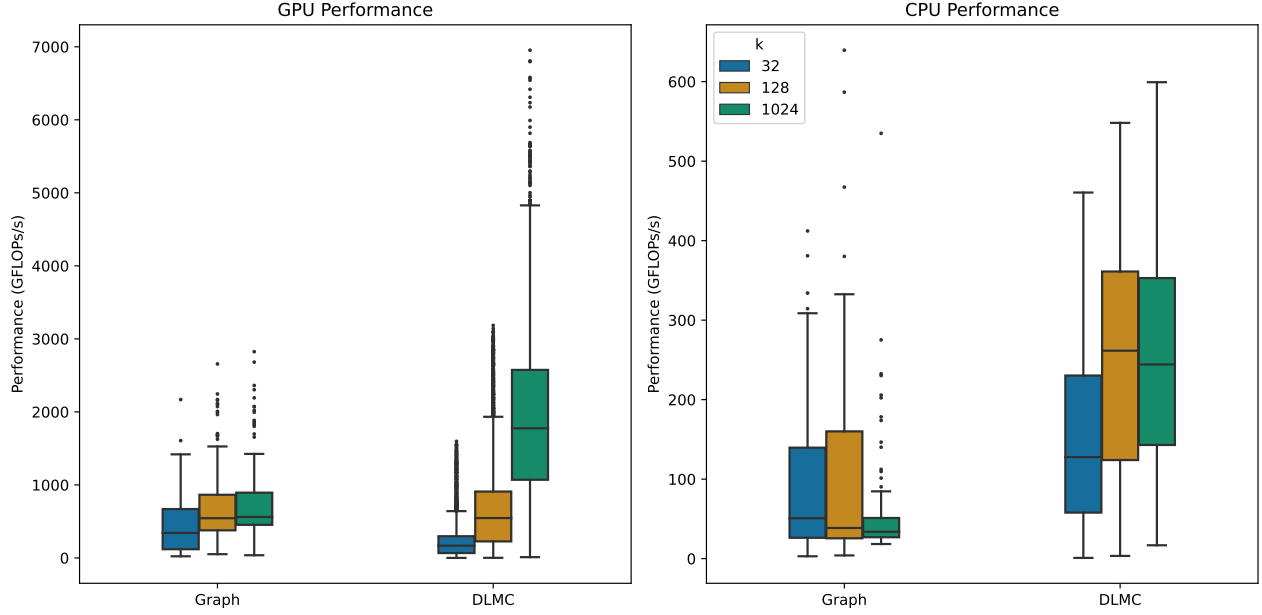


Figure 21: Effect of embedding dimension on the performance of SpMM for the two different datasets on GPU and CPU.

computational workload to exploit the massive parallelism of the hardware, and increasing k directly increases the arithmetic intensity, leading to proportional performance gains. On the other hand, graph matrices are too sparse, and their memory-bound nature limits the benefit of scaling k .

On the CPU, we must take the total problem size into account before analyzing the results. Figure 22 illustrates the different performance patterns as k varies for problem sizes smaller and larger than the LLC. For both datasets, when the embedding dimension is large (1024), performance does not improve substantially compared to the medium size (128) for problem sizes smaller than the LLC. For the larger-than-LLC cases, which occur only for graph matrices, we observe that the embedding dimension has no impact on performance, as the memory bandwidth becomes the dominant bottleneck limiting CPU performance, and additional computation from larger k values cannot be effectively utilized.

7.6 SDDMM

Before presenting the **SDDMM** results, it is important to clarify that the matrices referred to as part of the DLMC dataset are not directly taken from DLMC, but are instead generated matrices that combine feature values originating from the DLMC dataset with the structural sparsity patterns of matrices from the BigBird [265] paper.

Figure 23 presents the performance of the **SDDMM** kernel across the two datasets. On the GPU, the two datasets behave quite differently. The very high sparsity levels of the graph matrices prevent the **SDDMM** kernel from achieving high performance. Conversely, the DLMC matrices contain some rare cases of higher throughput and, on average, deliver $1.44\times$ better performance than the graph matrices. Overall, the **SDDMM** kernel reaches only 2.3% of the peak GPU performance, with a few outliers achieving up to 22%. On the CPU, where only a single implementation (ASpT-CPU) is available, the picture changes. Contrary to the GPU results, graph matrices consistently achieve higher performance than DLMC matrices. Interestingly, and unlike our previous observations, problem sizes that exceed the LLC capacity show better performance than those that fit within it. Nevertheless, the absolute performance levels remain low: only 2.3% of peak performance on average for graph matrices and about 1% for DLMC matrices.

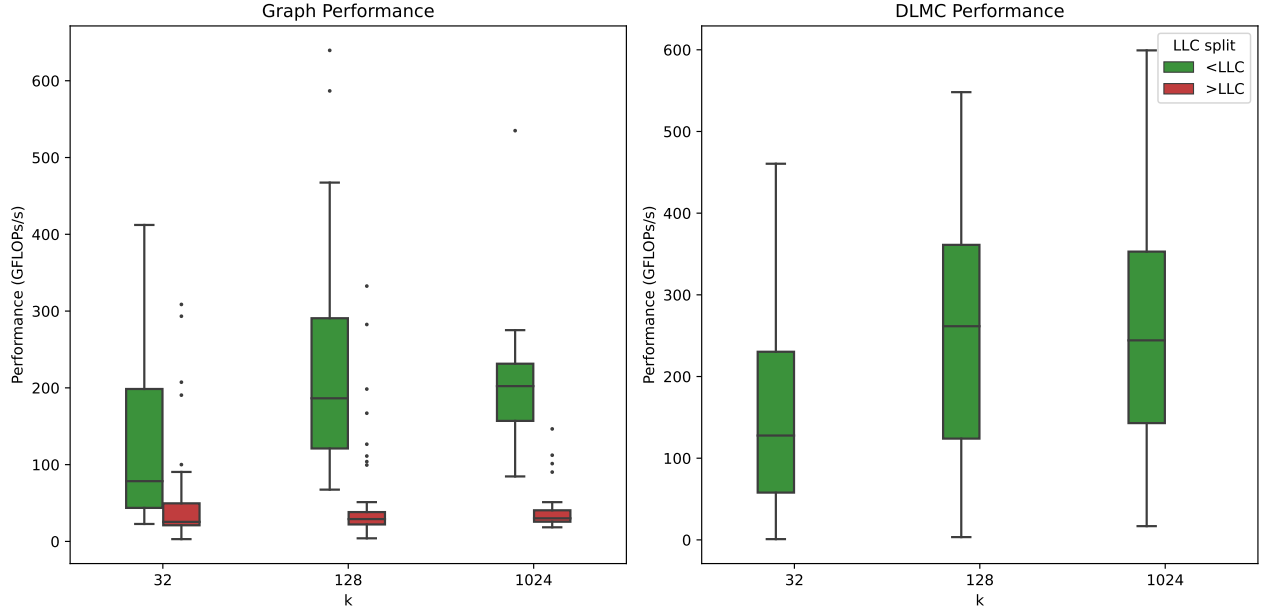


Figure 22: Performance of **SpMM** on CPU for the graph and DLMC datasets, with splitting between smaller- and larger-than-LLC problem sizes.

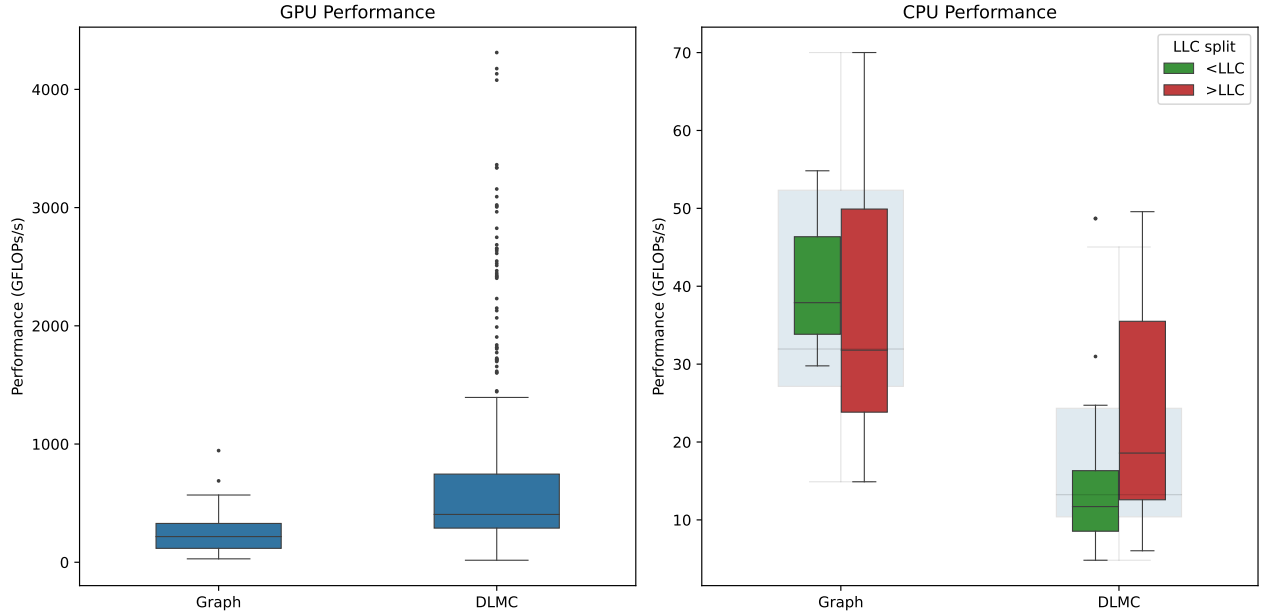


Figure 23: Performance of the **SDDMM** kernel for the two different datasets on GPU and CPU (embedding dimension $k=128$).

7.6.1 Format evaluation

Figure 24 presents the performance of the different **SDDMM** formats and implementations evaluated. On the GPU, the large performance gap between the graph and DLMC datasets is primarily driven by the cuSPARSE and ASPT-GPU implementations, which perform significantly better on the DLMC matrices. In contrast, the other two formats deliver consistently lower and comparable performance. On the CPU side, with only a single available format, the observations remain unchanged compared to the previous analysis.

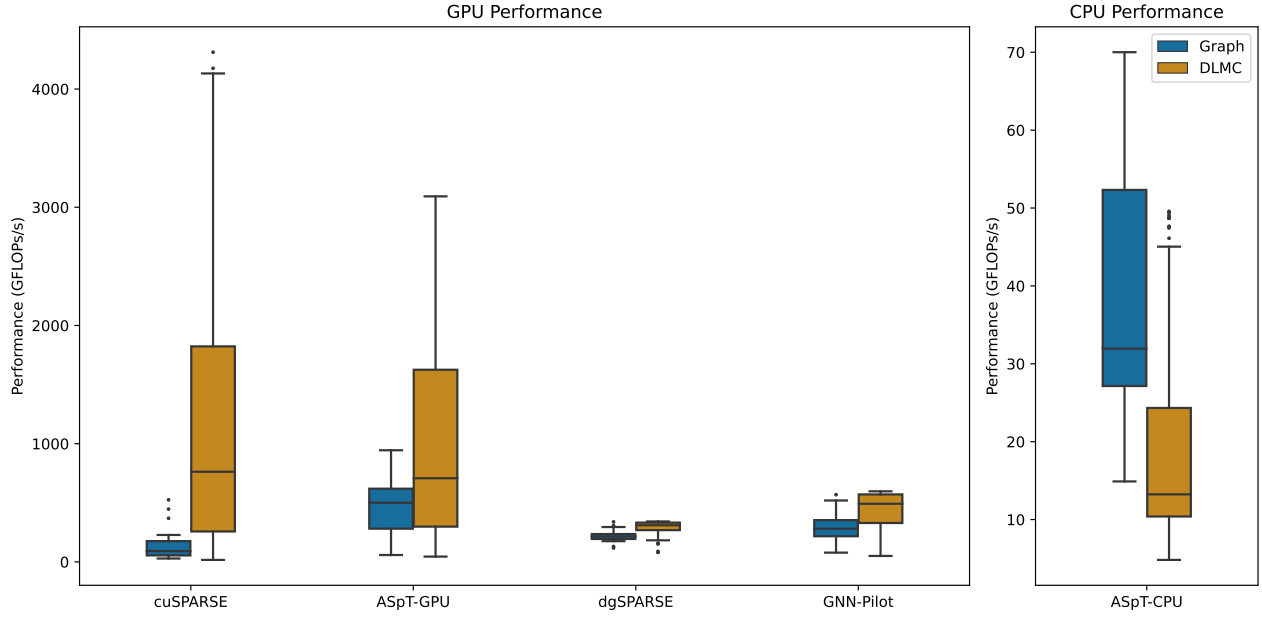


Figure 24: Performance of the evaluated **SDDMM** formats for the two different datasets on GPU and CPU (embedding dimension $k=128$).

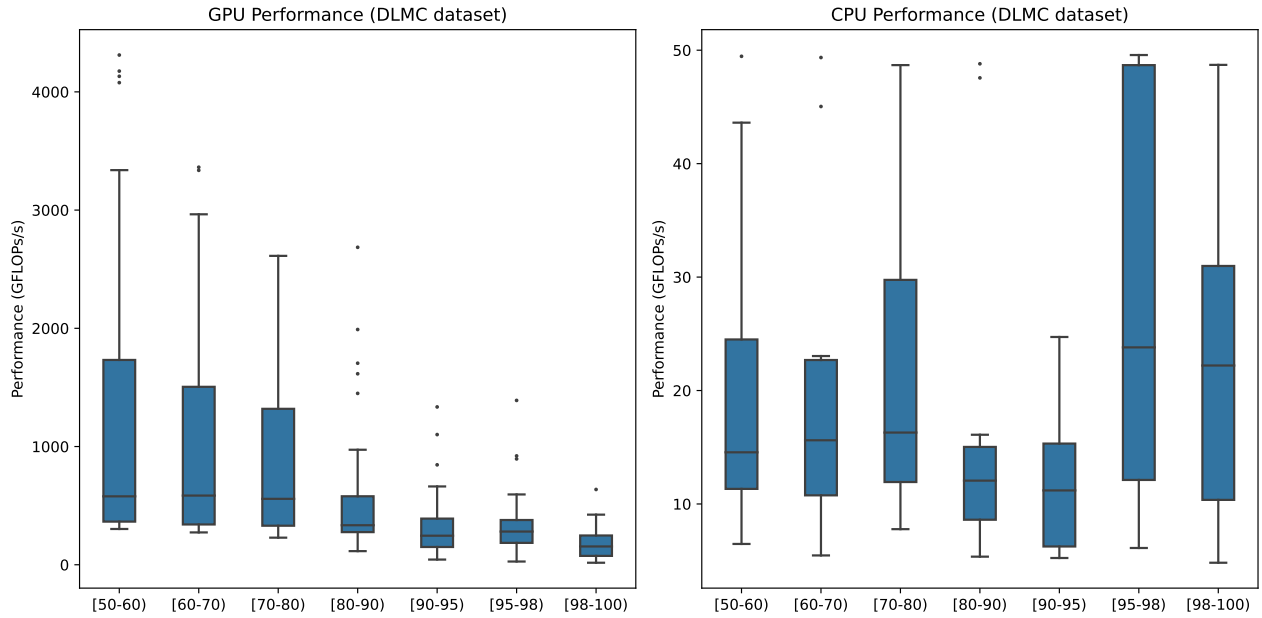


Figure 25: Effect of sparsity levels of the DLMC matrices on the performance of **SDDMM** on the GPU and the CPU (embedding dimension $k=128$).

7.6.2 Sparsity effect on performance

Figure 25 shows the effect of mask sparsity on **SDDMM** performance for both GPU and CPU. On the GPU, we observe a clear sensitivity to sparsity: once sparsity exceeds 80%, performance drops steeply. Overall, as sparsity increases from 50% to nearly 100%, average performance decreases by a factor of 3.8x, and the GPU is, similarly to the **SpMM** kernel, underutilized. On the CPU, performance patterns are less stable across sparsity levels. Given the already low performance of the available format, no firm conclusions can be drawn regarding the impact of sparsity in this case.

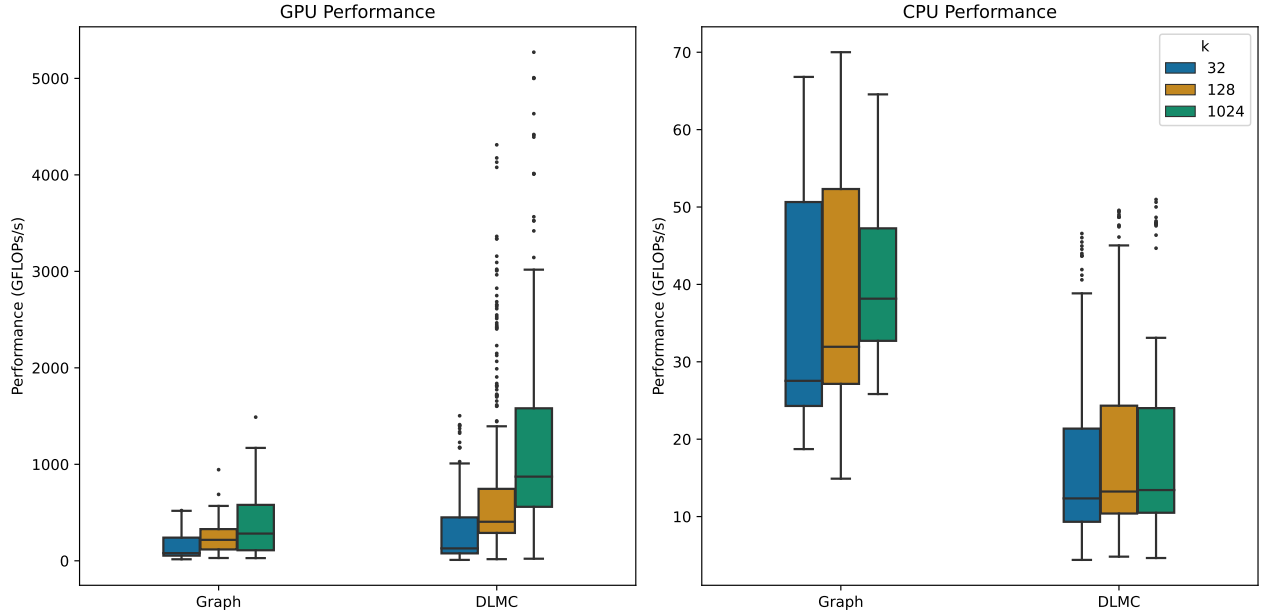


Figure 26: Effect of embedding dimension on the performance of **SDDMM** for the two different datasets on GPU and CPU.

7.6.3 Embedding dimension (k) effect on performance

Figure 26 presents the effect of the embedding dimension (k) on the performance of the **SDDMM** kernel, revealing distinct patterns for the graph and DLMM datasets. On the GPU, varying the embedding dimension has little impact on graph matrices. In contrast, DLMM matrices, being denser and thus more computationally demanding, show performance scaling by a factor of $6.7\times$ as k increases from 32 to 1024. This trend is consistent with what we observed for **SpMM**, though the scaling effect here is less pronounced. On the CPU, the analysis must again take into account the total problem size and the LLC capacity, with the results presented in Figure 27. For problem sizes that fit within the LLC, performance decreases on average as the embedding dimension grows. For problem sizes exceeding the LLC, however, the two datasets diverge: graph matrices experience a modest performance improvement, while DLMM matrices exhibit a drop in performance. However, given the limited computational capabilities of the only available CPU format, it is difficult to draw firm conclusions regarding the true effect of embedding dimension on CPU performance.

7.7 Porting effort

The process of building a common benchmarking framework and integrating the various **SpMM** and **SDDMM** implementations proved far from being trivial. We categorize the challenges along two axes: ease of integration and functional validation.

7.7.1 Ease of integration

Vendor-provided libraries such as NVIDIA cuSPARSE [181], Intel MKL [240], and AMD AOCL-Sparse [3] were the most straightforward to integrate. These libraries are well-documented and generally worked out-of-the-box, with only minor adjustments required. For instance, cuSPARSE required matrices to be stored in row-major layout with explicitly specified leading dimensions, while AOCL-Sparse required switching to the development branch of their repository to access the multi-threaded **SpMM** variant. In contrast, research-oriented formats lacked the same plug-and-play convenience. Many required substantial code modifications, and documentation was not detailed in most cases, making the integration process more challenging. A few representative cases include:

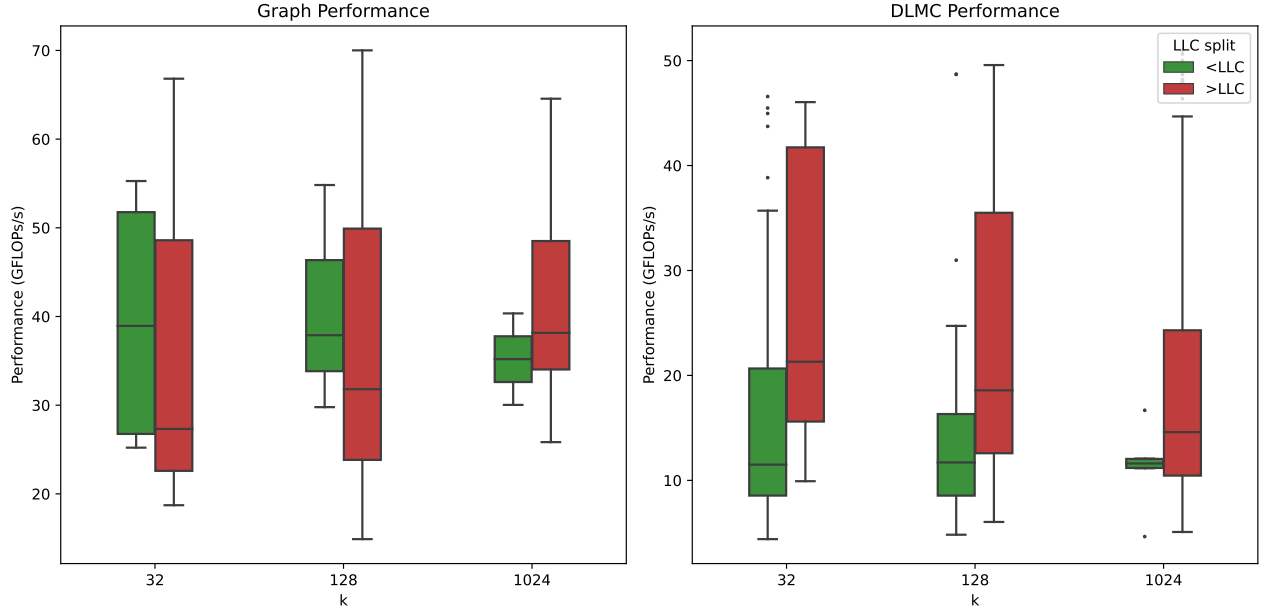


Figure 27: Performance of SDDMM on CPU for the graph and DLMC datasets, with splitting between smaller- and larger-than-LLC problem sizes.

- ASpT [96] (GPU): Required extensive editing to replace deprecated CUDA primitives (e.g., replacing `shfl` with `shfl_sync`). For the CPU implementation, we had to replace Intel’s `icpc` with `gcc`, while several compiler pragmas were unavailable.
- FusedMM [198] (CPU): Required detailed inspection of internal functions to isolate the `SpMM` operator. Unfortunately, the SDDMM operator could not be extracted in a standalone manner.
- DTC-SpMM [51], HC-SpMM [140], and GNNPilot [101] (GPU): Designed for PyTorch integration. To isolate their `SpMM` and SDDMM kernels, we first had to compile them within the PyTorch framework to obtain the correct compilation and linking instructions, before adapting the computational kernels for standalone benchmarking.
- RoDe [189] (GPU): Required removal of several external dependencies, including `absl` (Abseil Common Libraries) and `glog` (Google logging).
- Sputnik [63] and dgSPARSE [39] (GPU): Could be compiled successfully but required manual re-configuration of build and linking procedures.

7.7.2 Functional validation

Successful compilation did not always guarantee correct execution. While vendor libraries ran reliably, several research implementations encountered issues:

- Acc-SpMM [274] and HC-SpMM [140] (GPU): Ran successfully for only a single graph matrix, preventing the collection of meaningful results.
- Sputnik [63] (GPU): Produced correct results for both `SpMM` and SDDMM, but the SDDMM kernel was prohibitively slow and, thus, excluded from the final evaluation.
- DTC-SpMM [51] (GPU): Offered multiple variants, with some requiring specific embedding dimensions (k) to execute correctly. Similar restrictions on k were observed in HC and ACC implementations.

- RoDe [189] (GPU): The SDDMM implementation produced incorrect results with no straightforward fix.
- VectorSparse [22], Magicube [138] and SMaT [185] (GPU): Supported only FP16 precision, whereas our evaluation focused on FP32, making them unsuitable for inclusion.

7.7.3 Summary

Overall, the integration effort revealed a clear divide: vendor libraries were reliable, easy to use, and stable, but limited in flexibility and peak performance potential. Research kernels, while often promising higher performance, typically required extensive code modifications and debugging to become operational, with many exhibiting incomplete or unstable functionality.

8 Conclusions

To summarize our conclusions, we return to the initial questions posed in Section 1 and discuss our findings below.

In which forms does sparsity arise in DNN inference? We found that sparsity is discussed in several forms, as summarized in Figure 14. Sparsity can be enforced, i.e., explicitly injected in the process as is the case of weight pruning and sparse attention in LLMs. The sparsity can also be natural in the case of GNNs that represent a naturally sparse domain. Finally, sparsity can be ephemeral, that happens exactly due to the nature of several activation functions that zero out the outcome of DNNs layers. All aforementioned forms of sparsity ultimately resort to actual sparse computations in their implementations. On the other hand, 3D convolutions for point clouds and Mixture of Experts (MoE) models, although frequently categorized as sparse approaches, rely on dense computations.

We distinguish activation sparsity as a more challenging form in the sense that sparse data are dynamically generated during inference for each distinct inference operation. This means that sparse matrices need to be generated on the fly and any computational optimization (e.g., matrix reordering) must seriously consider the preprocessing cost.

In the case of weight pruning, there exist three general types of sparsity: structured sparsity that removes large parts of the network and leads to smaller dense networks, unstructured sparsity that freely removes any weight of the network according to some optimization criterion, and semi-structured sparsity, a hybrid of the above that is allowed to remove a specific ratio of nonzero elements (e.g., 2 out of 4) within an area of the original weight matrix. These lead to different performance/efficiency/accuracy tradeoffs that differ across computational devices.

How do the original dense computations transform to sparse computational kernels? Original DNN inference is dominated by dense matrix operations, specifically matrix multiplication, matrix-vector multiplication and convolution. Starting from the initial dense computation in each type of DNN layer (Section 2.1), we show in Section 3 that the involved kernels in sparse DNN inference are primarily sparse matrix by dense matrix multiplication (SpMM), sampled matrix-matrix multiplication (SDDMM), sparse matrix by dense vector multiplication (SpMV) and sparse convolution.

What is the current state-of-the-art in the implementation of these kernels? Sparse kernels, especially those that work on unstructured sparsity, are well known for their very low efficiency, as they suffer from memory bottlenecks and imbalance. There is a vivid research interest in optimizing the SpMM and SDDMM kernels on GPUs applying optimizations that target specifically the improvement of locality and load balance. SpMV, convolutions, and in general CPU implementations seem to be lagging behind. We argue that there is a lot of room for performance improvement in all kernels across computational devices.

What sparse datasets can be used to assist research and development in sparse DNNs? We studied the datasets used in the various research papers that deal with sparsity in the field and note the following:

-
- Sparse matrices that represent graphs are collected from well-known databases like Suitesparse [35], SNAP [135], OGB [99], TUD [176], and seem to capture well the properties of the domain.
 - Researchers rely heavily on the valuable DLMC [63] dataset for their experiments, which includes matrices from sparsified layers of the original Transformer and ResNet-50 networks. We argue, however, that these networks are rather old, and a new dataset reflecting the sizes and properties of current architectures needs to be generated.
 - Evaluating end-to-end performance in state-of-the-art works relies on ad-hoc sparsification approaches of existing networks, typically distinct in each research work. We find that this is an approach that can be followed primarily by domain experts. Access to publicly available entirely sparsified networks could be of value to performance engineers.
 - For experimentation with fixed attention masks in Transformers, Big Bird [265] provides a solid base. However, datasets to support experimentation with learnable sparse attention seem to be missing.
 - Experimenting with activation sparsity can only be done by incorporating the approach in an end-to-end network, a workflow well understood by domain experts but less familiar to the broader community. Datasets that reflect the outputs of different activation mechanisms in modern networks seem to be missing. These could also enable more intense involvement of performance engineers.

What is the current software support to deploy production systems that rely on sparse DNNs?

Vendors and researchers have invested significant effort to provide libraries and tools that manage sparsity. Semi-structured, N:M sparsity seems to be more mature, it has solid software and hardware support, and can be more easily incorporated in existing deployment pipelines. Sparsity in GNNs also enjoys very good support and is operational. Regarding unstructured sparsity in weights, activation, and attention, a large number of libraries for optimized kernels is available, together with other supporting tools like pruning software. However, an end-to-end framework or methodology to deploy a unstructured sparse DNN with production quality seems to be missing, to the best of our knowledge.

What is the performance behavior of sparse kernels in modern CPUs and GPUs? In Section 7 we performed a preliminary performance evaluation of the state-of-the-art **SpMM** and **SDDMM** kernels that work with unstructured sparsity. We summarize our findings below:

- As expected, GPU performance for both kernels greatly exceeds CPU performance.
- In most of the cases, however, the efficiency of the kernels is extremely low, with the GPU not surpassing 3% of peak. The CPU was able to reach a decent 20% over peak only for the **SpMM** kernel.
- In all cases, the sparsity of the matrix and the embedding dimension, significantly affect performance. In several cases, computations on graph matrices led to different performance behavior compared to that of DLMC matrices.
- Storage formats and optimizations influence performance. There does not seem to be a clear winner and further investigation is required on this aspect.

Our evaluation effort also served a second purpose, i.e., to assess how straightforward it is to perform comparison of related work. We found that a lot needs to be done in order to have a solid and straightforward benchmarking process. We summarize our thoughts below:

- As mentioned above, there are several gaps in the available datasets.
- Profiling, benchmarking and performance evaluation over complex, deeply stacked frameworks like PyTorch may be cumbersome and counter-intuitive.
- We seem to be missing a common, simple and indicative benchmark or benchmarking methodology that can be used to assess the end-to-end impact of sparsity in modern DNNs, with an additional focus on LLMs.

Appendix

A Weight pruning for model sparsification

There exist various approaches that apply weight pruning and differ in terms of when, how, and what to sparsify. These are analyzed in the following sections.

A.1 When to sparsify

Sparsification can take place *before*, *during* or *after* training:

1. **Sparsify before training:** The model is initialized with a fixed sparse connectivity pattern and trained from scratch without modifying the mask. Such static sparse initializations are simple and avoid the pruning cost, but generally lead to reduced accuracy. Examples include random sparse initialization and fixed-topology sparse training methods [172] and recent analyses showing that randomly pruned networks can sometimes train effectively from scratch [151].
2. **Sparsify during training:** Sparsity can also be introduced while the model is being optimized, and training may start from either a dense or a sparsely initialized network. First, such approaches may be *sparsity-enforcing*, where explicit top- k or magnitude-based rules maintain a desired sparsity level (either globally or per-layer), or *sparsity-inducing*, where regularizers such as L1, group-lasso, or L0 gates encourage weights to become zero without enforcing a specific target [159]. Second, the sparsity pattern may remain *static*, follow a *gradual pruning schedule* that increases sparsity over training [277], or evolve dynamically through *pruning-and-regrowth* mechanisms, as in dynamic sparse training (DST) approaches [172; 48; 234]. These mechanisms introduce or maintain sparsity during learning, forming the basis of *sparsity-aware training*, where sparsification is integrated into the optimization process rather than applied only after convergence.
3. **Sparsify after training:** The train-then-sparsify is the most common approach and the one implied in Figure 7. In this method, also known as post-training pruning, the network is first trained densely to convergence, making it easier to assess which weights are genuinely redundant. The trained model is then analyzed to identify low-importance parameters—typically those with small magnitudes or low sensitivity—and these weights are removed. Post-training pruning provides a natural baseline for evaluating sparsification methods, since the performance of the pruned model can be directly compared against that of the original dense network [131; 81; 80]. However, despite its intuitive nature and simplicity, a dense model that was not trained with sparsity in mind may not be optimally positioned for extreme sparsification, which motivates methods that incorporate sparsity constraints during training [48; 62].

A.2 How to sparsify

A.2.1 Structured sparsity

Structured pruning targets entire filters or neurons and achieves significantly better performance on current hardware due to its compatibility with dense computations [137; 81; 251]. It also reduces the representational overhead, as fewer indices need to be stored. Structured pruning tends to perform well even when applied randomly at initialization, making it useful for early-stage architecture search [160]. However, it typically leads to lower accuracy per pruned element and may require more training iterations or computational resources to achieve similar final results [174; 93].

A.2.2 Unstructured sparsity

Unstructured pruning, often called unstructured sparsity, is the most common type of sparsity investigated. This method, exemplified by weight-level pruning, generally preserves higher accuracy per pruned element compared to structured methods [81; 62]. Its fine-grained approach allows for precise control over which

parameters are removed, and the resulting sparse matrices can be efficiently represented using standard sparse matrix formats (see Section 2.3), which is why these neural networks can retain their accuracy even at high sparsity ratios. Despite the advantages of unstructured pruning, it comes with severe tradeoffs as it suffers from lower computational efficiency on modern hardware and incurs higher memory overhead because it needs to store indices for each individual pruned weight. In addition, unstructured pruning typically degrades accuracy when applied randomly at initialization, since the specific connections in the network are more important at that level of granularity. Although this type of sparsity can be induced in multiple ways, the resulting weight matrices can have distinct characteristics. While these properties could potentially be exploited to improve sparse kernel performance, such exploration is beyond the scope of this discussion. The following section will explore some of the most common and state-of-the-art pruning algorithms used by the community.

Random pruning The simplest way to introduce unstructured sparsity is, of course, randomly. This method is not used in practice, as the accuracy of the resulting model is significantly degraded most of the time [57]. Nevertheless, random pruning provides a convenient baseline for experimentation, serving as a control condition for evaluating more principled pruning methods.

Magnitude pruning This is the most intuitive, widely used, and often surprisingly effective method. It is based on the simple idea that the most influential weights of the model are those with the largest absolute values, so the rest can be removed with little effect on accuracy. In the post-training setting, this typically involves ranking weights by magnitude, pruning the smallest ones (either globally or per layer), and then fine-tuning the model to recover accuracy [80; 155]. Although one-shot magnitude pruning can lead to noticeable accuracy degradation at high sparsity levels, its performance after retraining is often competitive with more sophisticated methods, which, together with its minimal computational overhead, explains its widespread adoption. Beyond post-training pruning, magnitude-based criteria also play an important role in sparse-aware training methods where masks are learned jointly with model parameters. Such dynamic sparse training approaches rely on magnitude-based pruning steps due to their efficiency and simplicity, while focusing on how to enable better joint optimization of weights and masks [202; 110; 67; 234].

More elaborate pruning methods More elaborate pruning methods [131; 84] use second-order information, typically approximations of the Hessian or Fisher matrix, to estimate the sensitivity of the weights to be removed. These approaches aim to eliminate parameters that minimally affect the loss, often achieving higher accuracy than magnitude pruning at comparable sparsity levels. Although computing exact second-order information is expensive, modern scalable variants [221; 125; 242] employ block-diagonal, low-rank, or Hessian-free approximations, making curvature-based pruning practical for contemporary architectures.

Pruning methods for LLMs Based on the same idea, pruning LLMs is considerably more difficult due to the increased number of parameters and the prohibitive retraining requirements. This is why recent state-of-the-art unstructured pruning methods for these models such as SparseGPT [58], Wanda [226] and FISTAPruner [275], pivot from traditional fine-tuning to highly efficient ‘one-shot’ post-training approaches. These advanced techniques address the challenge in different ways: SparseGPT prunes weights layer by layer while locally adjusting them so the model’s output remains close to the original, Wanda evaluates both the weight values and their activation responses to decide which ones matter most, and FISTAPruner frames pruning as an optimization problem that automatically balances model compression with error correction. All are designed to achieve significant model compression and hardware efficiency while preserving high accuracy without the need for costly post-pruning retraining. To balance accuracy and compression though these methods usually do not reach very high sparsity ratios.

Other pruning methods Beyond simple magnitude pruning and its more advanced Hessian-based descendants, several other sophisticated approaches have been explored for post-training unstructured pruning without extensive retraining. These include gradient-based methods like SNIP [133], and GraSP [239] which evaluate weight importance by their impact on the loss’s gradients, movement pruning [212] which assesses importance by how much weights ‘move’ during a brief fine-tuning phase, favoring those that actively adapt. An emerging and theoretically grounded approach is spectral graph sparsification [229] which views the neu-

ral network as a graph and prunes connections while aiming to preserve the graph’s fundamental spectral properties, thereby maintaining critical information flow and connectivity. While these methods offer more sophisticated ways to identify and remove redundancies with high accuracy retention, they are generally more prevalent in research settings and less commonly adopted in widespread industrial deployments compared to magnitude pruning, largely due to the complexities of implementation, hardware compatibility challenges with highly sparse models, and the ongoing need for further validation on diverse large-scale tasks.

A.2.3 Semi-structured sparsity

Semi-structured pruning has recently gained significant attention as it strikes a balance between the flexibility of unstructured pruning and the computational efficiency of structured approaches. It imposes sparsity patterns with limited regularity, such as N:M sparsity—where exactly N out of every M weights remain nonzero within each predefined block [171]. This constrained pattern makes semi-structured sparsity more amenable to hardware acceleration than fully unstructured sparsity, as the regular grouping enables partially optimized kernels and lighter indexing overhead. At the same time, semi-structured sparsity retains considerably more flexibility than coarse structured pruning, often allowing for higher compression rates with minimal accuracy degradation.

Although semi-structured sparsity is not as hardware-friendly as fully dense computation or coarse structured pruning, it provides a practical middle ground for balancing model expressivity, computational efficiency, and memory savings. The growing support for these patterns in modern accelerators—most notably NVIDIA’s 2:4 sparsity support on Ampere and Hopper GPUs—has accelerated their adoption, making semi-structured sparsity a compelling middle ground in the neural network compression landscape.

Basic N:M pruning Simple N:M pruning strategies serve as foundational baselines for comparison with more sophisticated methods, defined by their simplicity and computational efficiency [171]. Random Pruning is a basic control method that enforces a fixed N:M sparsity pattern by randomly selecting N weights to retain within each block of M weights, without regard to weight importance. A more effective, yet still simple, approach is Magnitude-Based Pruning. This method adapts the common magnitude pruning heuristic to the N:M constraint by selecting the N weights with the largest absolute magnitudes within each block of M . This strategy is easy to implement and is compatible with hardware that supports N:M sparsity.

Advanced N:M pruning Beyond blockwise heuristics, recent work has introduced more sophisticated learning-based methods that aim to identify optimal N:M sparsity patterns, particularly for transformer architectures and large language models (LLMs). Most of these approaches operate during training or retraining, where the sparsity mask is learned jointly with the model parameters rather than imposed post hoc. While earlier post-training pruning approaches such as SparseGPT [58] and Wanda [226] established the feasibility of N:M sparsity in the LLM setting, newer methods pursue end-to-end optimization to learn masks that better preserve model performance. The Adaptive Sparse Trainer (AST) [104] is an efficient retraining framework that learns optimal masks during the weight update process. It integrates knowledge distillation from a dense teacher model to accelerate convergence and preserve performance, achieving near-dense model performance on LLaMA2-7B with minimal retraining cost. Similarly, MaskLLM [53] and MaskPro [228] reframe pruning as a learnable problem. MaskLLM uses Gumbel Softmax sampling to model N:M patterns as a learnable distribution, allowing for end-to-end training that directly optimizes the language modeling loss. MaskPro employs a linear-space probabilistic framework to learn a categorical distribution for each weight group, using a moving average of loss residuals to stabilize training. ProxSparse [149] further advances this paradigm with a regularized optimization framework. By transforming the non-differentiable mask selection into a gradual search process, it enables end-to-end learning with global gradient feedback, consistently outperforming previous baselines and demonstrating the clear benefits of a learned, global approach to pruning.

B Sparse Convolutions for Multidimensional Sparse Input

Multidimensional sparse input refers to data that exist in a high-dimensional space (e.g., a 3D volume, but also 4D spatio-temporal sequences [26]) and are naturally sparsely populated, meaning that most of the space

remains empty. A common example is 3D point clouds, i.e., unordered sets of 3D points, each associated with a feature vector, i.e., there are the point coordinates (e.g., x, y, z for a 3D space) and a feature vector at each point (e.g., R, G, B values for each nonzero point in the 3D space). The data is collected through sensors like LiDAR scanners and depth cameras [26].

Unlike 2D image pixels, this representation is irregularly distributed, making it challenging to apply standard convolutions for computer vision tasks [254]. Point cloud convolution approaches that first convert point clouds to a dense volumetric representation and then apply dense CNNs or point-based methods, that directly perform convolution on the k -nearest neighbor (spherical nearest neighbor of each point), are less efficient in large outdoor scenes [147; 230]. Thus, many state-of-the-art deep networks for point cloud segmentation and detection [75; 26; 255; 245] voxelize the data [75] and primarily use a different kind of convolution, coined ‘sparse convolution’ [230]. Sparse convolution avoids the irregular or large memory footprint of the other methods and is empirically proven to be able to scale up to large point clouds [230].

This category of point cloud convolution is different from the sparsified convolution layer discussed in Section 2.3, where there is some sparsity either in the weights (pruning) and/or the input (from previous ReLU activations) [230]. Both operations share the same name in literature (e.g., ‘SparseRT’ [249], which refers to convolution on pruned weights as sparse convolution), potentially causing some confusion. Sparse Convolution in the context of 3D point clouds aims to maintain the inherent sparse pattern of the input by computing features only at specific output coordinates [264] and therefore it can be viewed as a form of masked convolution [253].

A sparse convolution layer of dimension D and kernel size K has weights of shape $K^D \times C_{in} \times C_{out}$, where C_{in} and C_{out} are the input and output feature sizes. Kernel offsets can be defined as the K^D vectors that describe the relative positions around the center of the kernel window (e.g for kernel size = 3 the kernel offsets are the 3^D vectors of the form $\{-1, 0, 1\}^D$). The weights can be broken down into K^D matrices of shape $C_{in} \times C_{out}$, where each W_δ corresponds to the weights of a particular offset δ , thus called weight offset matrices.

For a convolution of stride s , an output point with coordinates q_i can be linked through an offset δ with an input point with coordinates p_j if $p_j = sq_i + \delta$. From this perspective, the features x_i of an output point q_i can be calculated as $x_i = \sum_{\delta} \sum_j 1[p_j = sq_i + \delta] (x_j \cdot W_\delta)$, where $1[\cdot]$ is the binary indicator of the condition $p_j = sq_i + \delta$ [230].

Following this observation, state-of-the-art implementations of sparse convolution compute as a first step an input-output map M , called kernel map, that has the mapping $\{p_j, q_i, \delta\}$ if $p_j = sq_i + \delta$ (*Map Step*) [230]. To check whether $sq_i + \delta$ is an input point, most frameworks use a hash table to store the input points and iteratively perform queries [230; 97; 26]. More recently, Minuet [264], introduced an innovative binary search-like approach to optimize the locality of the query operations.

It is important to note that this input-output mapping differs from the adjacency matrix used in graph convolution, as the latter uses the same weight matrix of shape $C_{in} \times C_{out}$ for all the different neighbors. On the contrary, in sparse convolution, an output point can be associated with different ‘neighboring’ input points through different offset and thus the computations would need different weight offset matrices. As a result, sparse libraries that accelerate graph convolution cannot be directly applied for sparse convolution [147; 230].

Sparse Convolution engines for 3D point cloud inference use the map step to result in dense computations. Each $\{p_j, q_i, \delta\}$ corresponds to the matrix multiplication of $x_i + = x_j W_\delta$, so iterating over all mappings will calculate the output features. However, matrix-vector multiplication tends to exhibit low utilization on GPUs. To address this, early implementations adopted the *Gather-Matmul-Scatter* dataflow [230]. In this process, all input feature vectors that share the same weight offset matrix are first gathered and concatenated into a contiguous feature matrix. A dense GEMM is then performed between this feature matrix and the weight offset matrix to produce partial sums. Finally, these partial sums are scattered and accumulated into their corresponding output feature vectors [230; 264].

An alternative dataflow is *Fetch-on-Demand* [147; 97], where instead of materializing the input feature matrix to a gather buffer in DRAM and the partial sums in a scatter buffer, it fetches on demand feature vectors on chip, performs a matrix multiplication and then directly writes the partial sum to the corresponding output. Thus, this dataflow achieves overlap between computations and memory accesses and can be viewed as a kernel fusion version of *Gather-Matmul-Scatter* [264].

Finally, a third dataflow introduced in [28] is *Implicit-GEMM*. By performing a im2col-like transformation on the input features, which is based on kernel map, sparse convolution can be viewed as a dense implicit GEMM of X_{im2col} and W , the total weight matrix of shape $(K^D C_{in} \times C_{out})$. Although it also overlaps memory accesses with computations through pipelining and minimizes write-back traffic, it still incurs noticeable redundant computations. TorchSparse++ [231] leverages the strengths of the different processing methods by proposing an adaptive computation dataflow selection.

C Mixture of Experts

Model scalability—specifically, increasing the number of parameters—is widely recognized as a key factor in improving the performance of Large Language Models (LLMs). However, compute resources pose a significant constraint in scaling such models. To balance quality with computational efficiency, it is often desirable to design models that contain a large number of parameters while minimizing the required compute. Mixture of Experts (MoE) [107; 217; 55] is a technique that enables the training of models with substantially more parameters, with far less compute compared to dense LLMs. MoE models can achieve comparable accuracy to their dense counterparts, while offering significantly faster training and inference times. In practice, this means that MoE models can scale in both model and data size to a larger extent under the same compute budget as dense models.

The core concept behind MoE layers is that each layer comprises multiple independent dense sub-networks, referred to as experts. Each sub-network or expert is responsible for processing a different subset of the input data and learns to specialize in a particular region of the input space. To route inputs appropriately, MoE layers incorporate a gating network—also known as a router—that determines which experts should be activated for a given input. During training, both the experts and the gating network are jointly trained. The gating network learns to assign each input to the most suitable expert(s), thereby enabling specialization.

Figure 28 shows an example of a sparse MoE layer. A sparse MoE layer consists of multiple experts (e.g., 8 experts), where each expert is usually implemented as a standard dense Feed-Forward Network (FFN) using conventional GEMM kernels. More advanced configurations are also possible, such as hierarchical MoEs, in which each expert is itself an MoE layer. A gating network (also referred to as a router) determines which experts are activated for each input token. In most cases, only a small subset of experts (e.g., top-2 or top-k) is selected per input token, meaning that each token is routed to one or a few experts, rather than all of them.

As a result, during training, the MoE technique enables significantly improved computational efficiency, as the gating network learns to effectively route input tokens to the most relevant experts. This efficiency extends to inference as well: although MoE models may contain a comparable number of parameters to dense LLMs, only a subset of these parameters—corresponding to the activated experts—is utilized for each input. Consequently, inference is substantially faster than in dense models with an equivalent parameter count. However, all experts must still reside in main memory (e.g., DRAM), resulting in a memory footprint that remains comparable to that of dense models.

In summary, unlike dense LLMs that process all input tokens through the entire network, MoEs introduce sparsity via conditional computation. This approach ensures that only a subset of the network—specifically, a few selected experts—is activated for each token. Although the activated experts are themselves dense feed-forward networks (FFNs) implemented using standard GEMM kernels, the overall sparse (conditional) computation enables substantial compute savings. This allows MoE models to scale to significantly larger parameter counts—improving accuracy and model quality—without a corresponding increase in computational cost.

D Quantization

Deep learning models continue to grow in depth and parameter count to improve task performance, address increasingly complex problems, and capture intricate patterns in the vast and ever-growing datasets available for training. Deploying such models in real-world, real-time scenarios often requires running them on resource-constrained hardware platforms, such as edge devices (e.g., mobile phones, smartwatches, and IoT devices). To tackle excessive memory footprint demands and provide computational efficiency in deep learning models, quantization [71] has become an increasingly prominent technique, particularly for Large Language Models (LLMs) [139; 37; 59; 258]. Quantization reduces the numerical precision of the values used to represent model parameters (weights) and activations—typically mapping high-precision formats like 32-bit floating point (FP32) to lower-precision formats such as 16-bit floating point (FP16), 8-bit integers (INT8), or even more aggressive formats such as 4-bit floating point (FP4) or 4-bit integer representations (INT4). This reduction in precision can enable the practical deployment of state-of-the-art models in constrained environments without significantly compromising accuracy.

Quantization can be applied to both model weights and activations, and by reducing the bit-width of these values, it offers several critical advantages for deep learning model deployment. First, quantization significantly reduces memory footprint requirements. For example, reducing numerical precision from FP32 to INT8 formats inherently yields a $4\times$ reduction in memory footprint. This reduction enables the deployment and execution of larger models on memory-constrained hardware (devices with limited memory capacity). Second, quantization facilitates faster inference. Lower-precision values reduce data movement and computational overhead, alleviating memory bandwidth pressure in modern computing systems and accelerating processing—particularly when lower-precision formats are supported by hardware. For instance, 8-bit integer operations are typically more computationally efficient than 32-bit floating-point operations on modern CPUs and GPUs [204; 114; 5; 211]. Moreover, recent hardware accelerators and GPU architectures are beginning to provide native support for even lower-precision formats, such as 4-bit floating point (NVIDIA Blackwell) [184], further enhancing performance. Third, reduced compute and memory usage directly translates to improved energy efficiency and less power requirements. This is especially beneficial for real-time inference on battery-powered or energy-constrained platforms. Finally, many mobile and embedded systems may lack dedicated floating-point units (FPUs), thus integer quantization plays a pivotal role in enabling the deployment of highly accurate deep learning models including LLMs, on such constrained devices—facilitating practical use cases in mobile phones, wearables, and other edge computing scenarios.

However, reducing numerical precision—by lowering the bit-width of values—can negatively impact model accuracy. To address this challenge, researchers have developed a variety of quantization methods for deep learning models [59; 37; 43; 126; 130], aiming to minimize or eliminate accuracy degradation.

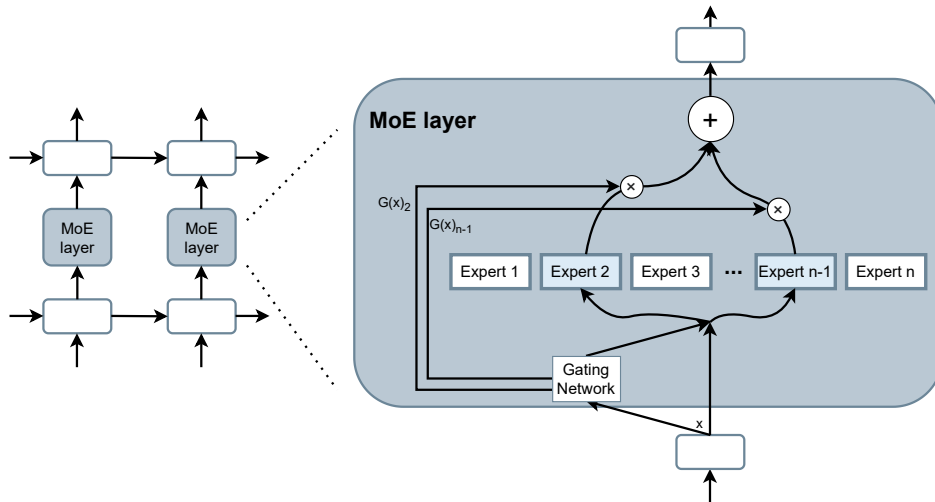


Figure 28: A sparse Mixture of Experts (MoE) layer.

With respect to how quantization is performed, there are two primary approaches:

1. **Quantization-Aware Training (QAT)** [106; 122; 47; 179]: In QAT, the model is trained with the target low-precision data type (e.g., INT8). Quantization is simulated during training, allowing the model to learn and adapt to quantization-induced noise. While QAT is effective at preserving model accuracy—especially for models with sensitive activation distributions—it requires access to the original training dataset and significantly increases training time.
2. **Post-Training Quantization (PTQ)** [122; 54; 153; 268]: PTQ applies quantization to an already-trained (pre-trained) model without modifying its original training process. While PTQ is simpler and less computationally demanding, it may result in larger accuracy drops compared to QAT. To mitigate this, some PTQ methods incorporate light fine-tuning [cite], using a small number of additional gradient update steps on a calibration dataset. Notably, this calibration dataset may differ from the original training dataset.

In terms of what to quantize, quantization methods are also classified as follows:

1. **Static Quantization** [106; 272; 243]: Both weights and activations are quantized prior to inference. A calibration dataset is used to estimate the dynamic range of activations, enabling the computation of appropriate scaling factors. This method tends to provide superior inference performance and efficiency, but requires access to a representative calibration dataset.
2. **Dynamic Quantization** [213; 260; 224; 227]: Weights are quantized offline, while activations remain in full precision (typically FP32) and are quantized dynamically at runtime (on-the-fly) during inference. Although dynamic quantization typically offers lower performance gains compared to static quantization, it is easier to apply and does not require a calibration dataset.

In summary, the choice of quantization method involves a trade-off between ease of application, computational cost, and the impact on model accuracy. For deployment in resource-constrained environments, the combination of static quantization and QAT often yields the most efficient and accurate results, albeit at the cost of increased complexity during model preparation.

Recent research [128; 83; 17] has shown that quantization to lower bit-widths can naturally induce sparsity in deep learning tensors. This effect arises because quantization compresses the range of values of tensors, causing small-magnitude elements—those falling below a certain threshold—to be rounded to zero. As a result, quantized tensors often contain a substantial proportion of zero-valued elements. For example, [17] demonstrate that quantizing state-of-the-art diffusion models to 4-bit floating point (FP4) introduces up to 34% sparsity in the model weights. Similarly, [128] show that the sparsity ratio increases as the bit-width decreases: quantizing tensors to 8-bit, 4-bit, and 2-bit formats leads to approximately 13%, 35%, and 59% sparsity, respectively.

As deep learning models—particularly LLMs—increasingly adopt aggressive quantization strategies (e.g., 8-bit, 4-bit, or even 2-bit precision) [16; 139; 188], induced sparsity is expected to become more pronounced. This trend presents a growing opportunity for systems and architecture researchers to design hardware and software optimizations that leverage sparsity to improve inference efficiency.

However, it is important to note that sparsity induced by quantization typically ranges from 30% to 65% [17; 128], which is substantially lower than the inherent sparsity observed in certain data domains. For instance, input data in GNNs often exhibit sparsity levels ranging from 85% to 99% [72]. As such, while quantization-induced sparsity offers meaningful performance benefits, it may require different optimization strategies compared to naturally sparse data such as graph data.

Recent work [83] investigates the interplay between quantization and pruning, demonstrating that these two techniques are not orthogonal; rather, their combination can amplify sparsity—producing a large number of zero-valued elements in tensors. This effect arises because quantization can distort the relative magnitudes and importance of tensor elements, which can influence the selection of elements during subsequent pruning.

Additionally, quantization can naturally introduce sparsity by forcing values that are close to zero to become exactly zero, thereby directly increasing the number of zero-valued elements in tensors. The study further shows that applying pruning followed by quantization, or vice versa, both lead to sparsity in the resulting quantized tensors. However, through both theoretical analysis and empirical evaluations, the authors find that applying pruning prior to quantization is the more optimal sequence. This ordering minimizes quantization-induced error and better preserves overall model accuracy. In contrast, applying quantization before pruning can degrade performance significantly, as quantization may obscure the true importance of certain weights, leading to the erroneous removal of critical parameters during the pruning step.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.
- [2] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.
- [3] AMD. AOCL-Sparse. <https://www.amd.com/en/developer/aocl/sparse.html>, 2024.
- [4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.
- [5] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. Nvidia hopper architecture in-depth. *NVIDIA Technical Blog*, 2022.
- [6] ARM. Arm performance libraries sparse algebra. <https://developer.arm.com/documentation/101004/2507/Sparse-Linear-Algebra>, 2025.
- [7] Iro Armeni, Ozan Sener, Amir R Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1534–1543, 2016.
- [8] Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, and Samuel Webb Williams. The landscape of parallel computing research: A view from berkeley. Technical report, University of California, Berkeley, 2006.
- [9] Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 688–697. IEEE, 2017.
- [10] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William Gropp, et al. *PETSc users manual*. Argonne National Laboratory, 2019.
- [11] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 9297–9307, 2019.
- [12] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
- [13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. Jax: composable transformations of python+numpy programs. <http://github.com/jax-ml/jax>, 2018.
- [14] Aydin Buluç and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.
- [15] Roberto L Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B Fraguera, and Torsten Hoefler. Venom: A vectorized n: M format for unleashing the power of sparse tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2023.

-
- [16] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36: 4396–4429, 2023.
 - [17] Cheng Chen, Christina Giannoula, and Andreas Moshovos. Low-bitwidth floating point quantization for efficient high-quality diffusion models. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 181–193. IEEE, 2024.
 - [18] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.
 - [19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.
 - [20] Xuhao Chen. Escoin: Efficient sparse convolutional neural network inference on gpus. *arXiv preprint arXiv:1802.10280*, 2018.
 - [21] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. fusegcn: Accelerating graph convolutional neural network training on gpgpu. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–9, 2020.
 - [22] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.
 - [23] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. Vectorizing sparse matrix computations with partially-strided codelets. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.
 - [24] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.
 - [25] Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, David Belanger, Lucy Colwell, et al. Masked language modeling for proteins via linearly scalable long-context transformers. *arXiv preprint arXiv:2006.03555*, 2020.
 - [26] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 3075–3084, 2019.
 - [27] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
 - [28] Spconv Contributors. Spconv: Spatially sparse convolution library. <https://github.com/traveller59/spconv/>, 2022.
 - [29] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. The rising costs of training frontier ai models. *arXiv preprint arXiv:2405.21015*, 2024.
 - [30] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 595–600, 2022.
 - [31] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

-
- [32] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *Advances in neural information processing systems*, 33: 4271–4282, 2020.
 - [33] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.
 - [34] Paresh Dave and Will Knight. Meta’s next llama ai models are training on a gpu cluster ‘bigger than anything’ else. <https://www.wired.com/story/meta-llama-ai-gpu-training/>, October 2024.
 - [35] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
 - [36] Deepspeed. Sparse Attention. <https://www.deepspeed.ai/2020/09/08/sparse-attention.html/>, 2025.
 - [37] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35:30318–30332, 2022.
 - [38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.
 - [39] dgSPARSE. dgSPARSE-Lib. <https://github.com/dgSPARSE/>, 2024.
 - [40] KHGGP Dollár, Ross Girshick, et al. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017.
 - [41] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
 - [42] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.
 - [43] Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. *arXiv preprint arXiv:2401.06118*, 2024.
 - [44] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms. *ACM Transactions on Mathematical Software (TOMS)*, 44(3):1–32, 2018.
 - [45] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
 - [46] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14629–14638, 2020.
 - [47] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.
 - [48] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International conference on machine learning*, pp. 2943–2952. PMLR, 2020.

-
- [49] Hugging Face. Hugging Face Hub. <https://huggingface.co/>, 2025.
 - [50] Hugging Face. TGI (Text Generation Inference). <https://github.com/huggingface/text-generation-inference/>, 2025.
 - [51] Ruibo Fan, Wei Wang, and Xiaowen Chu. Dtc-spm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 253–267, 2024.
 - [52] Zichen Fan, Steve Dai, Rangharajan Venkatesan, Dennis Sylvester, and Brucek Khailany. Sq-dm: Accelerating diffusion models with aggressive quantization and temporal sparsity. *arXiv preprint arXiv:2501.15448*, 2025.
 - [53] Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. Maskllm: Learnable semi-structured sparsity for large language models. *Advances in Neural Information Processing Systems*, 37:7736–7758, 2024.
 - [54] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph H Hassoun. Post-training piecewise linear quantization for deep neural networks. In *European conference on computer vision*, pp. 69–86. Springer, 2020.
 - [55] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.
 - [56] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.
 - [57] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.
 - [58] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*, pp. 10323–10337. PMLR, 2023.
 - [59] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
 - [60] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefer, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 239–251, 2025.
 - [61] Dimitrios Galanopoulos, Panagiotis Mpakos, Petros Anastasiadis, Nectarios Koziris, and Georgios Goumas. Div: An index & value compression method for spmv on large matrices. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 705–717, 2025.
 - [62] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.
 - [63] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.
 - [64] Mark Gales, Steve Young, et al. The application of hidden markov models in speech recognition. *Foundations and Trends® in Signal Processing*, 1(3):195–304, 2008.
 - [65] Yizhao Gao, Shuming Guo, Shijie Cao, Yuqing Xia, Yu Cheng, Lei Wang, Lingxiao Ma, Yutao Sun, Tianzhu Ye, Li Dong, et al. Seerattention-r: Sparse attention adaptation for long reasoning. *arXiv preprint arXiv:2506.08889*, 2025.

-
- [66] Georgios Georgiadis. Accelerating convolutional neural networks via activation map compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7085–7095, 2019.
 - [67] Athanasios Glentis Georgoulakis, George Retsinas, and Petros Maragos. Feather: An elegant solution to effective dnn sparsification. *arXiv preprint arXiv:2310.02448*, 2023.
 - [68] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. Spade: A flexible and scalable accelerator for spmm and sddmm. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–15, 2023.
 - [69] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. Hottiles: Accelerating spmm with heterogeneous accelerator architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1012–1028. IEEE, 2024.
 - [70] Benyamin Ghogh and Ali Ghodsi. Recurrent neural networks and long short-term memory networks: Tutorial and survey. *arXiv preprint arXiv:2304.11461*, 2023.
 - [71] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*, pp. 291–326. Chapman and Hall/CRC, 2022.
 - [72] Christina Giannoula, Peiming Yang, Ivan Fernandez, Jiacheng Yang, Sankeerth Durvasula, Yu Xin Li, Mohammad Sadrosadati, Juan Gomez Luna, Onur Mutlu, and Gennady Pekhimenko. Pygim: An efficient graph neural network library for real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–36, 2024.
 - [73] Yidong Gong and Pradeep Kumar. Gnnone: A unified system optimizations for gnn kernels. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pp. 15–27, 2024.
 - [74] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11): 139–144, 2020.
 - [75] Benjamin Graham, Martin Engelcke, and Laurens Van Der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 9224–9232, 2018.
 - [76] Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3(2):2, 2017.
 - [77] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Masr: A modular accelerator for sparse rnns. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–14. IEEE, 2019.
 - [78] Deniz Gurevin, Mohsin Shan, Shaoyi Huang, MD Amit Hasan, Caiwen Ding, and Omer Khan. Prunegnn: Algorithm-architecture pruning framework for graph neural network acceleration. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 108–123. IEEE, 2024.
 - [79] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.
 - [80] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
 - [81] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

-
- [82] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. ESE: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 75–84, 2017.
 - [83] Simla Burcu Harma, Ayan Chakraborty, Elizaveta Kostenok, Danila Mishin, Dongho Ha, Babak Falsafi, Martin Jaggi, Ming Liu, Yunho Oh, Suvinay Subramanian, et al. Effective interplay between sparsity and quantization: From theory to practice. *arXiv preprint arXiv:2405.20935*, 2024.
 - [84] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pp. 293–299. IEEE, 1993.
 - [85] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.
 - [86] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
 - [87] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.
 - [88] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.
 - [89] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991. IEEE, 2016.
 - [90] D Hendrycks. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
 - [91] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.
 - [92] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
 - [93] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.
 - [94] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
 - [95] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 66–79, 2018.
 - [96] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and Ponnuswamy Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp. 300–314, 2019.
 - [97] Ke Hong, Zhongming Yu, Guohao Dai, Xinhao Yang, Yaoxiu Lian, Ningyi Xu, and Yu Wang. Exploiting hardware utilization and adaptive dataflow for efficient sparse convolution in 3d point clouds. *Proceedings of Machine Learning and Systems*, 5:428–441, 2023.

-
- [98] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.
 - [99] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.
 - [100] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13. IEEE, 2020.
 - [101] Zhengding Hu, Jingwei Sun, and Guangzhong Sun. Gnnpilot: A holistic framework for high-performance graph neural network computations on gpus. *ACM Transactions on Architecture and Code Optimization*, 2025.
 - [102] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.
 - [103] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmv: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2020.
 - [104] Weiyu Huang, Yuezhou Hu, Guohao Jian, Jun Zhu, and Jianfei Chen. Pruning large language models with semi-structural adaptive sparse training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 24167–24175, 2025.
 - [105] Intel. OpenVINO. <https://github.com/openvinotoolkit/openvino/>, 2025.
 - [106] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.
 - [107] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.
 - [108] Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. Perceiver: General perception with iterative attention. In *International conference on machine learning*, pp. 4651–4664. PMLR, 2021.
 - [109] Hrvoje Jasak. Openfoam: Open source cfd in research and industry. *International journal of naval architecture and ocean engineering*, 1(2):89–94, 2009.
 - [110] Siddhant Jayakumar, Razvan Pascanu, Jack Rae, Simon Osindero, and Erich Elsen. Top-kast: Top-k always sparse training. *Advances in Neural Information Processing Systems*, 33:20744–20754, 2020.
 - [111] Nidhal Jegham, Marwan Abdelatti, Chan Young Koh, Lassad Elmoubarki, and Abdeltawab Hendawi. How hungry is ai? benchmarking energy, water, and carbon footprint of llm inference. *arXiv preprint arXiv:2505.09598*, 2025.
 - [112] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.
 - [113] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 376–388, 2020.

-
- [114] Leela S Karumbunathan. Nvidia jetson agx orin series. *A Giant Leap Forward for Robotics and Edge AI Applications. Technical Brief*, 2022.
 - [115] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.
 - [116] Randy H Katz, David E Culler, Seth Sanders, Sara Alspaugh, Yanpei Chen, Stephen Dawson-Haggerty, Prabal Dutta, Mike He, Xiaofan Jiang, Laura Keys, et al. Sustainable computing: Informatics and systems, 2010.
 - [117] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
 - [118] TN Kipf. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
 - [119] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.
 - [120] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.
 - [121] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
 - [122] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.
 - [123] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
 - [124] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and Ponnuswamy Sadayappan. Efficient tiled sparse matrix multiplication through matrix signatures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.
 - [125] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.
 - [126] Eldar Kurtic, Alexandre Marques, Shubhra Pandit, Mark Kurtz, and Dan Alistarh. "give me bf16 or give me death"? accuracy-performance trade-offs in llm quantization. *arXiv preprint arXiv:2411.02355*, 2024.
 - [127] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning*, pp. 5533–5543. PMLR, 2020.
 - [128] Andrey Kuzmin, Markus Nagel, Mart Van Baalen, Arash Behboodi, and Tijmen Blankevoort. Pruning vs quantization: Which is better? *Advances in neural information processing systems*, 36:62414–62427, 2023.
 - [129] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

-
- [130] Alberto Delmas Lascorz, Mostafa Mahmoud, Ali Hadi Zadeh, Milos Nikolic, Kareem Ibrahim, Christina Giannoula, Ameer Abdelhadi, and Andreas Moshovos. Atalanta: A bit is worth a “thousand” tensor values. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 85–102, 2024.
 - [131] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.
 - [132] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002.
 - [133] Namhoon Lee, Thalaisyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.
 - [134] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, et al. xformers: A modular and hackable transformer modelling library, 2022.
 - [135] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.
 - [136] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.
 - [137] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.
 - [138] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.
 - [139] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158*, 2024.
 - [140] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Yunjun Gao, and Yaofeng Tu. Hc-spm: Accelerating sparse matrix-matrix multiplication for graphs with hybrid gpu cores. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pp. 501–514. IEEE, 2025.
 - [141] Zitong Li and Aparna Chandramowlishwaran. Fused3s: Fast sparse attention on tensor cores. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 104–118, 2025.
 - [142] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022.
 - [143] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, et al. Efficient gpu kernels for n: M-sparse weights in deep learning. *Proceedings of Machine Learning and Systems*, 5:513–525, 2023.
 - [144] Junqing Lin, Honghe Zhang, Xiaolong Shi, Jingwei Sun, Xianzhi Yu, Jun Yao, and Guangzhong Sun. Ec-spm: Efficient compilation of spmm kernel on gpus. In *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 21–30, 2023.
 - [145] Junqing Lin, Jingwei Sun, Xiaolong Shi, Honghe Zhang, Xianzhi Yu, Xinzhi Wang, Jun Yao, and Guangzhong Sun. Lo-spm: Low-cost search for high-performance spmm kernels on gpus. *ACM Transactions on Architecture and Code Optimization*, 21(4):1–25, 2024.
 - [146] Junqing Lin, Jingwei Sun, Mingge Lu, and Guangzhong Sun. Toward efficient spmv in sparse llms via block extraction and compressed storage. *arXiv preprint arXiv:2507.12205*, 2025.

-
- [147] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. Pointacc: Efficient point cloud accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–461, 2021.
- [148] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [149] Hongyi Liu, Rajarshi Saha, Zhen Jia, Youngsuk Park, Jiaji Huang, Shoham Sabach, Yu-Xiang Wang, and George Karypis. Proxsparse: Regularized learning of semi-structured sparsity masks for pretrained llms. *arXiv preprint arXiv:2502.00258*, 2025.
- [150] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. *arXiv preprint arXiv:1801.10198*, 2018.
- [151] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Li Shen, Decebal Constantin Mocanu, Zhangyang Wang, and Mykola Pechenizkiy. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training. *arXiv preprint arXiv:2202.02643*, 2022.
- [152] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 339–350, 2015.
- [153] Xingchao Liu, Mao Ye, Dengyong Zhou, and Qiang Liu. Post-training quantization with multiple points: Mixed precision without mixed precision. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 8697–8705, 2021.
- [154] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pp. 2736–2744, 2017.
- [155] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.
- [156] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023.
- [157] Mohammad Loni, Aditya Mohan, Mehdi Asadi, and Marius Lindauer. Learning activation functions for sparse neural networks. In *International Conference on Automated Machine Learning*, pp. 16–1. PMLR, 2023.
- [158] Chao Lou, Zixia Jia, Zilong Zheng, and Kewei Tu. Sparser is faster and less is more: Efficient sparse attention for long-range transformers. *arXiv preprint arXiv:2406.16747*, 2024.
- [159] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.
- [160] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.
- [161] Cong Ma, Du Wu, Zhelang Deng, Jiang Chen, Xiaowen Huang, Jintao Meng, Wenxi Zhu, Bingqiang Wang, Amelie Chi Zhou, Peng Chen, et al. Nm-spm: Accelerating matrix multiplication using n: M sparsity with gpgpu. *arXiv preprint arXiv:2503.01253*, 2025.
- [162] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, pp. 3. Atlanta, GA, 2013.
- [163] Neural Magic. DeepSparse. <https://github.com/neuralmagic/deepsparse/>, 2025.

-
- [164] Neural Magic. SparseZoo. <https://github.com/neuralmagic/sparsezoo/>, 2025.
 - [165] Neural Magic. nm-llvm. <https://github.com/neuralmagic/nm-vllm/>, 2025.
 - [166] Ben Mann, Nick Ryder, Melanie Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 1(3):3, 2020.
 - [167] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 13–20, 2017.
 - [168] Pierre-Emmanuel Mazaré, Gergely Szilvasy, Maria Lomeli, Francisco Massa, Naila Murray, Hervé Jégou, and Matthijs Douze. Inference-time sparse attention with asymmetric indexing. *arXiv preprint arXiv:2502.08246*, 2025.
 - [169] Fanxu Meng, Pingzhi Tang, Xiaojuan Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. Transmla: Multi-head latent attention is all you need. *arXiv preprint arXiv:2502.07864*, 2025.
 - [170] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.
 - [171] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.
 - [172] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):2383, 2018.
 - [173] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International conference on machine learning*, pp. 2498–2507. PMLR, 2017.
 - [174] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.
 - [175] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S, and Sachin S Sapatnekar. Gnnie: Gnn inference engine with load-balancing and graph-specific caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 565–570, 2022.
 - [176] Christopher Morris, Nils M Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*, 2020.
 - [177] Panagiotis Mpakos, Dimitrios Galanopoulos, Petros Anastasiadis, Nikela Papadopoulou, Nectarios Koziris, and Georgios Goumas. Feature-based spmv performance analysis on contemporary devices. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 668–679. IEEE, 2023.
 - [178] Rishav Mukherji, Mark Schöne, Khaleelulla Khan Nazeer, Christian Mayr, and Anand Subramoney. Activity sparsity complements weight sparsity for efficient rnn inference. *arXiv preprint arXiv:2311.07625*, 2023.
 - [179] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.
 - [180] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

-
- [181] NVIDIA. NVIDIA-cuSPARSE. <https://docs.nvidia.com/cuda/archive/12.5.0/cusparse/contents.html>, 2024.
 - [182] NVIDIA. TensorRT. <https://github.com/NVIDIA/TensorRT/>, 2025.
 - [183] NVIDIA. TensorRT-LLM. <https://github.com/NVIDIA/TensorRT-LLM/>, 2025.
 - [184] Nvidia. NVFP4 Trains with Precision of 16-Bit and Speed and Efficiency of 4-Bit. <https://developer.nvidia.com/blog/nvfp4-trains-with-precision-of-16-bit-and-speed-and-efficiency-of-4-bit/>, 2025.
 - [185] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. High performance unstructured spmm computation using tensor cores. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2024.
 - [186] ONNX. ONNX. <https://github.com/onnx/onnx/>, 2025.
 - [187] OpenAI. Triton. <https://github.com/triton-lang/triton/>, 2025.
 - [188] Andrei Panferov, Jiale Chen, Soroush Tabesh, Roberto L Castro, Mahdi Nikdan, and Dan Alistarh. Quest: Stable training of llms with 1-bit weights and activations. *arXiv preprint arXiv:2502.05003*, 2025.
 - [189] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. A row decomposition-based approach for sparse matrix multiplication on gpus. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 377–389, 2024.
 - [190] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.
 - [191] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR, 2018.
 - [192] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
 - [193] Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A Smith, and Lingpeng Kong. Random feature attention. *arXiv preprint arXiv:2103.02143*, 2021.
 - [194] Hongwu Peng, Deniz Gurevin, Shaoyi Huang, Tong Geng, Weiwen Jiang, Orner Khan, and Caiwen Ding. Towards sparsification of graph neural networks. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 272–279. IEEE, 2022.
 - [195] Eric Qin, Ananda Samajdar, Hyounkjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70. IEEE, 2020.
 - [196] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. *arXiv preprint arXiv:1911.02972*, 2019.
 - [197] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 2002.
 - [198] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Fusedmm: A unified sddmm-spm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 256–266. IEEE, 2021.

-
- [199] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- [200] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.
- [201] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.
- [202] George Retsinas, Athena Elafrou, Georgios Goumas, and Petros Maragos. Online weight pruning via adaptive sparsity loss. In *2021 IEEE International Conference on Image Processing (ICIP)*, pp. 3517–3521. IEEE, 2021.
- [203] AMD ROCm. rocSPARSE. <https://rocm.docs.amd.com/projects/rocSPARSE/en/latest/index.html>, 2025.
- [204] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Y Jim Kim, Haihao Shen, and Barukh Ziv. Lower numerical precision deep learning inference and training. *Intel White Paper*, 3(1):1–19, 2018.
- [205] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*, pp. 234–241, 2015.
- [206] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.
- [207] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [208] Bharat Runwal, Tejaswini Pedapati, and Pin-Yu Chen. From peft to defit: Parameter efficient finetuning for reducing activation density in transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 20218–20227, 2025.
- [209] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.
- [210] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.
- [211] D Salvator. H100 transformer engine supercharges ai training, delivering up to 6x higher performance without losing accuracy. <https://blogs.nvidia.com/blog/h100-transformer-engine/>, 2022.
- [212] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in neural information processing systems*, 33:20378–20389, 2020.
- [213] Gabriele Santini, Francesco Paissan, and Elisabetta Farella. A probabilistic framework for dynamic quantization. *arXiv preprint arXiv:2505.10689*, 2025.
- [214] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.
- [215] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pp. 593–607. Springer, 2018.
- [216] Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding, and Omer Khan. Mergepath-spm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 145–156. IEEE, 2023.

-
- [217] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.
 - [218] Kyuhong Shim, Iksoo Choi, Wonyong Sung, and Jungwook Choi. Layer-wise pruning of transformer attention heads for efficient language modeling. In *2021 18th International SoC Design Conference (ISOC)*, pp. 357–358. IEEE, 2021.
 - [219] Jiho Shin, Hoeseok Yang, and Youngmin Yi. Sparseinfer: Training-free prediction of activation sparsity for fast llm inference. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pp. 1–7. IEEE, 2025.
 - [220] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
 - [221] Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression. *Advances in Neural Information Processing Systems*, 33:18098–18109, 2020.
 - [222] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, et al. Prospare: Introducing and enhancing intrinsic activation sparsity within large language models. *arXiv preprint arXiv:2402.13516*, 2024.
 - [223] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–77, 2022.
 - [224] Zhuoran Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. Drq: dynamic region-based quantization for deep neural network acceleration. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1010–1021. IEEE, 2020.
 - [225] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
 - [226] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.
 - [227] Ximeng Sun, Rameswar Panda, Chun-Fu Richard Chen, Aude Oliva, Rogerio Feris, and Kate Saenko. Dynamic network quantization for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7375–7385, 2021.
 - [228] Yan Sun, Qixin Zhang, Zhiyuan Yu, Xikun Zhang, Li Shen, and Dacheng Tao. Maskpro: Linear-space probabilistic learning for strict (n: M)-sparsity on large language models. *arXiv preprint arXiv:2506.12876*, 2025.
 - [229] Taiji Suzuki, Hiroshi Abe, Tomoya Murata, Shingo Horiuchi, Kotaro Ito, Tokuma Wachi, So Hirai, Masatoshi Yukishima, and Tomoaki Nishimura. Spectral pruning: Compressing deep neural networks via spectral analysis and its generalization error. *arXiv preprint arXiv:1808.08558*, 2018.
 - [230] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. TorchSparse: Efficient Point Cloud Inference Engine. In *Conference on Machine Learning and Systems (MLSys)*, 2022.
 - [231] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. Torchsparse++: Efficient point cloud engine. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 202–209, 2023.
 - [232] TensorFlow. TensorFlow. <https://github.com/tensorflow/tensorflow/>, 2025.

-
- [233] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
 - [234] Antoine Vanderschueren and Christophe De Vleeschouwer. Are straight-through gradients and soft-thresholding all you need for sparse training? In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 3808–3817, 2023.
 - [235] Gaël Varoquaux, Sasha Luccioni, and Meredith Whittaker. Hype, sustainability, and the price of the bigger-is-better paradigm in ai. In *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency*, pp. 61–75, 2025.
 - [236] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
 - [237] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
 - [238] Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. Fast transformers with clustered attention. *Advances in Neural Information Processing Systems*, 33:21665–21674, 2020.
 - [239] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020.
 - [240] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pp. 167–188. Springer, 2014.
 - [241] Hongyu Wang, Mingzhen Li, Weile Jia, Hailong Yang, and Guangming Tan. Fastspmm: Leveraging tensor cores for sparse matrix multiplication. In *Proceedings of the 22nd ACM International Conference on Computing Frontiers*, pp. 195–204, 2025.
 - [242] Huan Wang, Can Qin, Yulun Zhang, and Yun Fu. Neural pruning via growing regularization. *arXiv preprint arXiv:2012.09243*, 2020.
 - [243] Jinguang Wang, Jingyu Wang, Haifeng Sun, Tingting Yang, Zirui Zhuang, Wanyi Ning, Yuexi Yin, Qi Qi, and Jianxin Liao. Mergequant: Accurate 4-bit static quantization of large language models by channel-wise calibration. *arXiv preprint arXiv:2503.07654*, 2025.
 - [244] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.
 - [245] Neng Wang, Chenghao Shi, Ruibin Guo, Huimin Lu, Zhiqiang Zheng, and Xieyuanli Chen. Insmos: Instance-aware moving object segmentation in lidar data. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 7598–7605. IEEE, 2023.
 - [246] Qi Wang, Yaobin Wang, Yi Luo, Rong Luo, and Pingping Tang. Hr-spmm: Adaptive row partitioning and hybrid kernel design for sparse matrix multiplication. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 161–172, 2025.
 - [247] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.
 - [248] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, pp. 515–531, 2021.

-
- [249] Ziheng Wang. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. In *Proceedings of the ACM international conference on parallel architectures and compilation techniques*, pp. 31–42, 2020.
- [250] Ziheng Wang. Sparsednn: Fast sparse deep learning inference on cpus. *arXiv preprint arXiv:2101.07948*, 2021.
- [251] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.
- [252] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.
- [253] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. Unified convolution framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems*, 5:666–679, 2023.
- [254] Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pp. 9621–9630, 2019.
- [255] Xiaoyang Wu, Li Jiang, Peng-Shuai Wang, Zhijian Liu, Xihui Liu, Yu Qiao, Wanli Ouyang, Tong He, and Hengshuang Zhao. Point transformer v3: Simpler faster stronger. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 4840–4851, 2024.
- [256] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285*, 2023.
- [257] Chang Xiao, Peilin Zhong, and Changxi Zheng. Enhancing adversarial defense by k-winners-take-all. *arXiv preprint arXiv:1905.10510*, 2019.
- [258] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099. PMLR, 2023.
- [259] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- [260] Yuhui Xu, Shuai Zhang, Yingyong Qi, Jiaxian Guo, Weiyao Lin, and Hongkai Xiong. Dnq: Dynamic network quantization. *arXiv preprint arXiv:1812.02375*, 2018.
- [261] Zeyu Xue, Mei Wen, Zhaoyun Chen, Yang Shi, Minjin Tang, Jianchao Yang, and Zhongdi Luo. Releasing the potential of tensor core for unstructured spmm using tiled-csr format. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 457–464. IEEE, 2023.
- [262] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [263] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pp. 672–687. Springer, 2018.
- [264] Jiacheng Yang, Christina Giannoula, Jun Wu, Mostafa Elhoushi, James Gleeson, and Gennady Pekhimenko. Minuet: Accelerating 3d sparse convolutions on gpus. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 786–802, 2024.

-
- [265] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.
 - [266] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33, 2020.
 - [267] Bingyi Zhang and Viktor Prasanna. Dynaspase: Accelerating gnn inference through dynamic sparsity exploitation. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 233–244. IEEE, 2023.
 - [268] Jinjie Zhang, Yixuan Zhou, and Rayan Saab. Post-training quantization for neural networks with provable guarantees. *SIAM journal on mathematics of data science*, 5(2):373–399, 2023.
 - [269] Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. Spargeattn: Accurate sparse attention accelerating any model inference. *arXiv preprint arXiv:2502.18137*, 2025.
 - [270] Kaige Zhang, Xiaoyan Liu, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. Jigsaw: Accelerating spmm with vector sparsity on sparse tensor core. In *Proceedings of the 53rd International Conference on Parallel Processing*, pp. 1124–1134, 2024.
 - [271] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.
 - [272] Zhao Zhang, Yangcheng Gao, Jicong Fan, Zhongqiu Zhao, Yi Yang, and Shuicheng Yan. Selectq: Calibration data selection for post-training quantization. *Machine Intelligence Research*, pp. 1–12, 2025.
 - [273] Zhenyu Zhang, Zechun Liu, Yuandong Tian, Harshit Khaitan, Zhangyang Wang, and Steven Li R-sparse. Rank-aware activation sparsity for efficient llm inference. In *Proc. International Conference on Learning Representations (ICLR)*, 2025.
 - [274] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, et al. Acc-spm: Accelerating general-purpose sparse matrix-matrix multiplication with gpu tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 326–338, 2025.
 - [275] Pengxiang Zhao, Hanyu Hu, Ping Li, Yi Zheng, Zhefeng Wang, and Xiaoming Yuan. Fistapruner: Layer-wise post-training pruning for large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 29457–29475, 2025.
 - [276] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. {SparTA}:{Deep-Learning} model sparsity via {Tensor-with-Sparsity-Attribute}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 213–232, 2022.
 - [277] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.