PIT: Optimization of Dynamic Sparse Deep Learning Models via Permutation Invariant Transformation

Ningxin Zheng*, Huiqiang Jiang*, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang
Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, Lidong Zhou
Microsoft Research

Abstract

Dynamic sparsity, where the sparsity patterns are unknown until runtime, poses a significant challenge to deep learning. The state-of-the-art sparsity-aware deep learning solutions are restricted to pre-defined, static sparsity patterns due to significant overheads associated with preprocessing. Efficient execution of dynamic sparse computation often faces the misalignment between the GPU-friendly tile configuration for efficient execution and the sparsity-aware tile shape that minimizes coverage wastes (non-zero values in tensor).

In this paper, we propose PIT, a deep-learning compiler for dynamic sparsity. PIT proposes a novel tiling mechanism that leverages Permutation Invariant Transformation (PIT), a mathematically proven property, to transform multiple sparsely located micro-tiles into a GPU-efficient dense tile without changing the computation results, thus achieving both high GPU utilization and low coverage waste. Given a model, PIT first finds feasible PIT rules for all its operators and generates efficient GPU kernels accordingly. At runtime, with the novel SRead and SWrite primitives, PIT rules can be executed extremely fast to support dynamic sparsity in an online manner. Extensive evaluation on diverse models shows that PIT can accelerate dynamic sparsity computation by up to 5.9x (average 2.43x) over state-of-the-art compilers.

CCS Concepts: • **Software and its engineering** → **Dynamic compilers**; • **Computer systems organization** → **Neural networks**.

Keywords: Deep learning compilers, Dynamic sparsity, Dynamic compilers, Transformation

* Equal contribution.

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613139

1 Introduction

Tensor is the key data abstraction in deep learning. It provides a powerful and flexible way to represent contents including images, audio, and language sentences. Deep learning computation mostly involves operations over tensors (e.g., matrix multiplications). To efficiently execute deep learning models, accelerators like GPUs have been designed to perform these tensor operations in parallel. GPUs usually have multiple Stream Multiprocessors (SM), each with hundreds of CUDA cores that can simultaneously perform arithmetic operations on different portions of the tensor. To efficiently utilize these parallel processing capabilities, deep learning compilers use tiling to break up tensors into smaller, regular tensor slices, a.k.a. tiles, that can be processed in parallel by multiple SMs. Tiling has been demonstrated to be a key optimization technique for tensor computations by effectively exploiting data locality and parallelism.

Recent developments suggest that deep learning computations are increasingly *sparse*, i.e., operations on tensors with many zeros. In addition to sparse model weights, which are often static and known a priori, more sparsity patterns are found to depend on inputs and are only known at runtime, i.e., dynamic sparsity. For example, large language models (e.g., GPT [16, 50], OPT [66]) exhibit various types of dynamic sparsity. Firstly, transformer models (and other types of models) show inherent dynamic sparsity in their inputs, activation and gradients [46]. Only a very small number of elements in the activation map are non-zero [44] (e.g., 3.0% for T5-Base [55] and 6.3% for ViT-B/16 [23]). Secondly, dynamic sparsity is being leveraged to help further scale DNN models to a large size. Most existing models with more than one trillion parameters adopt the mixture of experts (MoE) structure, which activates experts sparsely and dynamically depending on the input [29]. Moreover, dynamic sparse training, which dynamically prunes less important connections in the model during training, is attracting more attention for its superior computational efficiency [26, 46].

It is a daunting challenge to efficiently compute deep learning models in the presence of dynamic sparsity. Modern deep learning compilers [9, 20, 71], including the sparsity aware compilers [10, 68], leverage the sparsity patterns in deep learning models known at the compile time to find the right kernel configuration. They are infeasible to be used for dynamic sparsity due to high compilation time. Some fine-grained sparsity solutions can be executed at runtime [3, 31],

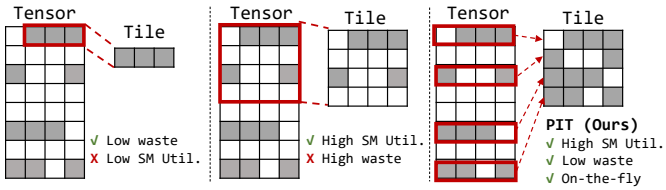


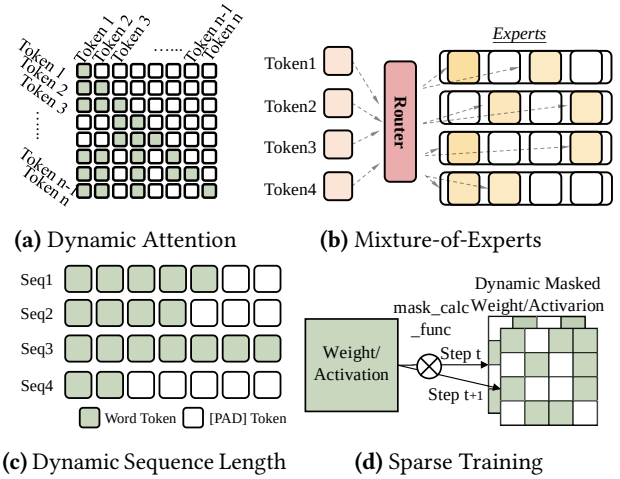
Figure 1. Tiling for dynamic sparsity (shaded blocks are non-zero values). Misalignment between sparsity granularity and the tile shape of efficient GPU kernels.

but they use special sparse formats (e.g., CSR: Compressed Sparse Row) incurring high overheads for format conversion.

Efficient GPU computation requires loading tiled data into shared memory for data reuse. However, the shape of hardware efficient tiles is usually misaligned with diverse and dynamic sparsity patterns. As shown in the left and mid figures in Figure 1, although sparsity-aligned tiles incur low coverage waste, they run slow on GPU due to low SM utilization and data reuse. But, since sparse values are usually non-continuously located, the tile shapes used in efficient GPU kernels cannot avoid covering a lot of zeros, leading to waste in sparse computation. Existing solutions for sparsity try to find better trade-offs between efficient tiling and sparsity shape alignment. They either require knowledge at the compile time, or incur significant overheads at runtime due to format conversation.

In this paper, we present PIT, a compiler for the efficient execution of deep learning models with dynamic sparsity. PIT resolves the misalignment of tile shapes by constructing GPU-efficient tiles with multiple sparsely located "micro-tiles" (e.g., the right figure in Figure 1). The construction is performed at runtime and its correctness is guaranteed mathematically by "Permutation-Invariant Transformation", a property widely existed in DL operators but is not well-exploited for dynamic sparsity. Most DL operators have one or more dimensions (we call them PIT-axis), whose computation can be arbitrarily reordered, without affecting the result. For example, in matrix multiplication (Matmul) whose tensor expression is $C[m,n]+=A[m,k]\times B[k,n]$, the columns of A along with the rows of B (i.e., the k dimension) can be permuted in any order without affecting the computation result. The rows of A along with the rows of C (i.e., the m dimension) can also be permuted without affecting the computation. By merging micro-tiles into a dense tile along a certain PIT-axis, PIT achieves the best of both worlds. It can leverage the efficient execution of dense tile at GPU SMs while achieving low wasted computation with the fine-grained micro-tile coverage of non-zeros.

A key challenge to leverage permutation-invariant transformation is the runtime transformation overhead. PIT solves the challenge by introducing *SRead* and *SWrite*, two data rearrangement primitives for loading and storing between sparsely located data in GPU global memory and the shared



(a) Dynamic Attention **(b)** Mixture-of-Experts

(c) Dynamic Sequence Length **(d)** Sparse Training

Figure 2. Examples of dynamic sparsity in deep learning.

memory. By piggybacking the data rearrangement on the data movement across the memory hierarchy during standard tensor computation, *SRead* and *SWrite* can achieve almost no extra overhead for permutation-invariant transformation. Even when the coordinates of sparse values in the tensors are unknown, PIT can detect them quickly in an online manner. Interestingly, by treating it as a special case of dynamic sparsity, PIT can also work effectively on deep learning models with static sparsity patterns where the misalignment of tile shapes is also a source of inefficiency.

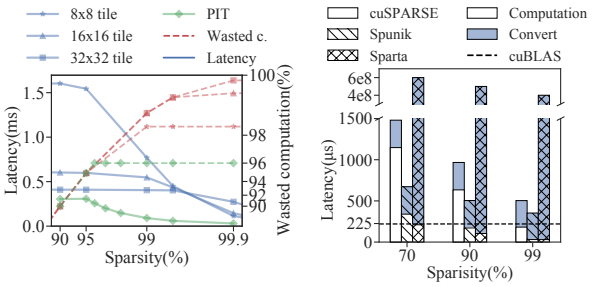
We extensively evaluate PIT on five representative models (i.e., Switch Transformer [63], OPT [66], BERT [22], LongFormer [14], Museformer [65]) and find that it significantly speeds up both inference and training while using less memory. In terms of inference, PIT achieve up to 5.9x speedup with less memory consumption compared to state-of-art solutions (§5.1). Additionally, we also applied PIT to speed up the large language model training and sparse training. PIT achieves up to 1.8x speedup on the OPT training, and up to 2.4x speedup for the sparse training compared to the previous solutions. Dynamic sparsity is becoming increasingly important in deep learning, and we plan to release PIT as open-source software to encourage further research on the optimization and algorithms on dynamic sparsity.

2 Background and Motivation

2.1 Dynamic Sparsity

As deep learning models become increasingly deep and large, parameters, activations, and gradients tend to approach zeros or become exactly zero, resulting in model sparsity. With the rapid development of Large Language Models (LLMs), it is a growing trend that many types of sparsity are shown to be dynamic, which is only known at runtime.

Dynamic Sparsity in LLMs. Dynamic attention allows models to compute only on the most informative parts of the data [29, 33, 34, 37, 38, 45, 57, 60, 69, 70, 72]. As shown in



(a) Latency and wasted computation of different tile sizes. **(b)** Conversion overheads.

Figure 3. Inefficient tiling of dynamic sparsity due to wasted computation and high sparse format conversion overhead.

Figure 2a, models can mask out irrelevant tokens, achieving higher accuracy with less computation. The masks are generated on the fly. Our evaluation also shows the activation outputs of three popular LLMs (OPT, Switch Transformer, and T5) have a sparsity ratio of 95-99.9% (i.e., percentage of zeros), which can be skipped without impacting the model accuracy (refer to §5.1).

Sparse Training. Sparse training is critical and widely used in many fields of deep learning, such as Dynamic Sparse Training (DST) [26], MAE [35], pruning [40], and supernet training[18, 19]. Figure 2d shows an example of dynamic pruning in training. The algorithm will mask out a portion of the weight according to the current model state. The mask in each step is constantly changing during training.

Mixture-of-Experts (MoE). MoE is a neural architecture that shows dynamic sparsity and is widely used in popular large models. Most LLMs over one-trillion parameters adopt MoE to scale up the model [24, 29, 42]. As shown in Figure 2b, a sequence of tokens passes through a gating function that assigns each token to expert(s) dynamically. Each expert handles only a proportion of the tokens of its expertise, by masking the other tokens not routed to it. Therefore, each expert's computation is sparse and depends on the input, which is only known at the runtime.

Dynamic Sequence Length. In natural language processing [22, 51, 55] and multi-modality models [53, 56], token sequences naturally have varied input and output lengths. Processing such sequences in batch requires padding them to the same sequence length (typically the maximum length in the batch), as shown in Figure 2c. Such padding leads to waste in computation and can be treated as dynamic sparsity.

2.2 Inefficiency Due to Dynamic Sparsity

Tiling in deep learning compilers. Deep learning compilers like TVM [20], Triton [9], Roller [71], often use tiling, a technique that slices a tensor into smaller tiles. By reusing cached tiles, tiling reduces the amount of data that needs to be transferred from slower memory like DRAM. By tuning tile shape (e.g., 32x32 or 16x64), compilers can optimize data reuse for a particular model computation on a specified hardware architecture, thus improving kernel performance.

Inefficient tiling in the presence of dynamic sparsity. Despite its effectiveness for conventional dense models, tiling can be inefficient for dynamic sparsity. As shown in Figure 1, tile shapes aligned with the sparse pattern minimize the coverage of non-zero values but are inefficient when executed on GPUs. On the contrary, GPU-efficient tiles introduce waste (covering too many zeros). Using actual sparse activations extracted from OPT [66] (a large language model), Figure 3a shows the performance of GPU kernels tuned with various tile shapes when executing a sparse matrix multiplication under different sparsity ratios. When the sparsity ratio is lower than 99.6%, 32x32 tiles are more efficient although it contains the most wasted coverage on zeros. The 8x8 tiles are faster only when sparsity is very high (>99.9% for this case) because of more saved computation than other tile shapes. Different tile shapes face the dilemma between sparsity-friendly tile coverage and GPU-efficient execution.

Sparsity-aware compilers or libraries. Some sparsity-aware compilers or libraries try to break the dilemma by compiling specialized GPU kernels (e.g., SparTA [68]), or transforming the data into a special format (e.g., cuSPARSE [3], Spunik [31]). They incur significant overheads that degrade runtime performance. Figure 3b compares the conversion overhead (compiling or format transformation) of SparTA, cuSPARSE, and Spunik when handling dynamic sparsity. SparTA takes 400-600 seconds to compile the specialized kernel, impractical to handle sparsity patterns changed at the runtime. Although cuSPARSE and Spunik can be used for dynamic sparsity, they suffer from large transformation overheads. As a result, the overall performance is even worse than directly using dense computation when the sparsity ratio is high and thus is inefficient to handle dynamic sparsity.

2.3 Opportunity

The right figure in Figure 1 shows that it is possible to load data sparsely located at different positions in parallel, and merge them into a dense tile with a shape efficient for computation. This sparse-to-dense transformation will not affect the correctness of the result. Figure 4 shows two examples of matrix multiplication. The first example multiplies a sparse tensor A with a dense tensor B. The shaded area has non-zero values and the rest are zeros. By rearranging the non-zero rows of A into a new tensor A', the computation can be conducted on dense matrix multiplication between A' and B' (the non-zero dense tile of B) producing the result tensor $C' = A' \times B'$. After writing the rows of C' to the original rows in C, we get the same result of $C = A \times B$. Note that, the rearrangement of A's non-zero rows can be in any permutation without affecting the correctness of each row in C'. The rows of C' are written back to C with the reverse permutation of

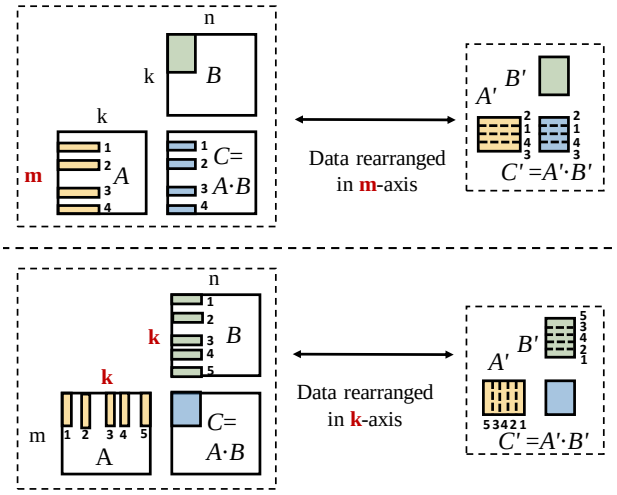


Figure 4. Examples of sparse matrix multiplication. By rearranging sparse data along an axis, the sparse computation can be equivalently done with dense matrix multiplication.

$A \rightarrow A'$ to restore the correct row indexes. The second example shows the matrix multiplication of two sparse matrixes A and B. By rearranging the data on the dimension k of A and B (i.e., columns of A and rows of B), the calculation can be done similarly using the matrix multiplication between two dense matrixes (i.e., A' and B'). As we have shown in Figure 3a, dense tiles are more GPU-efficient. If the data rearrangement in Figure 4 has negligible overhead at runtime, the dynamic sparse tensor computation can be conducted using dense tiles with low waste and high GPU efficiency, thus achieving superior performance. This motivates us to design PIT to systematically exploit this *permutation invariant transformation* of DL operators for efficient support to dynamic sparsity.

3 PIT Design

PIT is a compiler framework designed to address challenges introduced by dynamic sparsity. The core of PIT is its permutation invariant transformation (abbr. PIT transformation), which converts sparse tensors into a computation-efficient dense format in an online manner. Figure 5 illustrates PIT's architecture. PIT introduces micro-tile, a data unit with a minimum size used to compose a larger, hardware-friendly tile for efficient computation. Given sparse tensors, the PIT transformation policy identifies the most efficient micro-tile from all feasible micro-tiles derived from PIT rules, which are mathematically equivalent computation transformations of deep learning operators. The sparse kernel generator then creates the sparse kernel based on the selected micro-tile. To handle dynamic sparsity, the kernel takes sparse data and the index of micro-tiles with non-zero values for proper computation. The sparsity detector constructs the index online. The PIT property enables full parallelism in sparsity and index construction, minimizing the online execution overhead. In the sparse kernel, SRead and SWrite rearrange the sparse data into a dense format based on the constructed index, which is then processed by the highly efficient dense tile-based computation.

3.1 PIT Transformation Mechanism

Micro-tile. Micro-tile is a small data unit with a shape aligned with the read/write transaction granularity of the lower level memory of an accelerator (e.g., GPU). Micro-tile makes the access of sparse data as efficient as dense ones. For example, the read/write transaction of global memory in CUDA GPUs is 32 bytes, the smallest micro-tile size on this type of accelerator is 1x8 float32 (or 1x4 of float64), which is fine-grained enough for many sparse patterns. For example, in Figure 5, the input tensor has non-zero values in a granularity of 1x1, 1x2, 1x3, and 1x4. This can be covered with 1x4 micro-tiles, assuming 1x4 is the size of read/write transaction. This way, micro-tile achieves a good trade-off between computation efficiency and coverage waste.

An efficient dense tile has already been aligned with data access of GPU shared memory (e.g., minimizing bank conflict [71]) and saturated computation cores through well-optimized warp schedule [48]. By transforming the sparsely located micro-tiles to the required data format of the dense computation tile, the computation of a sparse operator is well aligned with every component of a GPU, including the global memory, the shared memory, and computation cores, thus achieving high efficiency. The transformation between sparse and dense computation tiles leverages permutation invariant transformation, a property that enables the dense tile to work on the rearranged micro-tiles correctly. Such a property commonly exists in deep learning operators, which will be elaborated in §3.2.

Figure 6 shows the definition of micro-tile on a sparse operator. It includes the micro-tile sizes for the operator's inputs/output and the dense computation tile, to which the micro-tiles are mapped. The attributes `TileInputFormats` and `TileOutputFormat` represent the data format (i.e., dense tile shapes) of the inputs and output respectively required by `DenseTileImpl`. A transformation policy (§3.2) determines such information, used for sparse kernel generation later.

SRead and SWrite. In contrast to dense tensor computations, whose tiles are loaded, processed, and stored continuously, PIT generates sparse kernels that employ SRead and SWrite to handle sparse data at the micro-tile level. As illustrated on the right of Figure 5, two primitives SRead and SWrite do online rearrangement of micro-tiles in input tensors to prepare data in `TileInputFormats` and write the data in `TileOutputFormat` to output micro-tiles respectively. The data rearrangement is piggybacked on the data movement across different memory levels, resulting in little additional overheads and eliminating the need for traditional

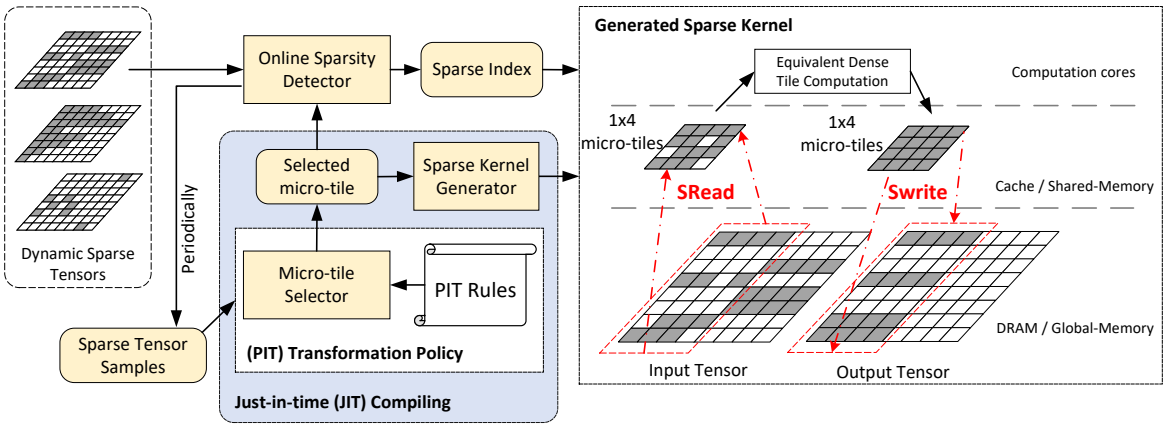


Figure 5. Architecture overview: PIT uses JIT compiling to generate sparse GPU kernels for dynamic sparse patterns. A sparse GPU kernel uses SRead to load micro-tiles from the input tensor (detected at runtime) to dense computation tiles at the shared memory. After the equivalent dense tile computation, it uses SWrite to write output micro-tiles to the output tensor. SRead and SWrite are conducted only on PIT-axis, where PIT guarantees the correctness of its data rearrangement.

```
class MicroTiledOp:
    # data format in global memory, A is sparse
    InputMicrotileSizes  # 1x4 for A, None for B
    OutputMicrotileSize  # None for C
    # data format in shared memory
    TileInputFormats     # 4x4 dense for A and B
    TileOutputFormat     # 4x4 dense for C
    DenseTileImpl        # C[4x4]=A[4x4]*B[4x4]
    def GenerateKernel()
```

Figure 6. The definition of micro-tiles for a sparse operator. The comments are the values of those attributes for the example micro-tile on the right of Figure 5.

```
/*Generated Sparse Kernel*/
__global__ void SparseKernelTemplate(
    struct Tensor Inputs, struct SparseIdx InIdx,
    struct Tensor Output, struct SparseIdx OutIdx,
){
    /* First allocate shared memory */
    InTiBlocks = AllocSharedM(TileInputFormats);
    OutTiBlock = AllocSharedM(TileOutputFormat);
    SRead(Inputs, InTiBlocks, InIdx);
    DenseTileImpl(InTiBlocks,OutTiBlock);
    SWrite(OutTiBlock, Output, OutIdx);
}
```

Figure 7. The sparse kernel template with SRead and SWrite.

data rearrangement outside the sparse kernel (e.g., constructing CSR format [17] for a sparse kernel).

Figure 7 illustrates the template of the PIT's sparse kernel. The kernel consists of two phases: data arrangement using SRead and SWrite, and the computation on dense tiles. Both the data rearrangement of SRead and SWrite require fast online construction of micro-tiles' indexes (i.e., `InIdx`, `OutIdx`) (§3.3). The indexes are constructed following the specified micro-tile shape defined in Figure 6. This design

Operator	Tensor Expression	PIT-axis
ReduceSum	$C[p]$ += $A[p,l]$	p, l
Vector Addition	$C[p] = A[p]+B[p]$	p
MatMul	$C[m,n]$ += $A[m,k]$*$B[k,n]$	m, n, k
BatchMatMul	$C[b,m,n]$ += $A[b,m,k]$*$B[b,k,n]$	b, m, n, k
Convolution	$C[n,f,x,y]$ += $A[n,m,x+i,y+j]$*$B[f,m,i,j]$	n, m, f

Table 1. Tensor expressions of widely-used operators and their PIT-axes that support shuffling their indexes without affecting correctness.

effectively separates data encoding/decoding from computation in the sparse kernel, introducing a novel sparse computation paradigm that combines data rearrangement with a (dense) computation tile.

3.2 PIT Policy

PIT defines a series of rules working on a certain tile axis that can correctly transform micro-tiles along this axis into GPU-efficient dense tiles. Specifically, a PIT rule contains the combination of a PIT-axis, a micro-tile shape, and a dense computation tile. Following a PIT rule, the system applies SRead/SWrite on the PIT-axis, loading/writing multiple sparsely located micro-tiles on this axis into/from the dense computation tile. A PIT rule ensures the computation on the PIX-axis must satisfy the permutation invariant property [41].

For ease of exposition, we use Einstein summation (einsum) notation [25] to express operations along tensor axes. Table 1 lists some common operators in deep learning and their corresponding einsum notations. An axis of an einsum notation is PIT-axis if and only if any shuffling of indexes on this axis does not affect the correctness of the operator. The following theorem finds all PIT-axes of an operator.

Theorem 1. *An axis is called PIT-axis, if and only if all computations on the axis are commutative and associative.*

The theorem is mostly self-evident, and we omit the proof due to space constraints. The commutative and associative property guarantees the correctness of random shuffling of micro-tiles on the PIT-axis and allows the parallel processing of micro-tiles in any order. First of all, the axes that derive new axes are not PIT-axes. For example, axes of x, i, y, and j in the convolution operator (Table 1) are not PIT-axes because they are not commutative due to the new axes ("$x+i$" and "$y+j$") derived by them. In the rest axes, there are two types of axes that are PIT-axes. An axis in the output tensor is called a *spatial axis*. All spatial axes are PIT-axes since they only change the data layout. An axis not in the output tensor is called a *reduction axis*, whose computations are mostly commutative and associative (e.g., sum, multiply, max, min), and thus PIT-axes. Table 1 also summarizes the PIT-axes of the listed operators. For every operator used by a model, PIT uses its tensor expression to find all PIT-axes based on the type of an axis, and whether it involves non-commutative or non-associative computations. We find only using one PIT-axis is general enough to cover most dynamic sparsity in deep learning. Although we do identify more complex PIT rules, e.g., permutation over multiple axes in (b, m)-axes or (b, n)-axes in BatchMatMul, we leave them to future works due to the limited space.

Micro-tile and Kernel Selection. PIT creates a database of sparse kernels, each of which applies PIT transformations on one PIT-axis of an operator. Each sparse kernel defines a micro-tile shape, which is determined by the PIT-axis and the tile shape of the dense kernel it uses for computation. When the memory layout of the sparse tensor is not contiguous on the PIT-axis, we set the shape of micro-tiles to 1 on the PIT-axis while keeping the shape of other axes the same as the tile shape of the dense kernel. This allows GPUs to load/write the values along the PIT-axis in parallel which can saturate the memory transaction. For example, consider a dense matrix-multiplication kernel with a tile size of $[M, K] \times [K, N]$, suppose the first input tensor is sparse and stored in the row-major memory layout (i.e., contiguous on K-axis in memory). If M is the PIT-axis, the micro-tile size will be $[1, K]$. If the memory layout of the sparse tensor is contiguous on the PIT-axis, we need to first change its format to make the data non-contiguous on the PIT-axis, e.g., from row-major to column-major, to saturate the memory transaction. This can be done in a piggyback manner at the output of previous operators generating this sparse tensor, thus its overhead is negligible.

Algorithm 1 shows how PIT selects the appropriate PIT-axis, micro-tile shape, and dense computation tile to generate the sparse kernels in a JIT manner. PIT iterates through all dense computation tiles and the PIT-axes of the operator (line 3-line 4). `GetTilesFromTileDB` returns all possible dense computation tile shapes with efficient GPU kernels, which are usually provided by existing DL compilers or implementation, e.g., TVM [20], OpenAI Block Sparse [6]. `GetPITAxis` returns all feasible PIT-axes of the operator we defined in Theorem 1. For each dense computation tile and PIX-axis, `GetMicroTile` finds the valid micro-tile shape as we elaborated above (line 6). `CoverAlgo` (line 8) calculates the number of micro-tiles required to cover all non-zero values in the sparse tensor (denoted as Num_{tiles}). The time cost of the generated sparse kernel is estimated as $Num_{tiles} \times T.tile\_cost$, where $T.tile\_cost$ is the running time of the corresponding sparse kernel via offline profiling. PIT selects the sparse kernel with the lowest time cost and its corresponding micro-tile to perform sparse computation. As for the input whose sparsity ratio is relatively low, this kernel selection algorithm makes PIT seamlessly fall back to the dense computation.

Algorithm 1: Kernel selection for a dynamic sparsity operator.

```
Data: Op: A dynamically sparse operator,
      D_sparse: A list of n sparsity samples of Op.
Result: Best: The best computation tile for Op.
1  Function KernelSelection(D_sparse, Op):
2      Best = null; Cost_optimal = inf;
3      foreach T ∈ GetTilesFromTileDB(Op) do
4          foreach A ∈ GetPITAxis(Op) do
5              Cost = 0;
6              micro_tile = GetMicroTile(T.SparseTensor, A);
7              foreach D ∈ D_sparse do
8                  Num_tiles = CoverAlgo(D, micro_tile, A);
9                  Cost += Num_tiles * T.tile_cost;
10             if Cost < Cost_optimal then
11                 Best = S;
12                 Cost_optimal = Cost;
13     return Best;
```

The offline profiling for the tile cost is lightweight. As PIT chooses to merge micro-tiles into a dense tile along a certain PIT-axis on-the-fly, it allows the offline profiling to be done in *a model, tensor shape, and sparsity pattern agnostic way*. PIT just records the execution time of different tile shapes (e.g., 32x32 and 64x64) for dense computation. Therefore, the offline profiling is conducted once per operator and per GPU type, which is very lightweight compared to long-running inference services.

3.3 Online Sparsity Detection

Efficient sparse computation requires the online detection of the changing sparsity pattern and computes only the non-zero values in sparse tensors. This implies that the index for the non-zero values should be constructed on-the-fly. However, creating sparse indexes on-the-fly can be challenging.

As illustrated in Figure 3b, cuSPARSE's conversion overhead can be significantly higher than computation time, particularly when sparsity is high.

We propose an effective mechanism for online index construction, i.e., constructing a sparse index at the granularity of *micro-tile* in an *unordered* manner. First, with the selected micro-tile, PIT detects non-zero values at the granularity of that micro-tile, which greatly reduces the size of the sparse index. Second, the PIT transformation enables sparsity detection and index construction to be highly concurrent in an out-of-order manner: the PIT-axis allows PIT to perform computations when an axis is permuted (§3.2). With this transformation, PIT no longer needs an ordered index along a specific axis. This substantially reduces constraints during the micro-tile based index construction, minimizing synchronization overheads across accelerator threads.

Specifically, the index construction task runs on the accelerator (e.g., GPU) with the construction task divided into tiles. For each tile, the task traverses a region of the sparse tensor and checks for the micro-tiles containing non-zero values. When a non-zero micro-tile is detected, its index (i.e., the offset within the tensor) is written to a pre-allocated index array. As multiple tiles concurrently update the index array with indexes of non-zero micro-tiles, they use `atomicadd` to determine unique positions in the array for recording these indexes, ensuring the safety of the update. Consequently, the resulting index arrangement for micro-tiles is unordered due to the unpredictable scheduling order of thread blocks.

Moreover, unlike existing sparse solutions (e.g., cuSPARSE, MegaBlocks), PIT constructs sparse indexes without changing the storage format of sparse tensors. At runtime, SRead and SWrite in sparse kernels use the index to load and rearrange the non-zero values directly from and to the original sparse tensors, at the micro-tile granularity. This significantly reduces memory access overheads introduced by data format conversion (e.g., from sparse data in a dense tensor to CSR format) and achieves zero-copy data rearrangement.

4 Implementation

We implement PIT and integrate it with PyTorch [52]. It consists of approximately 13,000 lines of C++ and CUDA code, and 5,600 lines of Python. PyTorch is a popular open-source DNN framework that supports various dynamic sparsity algorithms. PIT has generated approximately 1,500 sparse kernels by applying these PIT transformation rules to over 500 dense computation kernels, which include manually optimized kernels (such as OpenAI Block Sparse [6]), hardware instruction accelerated kernels (i.e., wmma [4]), and dense kernels generated by compilers like TVM [20]. These sparse kernels are stored in a database, and a performance look-up table is created in advance. This profiled performance is then used to guide online micro-tile selection. Although offline profiling takes several hours, it is done only once and can be accelerated by parallel profiling on multiple devices.

Models	Datasets	Model Structure	Precision	Devices
Switch Transformers[29]	MNLI [59]	Encoder Decoder MoE	fp16,fp32	A100
Swin-MoE [37]	ImageNet	Encoder MoE	fp16	A100
OPT [66]	Alpaca [58]	Decoder	fp32	V100
BERT [22]	GLUE [59], News [27] etc.	Encoder	fp32	V100
Longformer [14]	Arxiv [21]	Encoder	fp32	V100
MuseFormer [65]	LMD [54]	Decoder	fp32	V100

Table 2. Models and datasets in the evaluation.

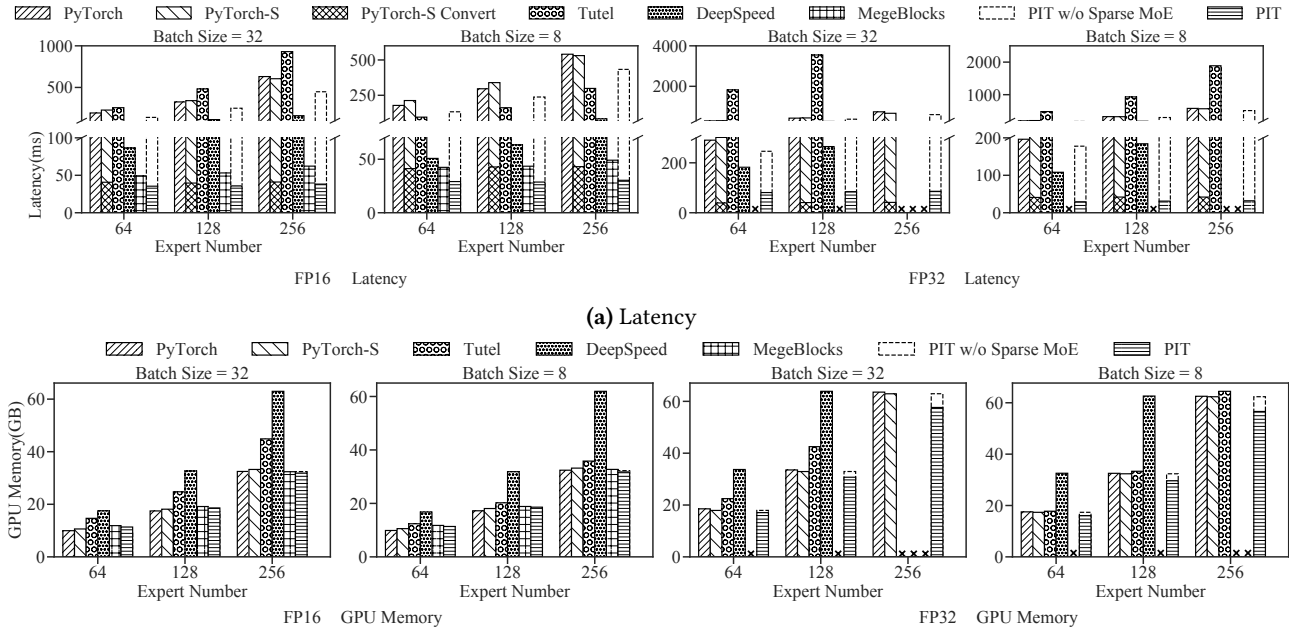
With the extensibility of dynamic sparsity optimizations, PIT have supported 31 Natural Language Processing (NLP) models. Thirteen (13) of them are large language models including OPT, T5, and Switch Transformer. Additionally, PIT supports 8 dynamic sparse attention models, 8 MoE models, and 13 sparse training algorithms. Integration of PIT is facilitated by its ability to accommodate minimal code modifications - with less than 10 lines of code changed in all evaluated scenarios. This feature enables swift adaptation of existing models for dynamic sparsity optimization, providing users with enhanced efficiency.

5 Evaluation

In this section we present comprehensive experiments to demonstrate the effectiveness of PIT from various perspectives. Specifically, we first evaluate the end-to-end inference performance of PIT with six representative models on both A100 and V100 (shown in §5.1). We also show the end-to-end performance of PIT across two distinct training scenarios (detailed in §5.2). Furthermore, micro-benchmarks are performed to highlight the effectiveness of PIT transformation (§5.3) and the conversion overheads (§5.4). Finally, we show the effectiveness of micro-tile online searching (§5.5) and detection of changing sparsity patterns (§5.6). In summary, our results show that:

- PIT achieves significant inference latency reduction and smaller memory footprints on five representative models, outperforming PyTorch, PyTorch sparse[1], Tutel, DeepSpeed, MegaBlocks, and TurboTransformer by up to 18.1x, 17.8x, 59.1x, 5.9x, 1.6x, and 1.9x respectively (§5.1).
- PIT also boosts the training efficiency significantly. Compared to the state-of-art solutions, PIT achieves up to 1.8x speedup for the OPT training, and up to 2.4x speedup for the sparse training (§5.2).

[1]The state-of-the-art sparse kernels wrapped in PyTorch.



(a) Latency

(b) GPU Memory

Figure 8. End-to-end latency per batch and memory footprints of Switch Transformer.

- With PIT transformation, PIT outperforms the state-of-art sparsity optimizations. Specifically, PIT achieves up to 88.7x, 5.8x, 17.5x, and 5.7x speedup over cuSPRARSE, Sputnik, OpenAI Block Sparse, SparTA respectively (§5.3).
- PIT can detect dynamic sparsity online with negligible overheads and achieves up to 4.7x speedup over previous state-of-art works when constructing the sparse index online (§5.4).

5.1 End-to-End inference

We compare PIT with state-of-the-art dense and sparse baselines on inference latency and memory usage for six representative models as shown in Table 2. The baselines include the most popular deep learning framework (PyTorch v1.11.0), two inference frameworks optimized for large-scale models (DeepSpeed [11] and TurboTransformers [28]), and model-specific optimization techniques (Tutel and MegaBlocks [30] for MoE models and Longformer-S for the Longformer model). We also create PyTorch-S, a variant of PyTorch that uses the best-performing sparse kernels from cuSPARSE (v11.6) [3], Sputnik [31], and Triton [9]. We select the best result among these sparse kernels for each model as the final performance of PyTorch-S. TurboTransformers only supports the BERT model and fails to run other models due to missing operators.

Switch Transformer. We evaluated the performance of PIT on Switch Transformer, a large language model that consists of Encoder, Decoder, and MoE structure at the same time. We measured the latency and memory usage of PIT on 1x A100-80GB GPU with different precisions, namely float32 and float16. The MoE layer of Switch Transformer assigns each token to one of the experts, which may produce an uneven token distribution among the experts. We compare PIT with several baselines: including PyTorch, which executes experts sequentially; Tutel and DeepSpeed, which use BatchMatmul to fuse all experts for parallel execution; MegaBlocks, which leverages sparse kernels to execute all experts simultaneously after reorganizing tokens in a sparse format. MegaBlocks only provides GPU kernels of float16 precision and thus is not evaluated in float32.

In the non-MoE layers, PIT optimizes dynamic sparsity caused by varying sequence lengths within the same batch. In the MoE layers, PIT employs SRead to load the relevant tokens for each expert, sparsely computes their assigned tokens, and writes the results directly to the corresponding positions using SWrite. We evaluated PIT using the MNLI dataset in GLUE [59]. Figure 8 shows end-to-end inference latency and memory cost of Switch Transformer with varying batch sizes and numbers of experts. As shown in Figure 8a, PIT outperformed other methods by allowing sparse calculation for all experts without computation waste. Compared to PyTorch, PyTorch-S, Tutel, DeepSpeed, PIT achieved 3.6x~18.1x, 3.7x~17.8x, 16.6x~59.1x, 2.3x~5.9x speedup respectively when the precision is float32. For the precision of float16, the speedup was 5.5x~17.8x, 6.4x~17.5x, 3.3x~24.3x, 1.8x~4.2x, and 1.4x~1.7x compared to PyTorch, PyTorch-S, Tutel, DeepSpeed, and MegaBlocks, respectively. Compared to Tutel and DeepSpeed, PIT avoids computational waste brought by the BatchMatmul, which requires padding the input of all experts to the same length. Moreover, compared to MegaBlocks, PIT uses SRead and SWrite to eliminate the

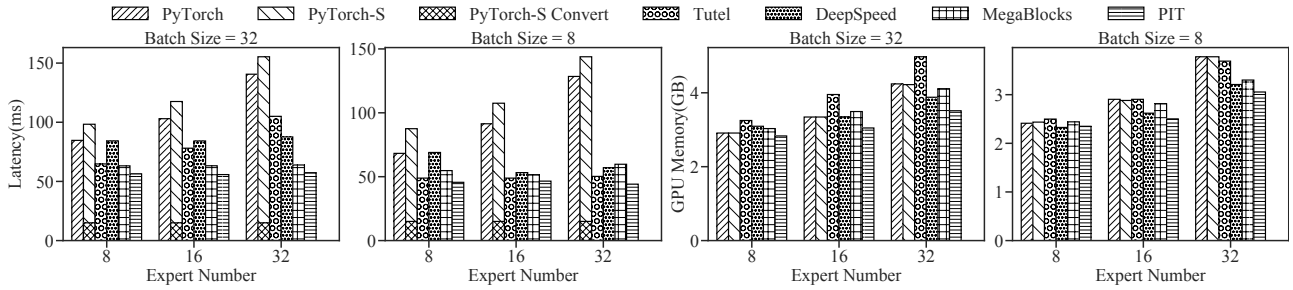


Figure 9. End-to-end latency and memory footprints of SwinMoE on A100.

expensive data reorganization cost during input preparation. "PyTorch-S Convert" highlights the sparse index construction overhead of PyTorch-S. Even though the computation in PyTorch-S has become faster, the cost of constructing sparse indices has neutralized the speed gains. To dissect the benefit under sparse MoE and varying sequence length, we also evaluated PIT without applying dynamic sparse MoE optimization (i.e., "PIT w/o Sparse MoE" in Figure 8. We find PIT's performance gain on Switch Transformer mainly comes from optimizing the dynamic sparsity in the MoE structure. We also evaluated the GPU memory usage shown in Figure 8b. PIT has the lowest memory usage compared to the baselines. Due to excessive padding when increasing the batch size and number of experts, Tutel and DeepSpeed run into Out-of-Memory (OOM).

Swin-MoE We assessed the performance of PIT on Swin-MoE, a large vision model with both Encoder and MoE structures. We measure the latency and memory usage of PIT on A100 under the float16 precision. For vision transformers, the input images within the same batch will be rescaled to the same resolution to achieve a consistent sequence length. In our experiments, shown in Figure 9, we compared Swin-MoE's end-to-end inference latency and memory footprints across different batch sizes and numbers of experts. MegaBlocks outperforms other baselines due to its simultaneous execution of all experts, efficiently utilizing sparse kernels to avoid computational waste. Compared to MegaBlocks, PIT further improves the performance by piggy-backing data reorganizations in the data movement across memory hierarchies. Compared to PyTorch, PyTorch-S, Tutel, DeepSpeed, MegaBlocks, PIT achieves 1.5x~6.3x, 1.5x~2.9x, 1.1x~1.8x, 1.2x~1.6x, 1.1x~1.4x speedup, respectively. PIT's performance improvement for Swin-MoE is less than that for Switch Transformer because the number of experts in Swin-MoE is significantly fewer than that in Switch-Transformer. As a result, the MoE layers only contribute 23.6% to 61.2% of the end-to-end latency when the number of experts varies from 8 to 32. When comparing the latency of the MoE layers alone, PIT is approximately 1.2x~1.7x faster than Megablocks. For a similar reason, PIT has a similar GPU memory usage compared to the baselines as shown on the right side of Figure 9.

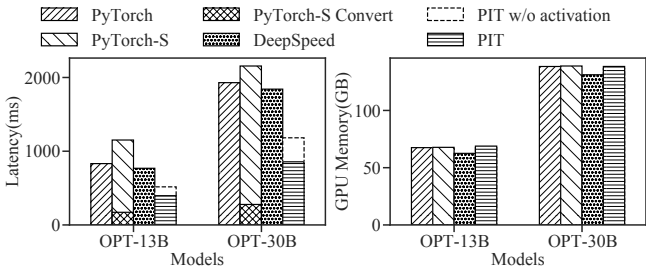


Figure 10. End-to-end latency per batch and memory footprints of OPT.

OPT is a decoder-only large language model [66]. We evaluated two versions with 13B and 30B parameters with the Alpaca [58] dataset on eight V100-32GB GPUs. PIT applies two dynamic sparsity optimizations on OPT: (1) eliminating the padding overhead from sentences with varying lengths in the same batch and (2) exploiting the fine-grained sparsity (up to 99%) created by the ReLU activation in the FFN layer. The batch size is set to 32. PyTorch-S uses Triton as the backend.

Figure 10 compares the end-to-end latency and memory footprints of the OPT. PIT outperforms PyTorch, PyTorch-S, and DeepSpeed by 2.1x~2.3x, 2.5x~3.0x and 2.0x~2.2x, respectively. The benefit of avoiding padding in dynamic sequences helps PIT to achieve 1.6x~1.7x speedup against the baselines (i.e., PIT w/o activation in Figure 10). By further exploiting the dynamic sparsity in the ReLU activation of FFN layers, PIT further boosts the performance by 1.3x~1.4x. In contrast to PyTorch-S, which uses Triton block sparse kernel of block size 32x32, PIT performs efficient computations using smaller micro-tile (i.e., 1x32) with SRead and SWrite, thus avoiding computation waste. Also, PyTorch-S suffers from the sparse format conversion overhead and thus has the highest latency. In terms of memory consumption, DeepSpeed has the lowest memory usage as it fuses the entire encoder layer into one operator and saves activation memory. PIT has a memory footprint similar to the other baselines.

BERT. We also tested PIT on BERT-base [22], an encoder-only language model, using Float32 on a single NVIDIA V100-32GB. In BERT, PIT only optimizes the dynamic sparsity

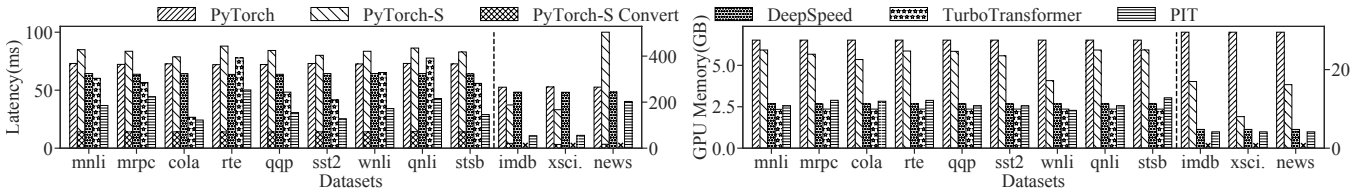


Figure 11. End-to-end latency and memory footprints of BERT on V100.

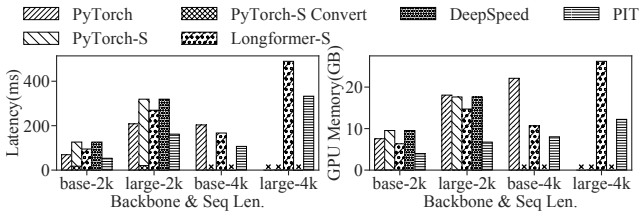


Figure 12. End-to-end inference latency and memory footprint of Longformer on V100.

caused by the different sequence lengths in the same batch. To evaluate its performance more comprehensively, we experimented with various datasets, including GLUE [59] (with mnli, mrpc, cola, rtc, qqp, sst2, wnli, qnli, stsb), IMDB [49], Multi-XScience [47], and Multi-News [27]. We add TurboTransformer [28] as a baseline, an inference framework that is specifically optimized for variable input sequence lengths by using smart dynamic batching. We used a batch size of 32. PyTorch-S employed Triton as the sparse backend library.

As Figure 11 illustrates, PIT outperforms PyTorch, PyTorch-S, DeepSpeed, and TurboTransformer by a factor of 1.3x to 4.9x, 1.8x~3.5x, 1.2x~4.5x, and 1.1x~1.9x, respectively. PyTorch-S performs poorly when the sequence lengths are short (e.g., no more than 128 tokens in GLUE). Because its backend (Triton) requires a coarse-grained sparsity (i.e., 32 tokens in Triton's kernel), PyTorch-S needs to pad the input sequence to multiple of 32. This incurs a high waste when the sequence lengths are short (e.g., a sequence of 16 tokens has to be padded to 32 causing a waste of 50%). TurboTransformer outperforms the other baselines by dividing the input into multiple small batches based on sentence lengths and processing them sequentially to avoid waste. PIT further outperforms TurboTransformer by processing the whole batch in parallel without waste. On memory usage, PIT consumes less memory than PyTorch and PyTorch-S, and similar memory to DeepSpeed and TurboTransformer. DeepSpeed and TurboTransformer optimize memory usage by fusing the entire layer into a single operator to reduce activation memory, which is compatible with PIT's sparsity optimizations. TurboTransformer crashes when the input sequence length increases due to kernel implementation issues.

Longformer is an encoder-only language model with dynamic sparse attention [14]. Longformer adaptively pays attention to several important words (e.g., class token) of the input. The position of dynamic attention varies for different inputs, which is the source of dynamic sparsity. Figure 12 illustrates Longformer's inference latency and memory cost for input sequence lengths of 2048 and 4096. PIT optimizes the dynamic sparsity in the dynamic sparse attention. To evaluate comprehensively, we also add the sparse implementation specifically optimized for the Longformer (represented by Longformer-S [1]). PyTorch-S selects Triton as the backend. Figure 12 shows the latency and GPU memory usage on a single V100. PIT is faster than PyTorch, Longformer-S, PyTorch-S, and DeepSpeed by up to 1.9x, 1.8x, 2.4x, and 2.4x, respectively. Longformer-S outperforms PyTorch-S because of its specifically optimized GPU kernels for its designed sparsity pattern through sparse pattern decomposition. However, its design is hard to be used by other models. DeepSpeed uses Triton to implement their sparse attention, so it has a similar performance to PyTorch-S. The index construction overhead of PyTorch-S is shown in "PyTorch-S Convert", which accounts for 6.3% ~13.9% of the end-to-end latency. Moreover, PyTorch-S performs even worse when it selects the fine-grained sparse library as the backend because the sparsity ratio is not high enough. When PyTorch-S selects Triton (Block Sparse) as the backend, it is slower than PIT due to the wasted computation caused by the dynamic global attention[14]. Longformer-S is more efficient since it has no computation waste by rearranging the input tensor, but it introduces large data rearrangement overheads. In contrast, PIT organizes the small micro-tiles on the fly with negligible overheads and computes them in an efficient dense computation tile directly without computation waste using SRead and SWrite. As for memory usage, PIT uses the least memory. PyTorch-S and DeepSpeed crashed due to out-of-memory when the input sequence length reached 4096. Both PyTorch-S and DeepSpeed have to use block sparse (32×32 in Triton) to cover all remaining values, leading to computation waste and a higher sparsity ratio. Longformer-S introduces extra memory cost due to its data re-arrangement, which creates many temporary intermediate tensors.

Museformer is a decoder-only language model that also generates the dynamic sparsity pattern according to the input data to improve model performance [65]. Figure 13 shows the inference latency and memory footprint under different input sequence lengths. PIT is 2.5x, 2.0x, and 2.0x faster

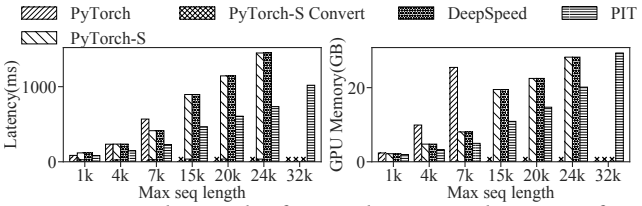


Figure 13. End-to-end inference latency and memory footprint of Museformer on V100.

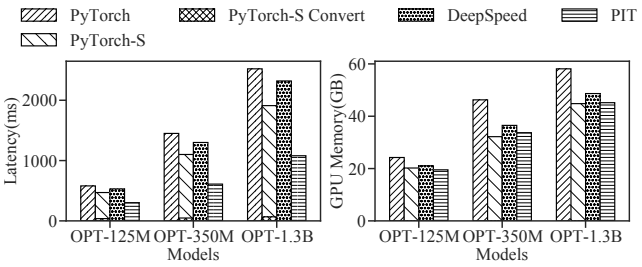


Figure 14. End-to-end latency per batch and memory footprints of OPT training.

than PyTorch, PyTorch-S, and DeepSpeed respectively before they crash due to out-of-memory. The time of sparse index construction accounts for up to 23.2% of the end-to-end latency in PyTorch-S for short sequences. As the input length increases, the amount of calculation becomes larger, and the time proportion of index construction can be gradually diluted. As for memory usage, PIT shows the lowest memory footprint. PyTorch consumes much more memory because it cannot understand and optimize the dynamic sparsity. Compared to PyTorch-S and DeepSpeed, PIT reduces computation waste by PIT transformation, resulting in lower memory consumption.

5.2 End-to-End training

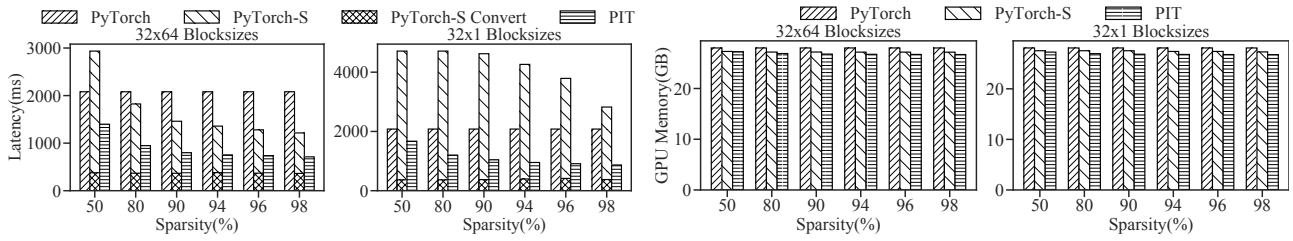
In this section, we evaluate PIT on both NVIDIA A100 and V100 GPUs to demonstrate its superior performance when using dynamic sparsity to accelerate training.

OPT Training. We fine-tuned the 125M, 350M, and 1.3B OPT models using the Alpaca dataset on an NVIDIA A100-80GB GPU. We utilized PIT to optimize the dynamic sparsity due to varying sentence lengths within the same batch. Due to memory limitations, we set the training batch size to 8. In our experiment, we compared the performance of PIT with PyTorch, PyTorch-S, and DeepSpeed. The results, as shown in Figure 14, demonstrate the time-cost and memory footprint of a forward and backward pass. PIT achieved 1.9x~2.4x, 1.6x~1.8x, and 1.8x~2.2x faster speed than PyTorch, PyTorch-S, and DeepSpeed, respectively. Similar to the inference optimization of varying sentence lengths, PIT saves the computation of padding in PyTorch and DeepSpeed. Compared to PyTorch-S, PIT supports more fine-grained sparsity granularity (1 token) than Triton's block sparse granularity (32 tokens) leading to more efficient computation. Also, PIT saves memory access overheads caused by reformatting data from dense to sparse formats in PyTorch-S. In terms of memory footprint, PIT and PyTorch-S have the smallest memory footprints during training with dynamic sparsity. Compared to inference, DeepSpeed cannot save the activation memory by fusing the entire layer into one operator in training, thus leading to more memory consumption.

Sparse Training. We use iterative pruning [43], a common approach of model compression, to demonstrate how PIT can leverage dynamic sparsity to speed up sparse training. At each step, the pruning algorithm generates a mask based on the weight's magnitude, which varies for different inputs. We evaluated our approach on the GLUE dataset [59] using the BERT model, which we pruned using block-wise sparsity at two granularities: 32×64 and 32×1. Figure 15a shows the time to process a batch of data (including both forward and backward passes) for each sparsity granularity setting, with a batch size of 32. PyTorch-S used Triton as the backend. When the sparsity granularity was 32×64, PIT achieved a 1.5x~3.0x and 1.7x~2.2x speedup over PyTorch and PyTorch-S, respectively. Although the sparsity granularity of 32×64 is larger than the computation tile of PyTorch-S's block sparse kernel (i.e., 32×32), the sparsity pattern of each layer constantly changed during pruning. It requires every layer of PyTorch-S to rebuild the sparse indices once per batch. This makes PyTorch-S suffer from heavy index construction overhead. PIT outperforms PyTorch-S on the granularity of 32×64 mainly due to its fast index construction.

Existing research has already shown a smaller sparsity granularity could bring higher accuracy but challenges execution performance optimization [40]. In our evaluation, using the granularity is 32×1, the pruned model achieved 0.26%~ 0.72% accuracy gain on the MNLI dataset and 0.11% ~ 1.26% accuracy gain on the SST-2 dataset. For latency, PIT outperformed PyTorch and PyTorch-S by 2.4x and 4.8x, respectively. The computation time of PyTorch-S increased significantly due to the misaligned data sparsity granularity (32×1) and the block-sparse GPU kernels (32×32 or 16×16). It is worth noting that PIT achieved almost the same speed as 32×64 when the sparsity granularity was set to 32×1, because PIT can use the fine-grained 32×1 micro-tile to cover the sparse data while using the 32×64 kernel for the most efficient computation, achieving the best of both worlds.

Figure 15b shows the memory usage during the pruning process. PIT used the least memory compared to PyTorch and PyTorch-S. The memory footprint only dropped slightly as the sparsity ratio increased, as the iterative pruning algorithm only prunes the model weights during the pruning, and weight tensors take up only a small fraction of memory. In contrast, PyTorch and PyTorch-S store the dense weights and gradients, so their memory footprint almost does not change with the sparsity ratio.



(a) Latency **(b)** Memory

Figure 15. End-to-end latency per batch and memory footprints of magnitude iterative pruning under 32×64 and 32×1 block.

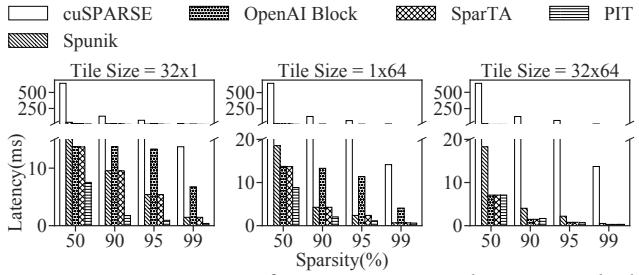


Figure 16. Comparison of cuSPARSE, Sputnik, OpenAI Block Sparse, SparTA and PIT on matrix multiplication ($4096 \times 4096 \times 4096$) under different sparsity ratios.

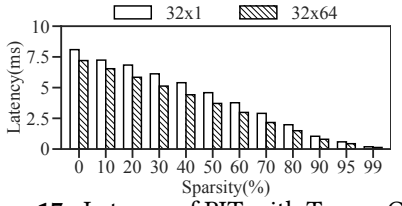


Figure 17. Latency of PIT with Tensor Core.

5.3 Effectiveness of PIT transformation

PIT transformation on dense kernels. In this study, we evaluate the performance of PIT's sparse matrix multiplication kernel with different sparsity granularities and shapes. To demonstrate the effectiveness of PIT transformation, we compare PIT with several other sparse libraries, including cuSPARSE, Sputnik, and OpenAI Block Sparse (Triton). We also introduce SparTA [68], a state-of-the-art sparse deep-learning compiler designed for static sparsity optimization. For this experiment, we use a static sparsity pattern to evaluate the computation efficiency, therefore the latency results shown in Figure 16 do not include conversion or compiling overhead. When the sparsity granularity is 32×64, PIT, SparTA, and OpenAI Block Sparse have similar latency because they use the same dense computation tile. cuSPARSE and Sputnik perform poorly due to their inefficient fine-grained computation granularity. When the sparsity granularity is 32×1 and 1×64, Sputnik and SparTA outperform cuSPARSE and OpenAI Block Sparse due to better granularity alignment. For the sparsity granularity of 32×1, PIT is 4.3x~5.8x faster than Sputnik and 1.5x~5.7x faster than SparTA. For the sparsity granularity of 64×1, PIT is 1.1x~2.3x faster than Sputnik and 1.1x~2.2x faster than SparTA. The PIT transformation contributes to the performance gain of PIT over Sputnik and SparTA by allowing PIT to perform the efficient computation even under a small sparsity granularity (i.e., 32×1) . In addition, PIT has a negligible overhead on SRead and SWrite running at a speed close to the original dense computation tile for different sparsity granularities.

PIT transformation on hardware instructions. In this experiment, we illustrate how PIT transformation can loosen the constraints on hardware instructions, like wmma [4]. We conduct sparse matrix multiplication on two different sparsity granularities (32×1 and 32×64) for a $[4096, 4096] \times [4096, 4096]$ matrix multiplication. The first input tensor is sparse and stored in column-major format. The wmma instruction only supports three shapes ($[16, 16] \times [16, 16]$, $[32, 8] \times [8, 16]$, $[8, 32] \times [32, 16]$) in half-precision, making it unsuitable for a 32×1 sparsity granularity. We use PIT transformation to derive two sparse kernels with micro-tiles of 32×1 and 32×64, respectively. We apply the sparse kernel with a micro-tile of 32×1 to perform sparse matrix multiplication with 32×1 sparsity granularity, while the kernel with a micro-tile of 32×64 is used for 32×64 sparsity granularity. Figure 16 shows the two sparse kernels generated by PIT have similar latency at different sparsity ratios. It proves PIT transformation introduces little overhead.

5.4 Conversion Overhead

In this experiment, we highlight the conversion overhead of PIT in more detail. Specifically, we evaluate the conversion overhead under different sparsity granularities and sparsity ratios on a V100-32GB GPU[2]. Figure 18 shows the sparse index construction time of PIT and PyTorch-S. PyTorch-S selects the index construction function provided by cuSPARSE when the granularity is 1×1 and the function provided by Triton when the granularity is 16×16 and 32×32. PIT is 3.6x~4.7x faster than cuSPARSE when the granularity is 1×1, 11.2x~14.2x faster than Triton when the granularity is

[2]Different from V100-32GB, PIT has a latency similar to PyTorch-S on V100-16GB for tile size 1x1. The performance advantage of PIT over other tile size on V100-16GB remains the same as in Figure 18.

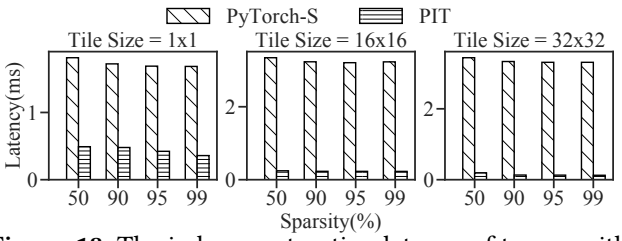


Figure 18. The index construction latency of tensor with shape of 4096×4096.

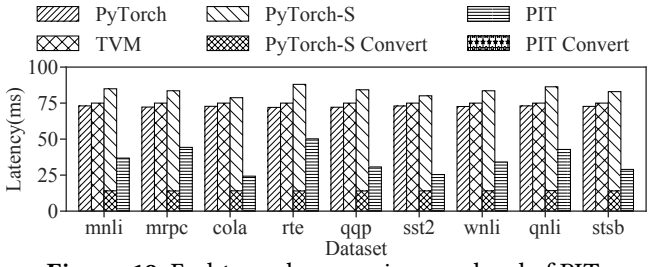


Figure 19. End-to-end conversion overhead of PIT.

16×16, and 13.3x~26.5x faster than Triton's index construction when the granularity is 32×32. As far as we know, PIT is the first to support the fast online index construction for all kinds of sparsity granularities.

To further evaluate the conversion overhead, we also measure the proportion of the conversion overhead in the end-to-end latency of PIT. Specifically, PIT optimizes the dynamic sparsity caused by different sequence lengths in the BERT model. In addition to PyTorch, we also introduce TVM, a popular dense tensor compiler. Each task in TVM is finetuned 2000 steps by Auto-Scheduler (Ansor [67]). Figure 19 shows the end-to-end latency of PIT and the baselines, including the index construction overhead of PIT ("PIT Convert") and PyTorch-S ("PyTorch-S Convert"). The conversion overhead of PIT accounts for 0.7% to 1.1% of the end-to-end latency, which is almost invisible in the figure.

5.5 Micro-Tile Online Searching

Different sparsity patterns and different sparsity ratios may lead to different optimal micro-tiles. PIT considers the efficiency of the computation kernel and the computation waste at the same time to find the best micro-tile configuration. Table 3 shows the searched micro-tiles of $4096 \times 4096 \times 4096$ matrix multiplication under different sparsity granularities and ratios. Take the first line of Table 3 as an example. The algorithm finds it most efficient to use a micro-tile of 16×1 to cover the granularity of 2×1 when the sparsity ratio is 95%. The micro-tile of 16×1 for 2×1 data leads to a sparsity ratio of 66.39% in PIT's computation. The micro-tile 16×1 is derived from the dense computation tile $16 \times 32 \times 128$ by applying PIT transformation on the second axis of the first input tensor. PIT balances the trade-off between the efficiency of the kernel and the computation waste on the fly. It takes

Sparsity Granularity	Origin Sparsity Ratio(%)	Micro Tile	Sparsity Ratio After Cover (%)	Origin Dense Kernel	Latency (ms)
(2,1)	95	(16, 1)	66.39	$[16, 32] \times [32, 128]$	8.04
(2,1)	99	(8, 1)	96.06	$[8, 32] \times [32, 128]$	2.34
(4,1)	95	(16, 1)	81.45	$[16, 32] \times [32, 128]$	4.29
(4,1)	99	(16, 1)	96.05	$[16, 32] \times [32, 128]$	1.37
(8,1)	95	(8, 1)	95	$[8, 32] \times [32, 128]$	2.34
(8,1)	99	(32, 1)	96.02	$[32, 64] \times [64, 32]$	0.90
(32, 1)	95	(32, 1)	95	$[32, 64] \times [64, 32]$	0.94
(32, 1)	99	(32, 1)	99	$[32, 64] \times [64, 32]$	0.39

Table 3. The micro tile online search results for different sparsity granularity and sparsity ratios.

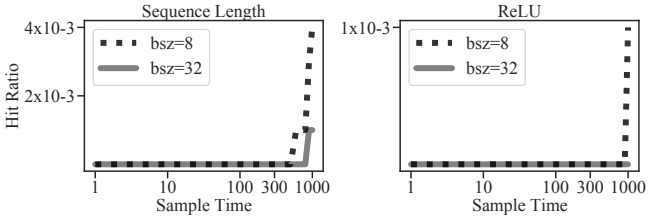


Figure 20. Investigation of the dynamics of sparsity patterns in varying sequence lengths and ReLU.

30us~100us for PIT to search for the best micro-tile and the corresponding dense computation tile, which is fast enough for online searching.

5.6 Dynamic Sparsity Pattern Study

A potential alternative solution for dynamic sparsity is to memorize frequent sparsity patterns, compile optimized kernels for them, and reuse these kernels when the pattern appears again. In this micro-benchmark, we invalidate this solution by investigating how frequently the dynamic sparse pattern will appear multiple times. We use two widely-existing dynamic sparsity patterns, i.e., the varying input sequence lengths and the dynamic sparsity caused by the ReLU operator. Specifically, we traverse the MNLI dataset with different batch sizes (8 and 32) and check whether the sparse pattern of the current input batch has appeared in the previous batches. Figure 20 shows the cumulative hit ratio of dynamic sparsity that has appeared in previous batches. We find the repetition ratios of the two sparsity patterns are both notably low. The varying sequence lengths only have 0.4% requests hitting a sparsity pattern that appeared in the previous batches. The ratio is even lower to 0.1% for ReLU. Given the dynamic nature of such sparsity, many previous research works for static sparsity optimization are no longer applicable to dynamic sparsity. The sparse kernel optimized for a specific sparsity pattern is almost non-reusable.

6 Related Works

Dynamic sparsity has emerged as a critical area for improving the efficiency of deep learning models. We discuss the

different categories of existing solutions, including the software solutions for both static sparsity patterns and dynamic sparsity patterns, as well as the hardware solutions. The comparison between PIT and these solutions can help to better understand the technical contribution of PIT.

Optimizations for Static Sparsity Patterns. Ahead-of-time compiler-based techniques like SparTA [68], TACO [7], SparseTIR [64], Tiramisu [12], and SparseRT [61] involve searching for an appropriate kernel configuration for a specific sparsity pattern. These techniques can achieve high performance for a given sparsity pattern but fail to handle dynamic sparsity patterns, which are only known at runtime. The key contribution of PIT is to support dynamic sparsity patterns on the fly through its low-overhead online sparsity detection and sparse-dense data transformation (i.e., SRead and SWrite). Moreover, for most static sparsity patterns, PIT can achieve performance similar to the ahead-of-time compilers, even though PIT can only detect the sparsity patterns at runtime, greatly saving the compiling overhead.

Optimizations for General Sparsity Patterns. Although general sparse libraries do not require ahead-of-time compilation, e.g., OpenAI's block sparse kernel [9], cuSPARSE [3], cuSPARSELT [5], HipSparse [8], Sputnik [31], nmSparse [15], these libraries only support or work effectively on specific data granularity and computation granularity, making them difficult to support more complex sparsity patterns. For example, OpenAI's block sparse kernel [9] only supports sparse data blocks of 32×32, leading to wasted computation when the sparsity pattern is more fine-grained (e.g., 1×32). PIT solves the problem by micro-tile. PIT supports the online construction of larger tiles by sparsely reading/writing multiple micro-tiles, which achieves both high GPU efficiency and low sparsity coverage cost. Although ASpT [36] proposes an adaptive tiling mechanism for fine-grained sparse matrix multiplication, it introduces significant offline data rearrangement overheads. But PIT can construct the tile with negligible overheads as long as a micro-tile can saturate the GPU memory transaction.

Hardware Optimizations for Sparsity Computation. In addition to the above software solutions, many hardware optimizations have been proposed for the sparse computation of deep learning models [2, 13, 17, 32, 62]. These hardware optimization techniques often target specific sparsity patterns. For example, NVIDIA's Sparse Tensor Core [2] in A100 GPUs only supports a strict 2-in-4 pattern, i.e., every 1×4 tile should have exactly 2 zeros, which greatly limits the applicability to more diverse sparsity patterns. PIT not only does not assume the sparsity pattern but also has the ability to augment the limited sparsity patterns of existing hardware solutions. For instance, when a matrix has mixed multiple 1x4 tiles with two sparsity patterns: two zeros and all zeros, PIT can construct micro-tiles to only feed the two-zero cases to the Sparse Tensor Core, avoiding the unnecessary computation of all-zero tiles. Such a GPU kernel can be implemented by combining PIT's SRead/SWrite with the "mma.sp" PTX instruction of Sparse Tensor Core, which we leave as future work.

Optimization for LLMs. Among recent advances to optimize the inference of large language models, vLLM [39] is the most relevant to PIT. vLLM proposed Paged Attention to "sparsely" load/save tokens from/to different physical addresses of GPU memory, breaking the continuous storage limitation of tokens. It saves excessive padding of varying sequence lengths and redundant copies during beam search. Paged Attention can be treated as a domain-specific solution for the special dynamic sparsity pattern in generative language models. PIT can be used to implement vLLM with a customized PIT transformation policy. By design, PIT is a general solution for dynamic sparsity that facilitates the support of varying sequence lengths and more challenging sparsity patterns (e.g., MoE, sparse attention, sparse training).

7 Conclusion

PIT takes a principled approach to support efficient execution of dynamically sparse models on commodity accelerators, based on *permutation invariant transformation*, a property commonly existing in deep learning computations. With this property, PIT constructs computation-efficient dense tiles from hardware-friendly micro-tiles in an online manner. PIT demonstrates a novel and effective way of handling dynamic sparsity, a growing trend in deep learning. The extensive evaluation shows PIT can accelerate dynamic sparsity computation in both inference and training by up to 5.9x over state-of-the-art solutions.

Acknowledgement

We thank our shepherd, Thomas Pasquier, for his valuable suggestion that improves the quality of our paper. We also thank the anonymous reviewers for their constructive feedback. Quanlu Zhang, Zhenhua Han, and Yuqing Yang are the corresponding authors.

References

[1] Longformer. https://github.com/allenai/longformer, 2020.
[2] Accelerating inference with sparsity using the nvidia ampere architecture and nvidia tensorrt. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/, 2021.
[3] The api reference guide for cusparse, the cuda sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html, 2021.
[4] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma, 2021.
[5] cusparselt: A high-performance cuda library for sparse matrix-matrix multiplication. https://docs.nvidia.com/cuda/cusparselt/index.html, 2021.
[6] Openai block sparse. https://github.com/openai/blocksparse.git, 2021.
[7] Reproducing oopsla 2020 results. https://github.com/tensor-compiler/taco/tree/oopsla2020, 2021.

[8] Rocm sparse marshalling library. https://github.com/ROCmSoftwarePlatform/hipSPARSE, 2021.

[9] Triton. https://github.com/openai/triton.git, 2021.

[10] Tvm sparsity code. https://github.com/apache/tvm/blob/254563a3140cf63fe77a46058688209de3aa213c/python/tvm/topi/cuda/sparse.py#L96, 2021.

[11] Deepspeed. https://github.com/microsoft/DeepSpeed, 2022.

[12] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for dense and sparse deep learning. *ArXiv preprint*, abs/2005.04091, 2020.

[13] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.

[14] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *ArXiv preprint*, abs/2004.05150, 2020.

[15] Lin Bin, Zheng Ningxin, Wang Yang, Cao Shijie, Ma Lingxiao, Zhang Quanlu, Zhu Yi, Cao Ting, Xue Jilong, Yang Yuqing, and Yang Fan. Efficient gpu kernels for n:m-sparse weights in deep learning. *MLSys2023*, 2023.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[17] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[18] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[19] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12270–12280, 2021.

[20] 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)*, pages 578–594, 2018.

[21] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, New Orleans, Louisiana, 2018. Association for Computational Linguistics.

[22] 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)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[24] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.

[25] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10):891–921, 1905.

[26] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 2943–2952. PMLR, 2020.

[27] Alexander Fabbri, Irene Li, Tianwei She, Suyi Li, and Dragomir Radev. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1074–1084, Florence, Italy, 2019. Association for Computational Linguistics.

[28] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[29] 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.

[30] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. *MLSys2023*, 2023.

[31] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[32] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[33] Amirhossein Habibian, Davide Abati, Taco S Cohen, and Babak Ehteshami Bejnordi. Skip-convolutions for efficient video processing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2695–2704, 2021.

[34] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[35] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

[36] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[37] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *ArXiv preprint*, abs/2206.03382, 2022.

[38] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning*

Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020.

[39] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Shen, Lianmin Zheng, Cody Yu, Joey Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP 23)*, 2023.

[40] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander Rush. Block pruning for faster transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10619–10629, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics.

[41] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3744–3753. PMLR, 2019.

[42] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[43] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5958–5968. PMLR, 2020.

[44] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. Large models are parsimonious learners: Activation sparsity in trained transformers. *ArXiv preprint*, abs/2210.06313, 2022.

[45] Liu Liu, Zheng Qu, Zhaodong Chen, Yufei Ding, and Yuan Xie. Transformer acceleration with dynamic sparse attention. *ArXiv preprint*, abs/2110.11299, 2021.

[46] Shiwei Liu and Zhangyang Wang. Ten lessons we have learned in the new "sparseland": A short handbook for sparse neural network researchers. *ArXiv*, abs/2302.02596, 2023.

[47] Yao Lu, Yue Dong, and Laurent Charlin. Multi-XScience: A large-scale dataset for extreme multi-document summarization of scientific articles. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8068–8074, Online, 2020. Association for Computational Linguistics.

[48] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 881–897, 2020.

[49] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, 2011. Association for Computational Linguistics.

[50] OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

[51] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *ArXiv preprint*, abs/2203.02155, 2022.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[53] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[54] Colin Raffel. Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching. 2016.

[55] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[56] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *ArXiv preprint*, abs/2204.06125, 2022.

[57] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *Advances in neural information processing systems*, 34:13937–13949, 2021.

[58] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[59] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[60] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[61] 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*, pages 31–42, 2020.

[62] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[63] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, 2020. Association for Computational Linguistics.

[64] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. *ArXiv preprint*, abs/2207.04606, 2022.

[65] Botao Yu, Peiling Lu, Rui Wang, Wei Hu, Xu Tan, Wei Ye, Shikun Zhang, Tao Qin, and Tie-Yan Liu. Museformer: Transformer with fine- and coarse-grained attention for music generation. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[66] 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*, abs/2205.01068, 2022.

[67] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[68] 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)*, pages 213–232, 2022.

[69] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115, 2021.

[70] Zhe Zhou, Junlin Liu, Zhenyu Gu, and Guangyu Sun. Energon: Towards efficient acceleration of transformers using dynamic sparse attention. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[71] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, 2022. USENIX Association.

[72] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. *ArXiv preprint*, abs/2202.08906, 2022.