DynaSpa: Exploiting Spatial Sparsity for Efficient Dynamic DNN Inference on Devices

Renyuan Liu
George Mason University
Fairfax, VA, USA
rliu23@gmu.edu

Yuyang Leng
George Mason University
Fairfax, VA, USA
yleng2@gmu.edu

Shilei Tian
Stony Brook University
Stony Brook, NY, USA
shilei.tian@stonybrook.edu

Shaohan Hu
Global Technology Applied Research, JPMorganChase
New York, NY, USA
shaohan.hu@jpmchase.com

Chun-Fu (Richard) Chen
Global Technology Applied Research, JPMorganChase
New York, NY, USA
richard.cf.chen@jpmchase.com

Shuochao Yao
George Mason University
Fairfax, VA, USA
shuochao@gmu.edu

Abstract

Recent advancements in exploring machine learning models' dynamic spatial sparsity have demonstrated great potential for superior efficiency and adaptability without compromising accuracy when compared to conventional static-and-dense DNNs. However, realizing theoretical inference acceleration under practical deployment environments is still faced with significant system challenges. Current vendor libraries and tensor compilers fall short due to their extra data copy operations or insufficient computation schemes, especially for DNN operators with dynamic spatial sparsity.

To bridge this gap, we propose DynaSpa, an automated kernel generation framework that enables efficient on-device inference for DNNs with dynamic spatial sparsity across diverse computing platforms. DynaSpa jointly optimizes computation and sparse patterns, while also leveraging the underlying hardware characteristics. DynaSpa consistently outperforms state-of-the-art vendor libraries and tensor compilers on embedded and mobile GPUs. For DNN operators with spatial sparsity ratio between 50% ~ 90%, DynaSpa achieves a speedup of ×1.3 ~ ×4.4 for Jetson AGX Orin GPU, ×1.6 ~ ×7.7 for Jetson AGX Xavier GPU, and ×1.5 ~ ×7.8 for Adreno mobile GPU, when compared to their respective dense counterparts.

CCS Concepts

• **Computing methodologies → Machine learning**; • **Software and its engineering → Dynamic compilers**.

Keywords

Mobile computing, Dynamic sparsity

ACM Reference Format:
Renyuan Liu, Yuyang Leng, Shilei Tian, Shaohan Hu, Chun-Fu (Richard) Chen, and Shuochao Yao. 2024. DynaSpa: Exploiting Spatial Sparsity for Efficient Dynamic DNN Inference on Devices. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '24), November 4–7, 2024, Hangzhou, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3666025.3699348

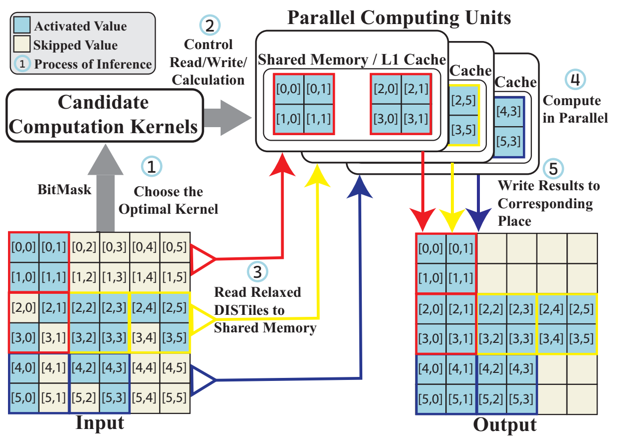
1 Introduction

Deep Neural Networks (DNNs) are crucial in applications like autonomous driving, augmented and mixed reality, and video analytics. With rising concerns about latency and privacy, there is a growing need to deploy task-specific networks on mobile, embedded, and edge devices [1–3]. Researchers have extensively studied ways to accelerate inference while maintaining output quality [1–10]. Modifying DNN models to suit resource-limited devices has proven effective [4, 8, 10–14].

Dynamic neural networks [15–23] achieve inference acceleration by adjusting the model based on input, reducing computational requirements, especially in the number of arithmetic operations. Leveraging spatial sparsity has become a popular dynamic technique used in various model architectures, ranging from convolutional neural networks [15–17] to transformer models [18–20]. These networks focus computation on selected data areas, ignoring the rest. This approach has led to significant theoretical computation reduction in tasks like classification, detection [15–21], and generative AI workloads like image generation [22, 23], making it a promising solution for on-device inference.

Unfortunately, it remains challenging to achieve practical inference speedups on mobile and embedded GPUs because merely reducing the arithmetic operations in DNN models does not inherently assure a practical acceleration. Dynamic neural networks introduce a huge amount of sparse tensor computation, which becomes a new performance bottleneck. Existing sparse tensor libraries, such as cuSparse [24], Sputnik [25], and Intel MKL [26], outperform their dense counterparts only when the sparsity ratio exceeds 90% [24, 25]. However, dynamic DNNs typically have moderate sparsity levels (50%-90%), which are insufficient for these sparse kernels to be more efficient than dense ones. A particularly time-consuming step in these libraries is the copying of non-zero values to the global memory before computation, a process that significantly hampers efficiency.

To overcome the aforementioned limitations of the existing methods, we propose DynaSpa, a novel sparse tensor computation framework designed to enable the practical acceleration of tensor programs with spatial sparsity on resource-limited devices.



Activated Value
Skipped Value
Process of Inference
Control Read/Write/ Calculation
Parallel Computing Units
Shared Memory / L1 Cache
Cache
Cache
Candidate Computation Kernels
Compute in Parallel
BitMask
Choose the Optimal Kernel
Read Relaxed DISTiles to Shared Memory
Write Results to Corresponding Place
Input
Output

Figure 1: DynaSpa inference example.

DynaSpa operates by first conducting an offline search to identify a set of computation kernel candidates suited for various sparse inputs. Then, during online inference, DynaSpa employs a bitmask representing the input interest area to select the most appropriate kernel from these candidates. Activated input data is loaded into the shared memory of parallel execution units using an efficient tiling method and processed with the most effective schedule defined by the kernel. Figure 1 illustrates DynaSpa's inference process. To realize this process and address all the previously mentioned challenges, we propose three composability designs.

Relaxed Sparsity Composition. To ensure optimal load balance and effective data reuse in the computation kernel, we propose an abstraction named Dense-In-Sparse Tile (DISTile). This approach relaxes the constraint of eliminating all unnecessary computation. The activated parts of the input data are composed into multiple DISTiles, which are then evenly distributed among all GPU computational units for parallel processing.

Polyalgorithm Kernel Composition. To support dynamic sparse DNN operators, DynaSpa leverages data-driven auto-scheduler techniques [27, 28] to search for several kernel candidates offline, collectively termed as polyalgorithm kernel (PolyKernel) , to cover all the possible sparsity patterns. When given an operator with a sparsity mask during runtime, DynaSpa uses a bitmask and bitwise operations to select and dispatch the most suitable kernel, ensuring both accurate computation and minimized execution time.

Analytical Cost Model Composition. While the previous composability designs enable tensor compilers to generate efficient kernels for DNN operators with spatial sparsity, they also enlarge the search space. To address this, DynaSpa enhances the auto-scheduler's data-driven cost model by integrating an analytical hardware model designed to estimate the upper performance bound of tensor operations. This model streamlines the search space by filtering out suboptimal kernel implementations.

We evaluated DynaSpa on a wide range of tensor workloads with spatial sparsity 50% ~ 90% on embedded (Jetson AGX Orin and Jetson AGX Xavier) and mobile GPUs (Adreno 650). Compared to manually optimized dense tensor libraries (i.e., cuDNN [29] and TFLite [30]) and tensor compilers (i.e., AutoTVM [27] and PIT [31]), DynaSpa achieves ×1.3 ~ ×4.4 speedup for Jetson AGX Orin GPU, ×1.6 ~ ×7.7 speedup for Jetson AGX Xavier GPU, and ×1.5 ~ ×7.8 speedup for Adreno mobile GPU. Compared to the sparse tensor library, cuSparse, DynaSpa can achieve up to ×32 speedup. There is no loss in accuracy compared to dynamic neural networks.

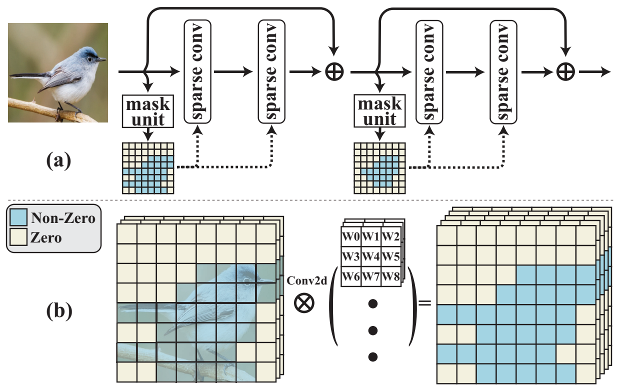


Figure 2: A dynamic DNN example. (a) Residual structure w/ spatial mask unit [16] (b) Illustration of sparse convolution.

In summary, our paper makes the following contributions:

- It presents DynaSpa, a framework to generate compute kernels for high-performing spatially-sparse DNNs.
- It proposes a PolyKernel auto-scheduling framework with runtime dispatch to support dynamic sparse workloads.
- It introduces an analytical performance model for search space reduction.
- It presents the complete implementation and comprehensive evaluation of DynaSpa, demonstrating its superior performance over state-of-the-art systems with a variety of spatially-sparse DNNs on mobile/embedded GPUs.

2 Background & Motivation

In this section, we discuss the technical background of the problem at hand and motivate our proposed solution.

2.1 Spatial Sparsity in DNNs

Spatial sparsity, also known as activation sparsity, is crucial for enhancing DNN efficiency [15–23]. Unlike weight sparsity, common in model pruning [11, 12, 32], which is fixed post-training, spatial sparsity is dynamic and input-dependent. Weight sparsity allows modification of filters and can utilize existing libraries for speedup [12, 32]. In contrast, spatial sparsity requires system support for acceleration due to its dependency on input data semantics. Hence, this paper focuses on spatial sparsity.

There are primarily two sources of spatial sparsity in tensor operations: *i*) the spatial sparsity inherited from the tensor input, including sparse activation maps with ReLU function or from input data [15, 33, 34] and gradual interactive editing in image generation [22, 23]; and *ii*) the generated sparse output activation map with a trainable tiny component [16–20]. Fortunately, we can formulate both with a single format-agnostic computation description. Without loss of generality, we take the convolution operation as an example: $Y_{h,w,o} = M_{h,w} \odot X_{p,q,c} \otimes K_{r,s,c,o}$, where $\odot$ and $\otimes$ represent the element-wise product and convolution respectively; X, Y, K, M denote input, output, filters, and the spatial mask respectively, while the subscripts indicate the tensor shape. Every element within the spatial mask M assumes a binary value of either 0 or 1. If M solely comprises 1s, it defaults to a standard dense convolution operation. Figure 2 (a) shows a dynamic neural network with spatial

Table 1: Comparison of different algorithms.

Algorithm	Aware of Sparse Pattern	No Extra Data Copy	Jointly Optimize Computation and Sparse Patterns	Runtime Overhead
cuDNN	×	✓	×	Very Low
cuSparse	✓	×	✓	Very Low
PIT	✓	✓	×	High
DynaSpa	✓	✓	✓	Low

sparsity from DynConv [16]. Figure 2 (b) illustrates the corresponding sparse convolution. Spatially sparse computations for other operations (e.g., attention) can also be defined in a similar fashion.

2.2 Practical Gaps for Spatial Sparsity

Existing studies on spatially-sparse model designs emphasize creating trainable models with spatially-sparse operations. However, they often neglect practical deployment considerations. Many implementations introduce element-wise products with sparsity masks subsequent to performing the original full-dense operations. Therefore, no practical speedup is achieved, as illustrated in Figure 3, where we execute all operations from a spatially-sparse DNN [16] with existing sparse tensor libraries on embedded/mobile GPUs.

Gather-and-Scatter. The Gather-and-Scatter operation is currently the predominant implementation for spatially-sparse DNNs [15, 16, 22]. Its core concept is both simple and intuitive. Based on the spatial mask, the `gather` function extracts each input tensor patch and concatenates them into a batched tensor. Next, it carries out the original operation (e.g., `Conv2d` and `MultiheadAttention`). Finally, the `scatter` function, which acts as the inverse of the gather operation, places the computed result back to its respective index in the output tensor. While the Gather-and-Scatter approach is easy to implement, it often leads to undesired memory access patterns, poor cache locality, and long kernel launch time, especially when the three steps are not jointly optimized manually or automatically.

Vendor Library & Tensor Compiler. The vendor library for NVIDIA GPUs, i.e., cuSparse [24], can support spatial sparsity but with a particular focus on high-degree sparsity. Furthermore, the majority of sparsity optimizations from vendor libraries (e.g., cuSparse [24] and TFLite [30]) focus on static weight sparsity, which is hardly applicable to spatially-sparse operators. Sparse tensor compilers, such as TACO [35, 36], decouple sparse format specification and computation descriptions. SparseTIR [37] advances this concept by supporting multiple composable formats. SparTA [38] proposes sparse annotations tailored for network pruning and quantization. However, these works are geared towards static sparsity, making them suitable for weight sparsity, but not directly applicable to activation/spatial sparsity. Activation/spatial sparsity patterns change with different inputs, so methods designed for static sparsity can generate efficient kernels for a specific sparse pattern but incur high overhead for dynamic sparsity. These methods must copy and store non-zero data in a dense format (e.g., compressed sparse row format) [24, 35, 36] and generate kernels based on this format. This step must be repeated before each computation for dynamic sparsity, leading to significant overhead.

2.3 Related Works

Recent progress in tensor compilers has been made in supporting dynamic dense neural networks. Most efforts focus on graph-level and control-flow optimizations, including Nimble [39], DISC [40], Cortex [41], and Cocktailer [42]. However, these approaches are orthogonal to our challenges when addressing dynamic spatial sparsity. DietCode [43] proposes an auto-scheduler to handle dynamic input shapes. However, it is still tailored for dense workloads and cannot deal with the dynamic spatial sparsity. Another line of work focuses on reducing the compilation time of tensor compilers. Romou [7] identifies performance bottlenecks in mobile GPUs, which we have also absorbed into our schedules. Roller [44] introduces a constructive approach to generate kernels. Instead of using a data-driven auto-scheduler, Roller relies solely on a direct search to identify kernel implementation parameters optimized for maximum throughput. PIT [31], a just-in-time (JIT) compiler, supports dynamic sparsity only for matrix multiplication due to the restriction of permutation-invariant operations. As a result, we must use explicit GEMM-based algorithms for convolutions, which are typically 1.5 times slower than other convolution algorithms [45]. Additionally, as a JIT compiler, PIT naturally incurs extra runtime overhead and cannot jointly optimize computation schemes and sparse patterns, easily leading to resource wastage. Furthermore, each layer's sparsity often varies for different inputs, increasing PIT's runtime overhead (40 μs to 500 μs) [31]. Our approach, using bitmasks, significantly reduces index generation and kernel selection time (less than 10 μs), as discussed in Section 3.4. Table 1 shows the differences between DynaSpa and other algorithms.

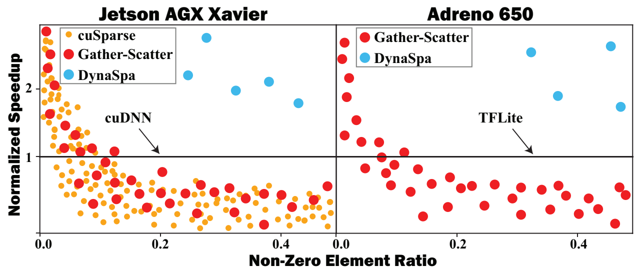


Figure 3: No practical speedup for moderated (50%-99%) sparsity with sparse tensor libraries.

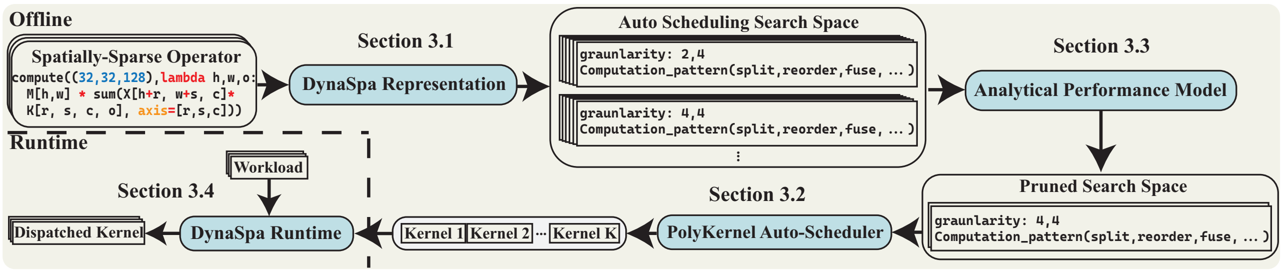
3 DynaSpa Design

This section presents DynaSpa's design details, an overview of which is illustrated in Figure 4. DynaSpa divides data into DISTiles and only reads the non-zero tiles into the L1/Shared Memory, thereby skipping the reading and computing other zero regions (Section 3.1). Using the PolyKernel Auto-Scheduler, DynaSpa can offline select candidates from various possible DISTile sizes and computation methods (Section 3.2). To reduce the search space of the PolyKernel Auto-Scheduler, DynaSpa uses an analytical performance model to eliminate poorly performing search points (Section 3.3). At runtime, DynaSpa can select the optimal computation kernel from the candidates with low overhead (Section 3.4).

3.1 DynaSpa Representation

First we introduce the DynaSpa representation, a new abstraction for spatially-sparse tensor operators that supports relaxed sparsity composition.

Dense-In-Sparse Tile (DISTile). One challenge in optimizing efficient computing kernels for spatially-sparse operators is the poor memory access pattern and limited data reuse, where a trade-off exists between memory efficiency and computation efficiency.



```
compute((32,32,128),lambda h,w,o:
  M[h,w] * sum(X[h+r, w+s, c]*
  K[r, s, c, o], axis=[r,s,c]))
```

```
graunlarity: 2,4
Computation_pattern(split,reorder,fuse, … )
```

```
graunlarity: 4,4
Computation_pattern(split,reorder,fuse, … )
```

```
graunlarity: 4,4
Computation_pattern(split,reorder,fuse, … )
```

Figure 4: DynaSpa overview.

Spatial sparsity is usually globally sparse but locally dense. DynaSpa employs relaxed sparsity composition, admitting extra unneeded computations (involving zero-valued multiplications) to achieve a more regular block-sparse structure.

Therefore, we propose a new tile abstraction called Dense-In-Sparse Tile (DISTile) that composes spatially-sparse tensor computation into a virtually dense nested loop. Traditional computation kernels (i.e., cuDNN [29] and TFLite [30]) also divide data into tiles and assign the tiles to computation units. However, DISTile focuses only on the tiling of non-zero regions, avoiding the need to read and compute zero regions, thereby conserving resources for edge GPUs. As shown in Figure 5, the DISTile encapsulates a multi-dimensional sparsity `granularity`, defined along each loop axis of a given sparsity mask and tensor expression `expr`. Given `granularity` and sparsity mask, `Generate_DISTiles` can infer DISTiles and their top-left index `ind` for addressing. In addition, to simplify the illustration, in Figure 5, `Generate_DISTiles` is applied to the whole sparsity mask, and thus the DISTiles extracted are horizontally and vertically aligned. However, DynaSpa provides an option to apply `Generate_DISTiles` independently to the individual connected components of the sparsity mask. This leads to unaligned DISTiles across different connected components. While this approach offers flexibility, it introduces additional runtime overhead due to the parallel connected-component labeling process [46] in DISTile generation.

DISTile operates on the output mask, and DynaSpa reads the corresponding input and weight data directly from DRAM/Global Memory into the L1 cache/Shared Memory based on the DISTile index. In contrast, cuSparse [24] and Gather-Scatter methods [15, 16, 22] typically require rearranging sparse data into a dense format in DRAM/Global Memory explicitly before computation. DynaSpa eliminates the copy overhead by directly accessing the necessary data.

Phantom Read/Write. In `Generate_DISTiles`, the shape of the sparsity mask usually cannot be evenly divided by `granularity`. DynaSpa is designed to support any valid `granularity` to generate the best-performing kernel during auto-scheduling. However, improperly handling non-divisible `granularity` value can adversely impact performance. For example, as shown in Figure 5, there is a workload with a sparsity mask of shape $[5 \times 5]$ and `granularity` $[2 \times 2]$. The generated `DISTile3` includes two sub-workloads, located at $(2, 5)$ and $(3, 5)$, which do not exist in the original computation. We call these sub-workloads *phantom*.

In DISTile, we introduce phantom read/write to address two key questions: should these phantom workloads be computed, and how should the read and write operations be carried out? Regarding the first question, although phantom computations are redundant, they help eliminate branch instructions and prevent thread divergence, which can serialize GPU warps and increase latency [47]. Regarding the second question, to avoid unnecessary global memory operations, DISTile does not read data from global memory for phantom workloads. Instead, it checks boundaries and assigns 0 to the cache. During write-back, it rechecks boundaries and stops write-back for phantom workloads, ensuring tensor program integrity. The boundary checks have a negligible impact on latency, as long-latency global memory operations hide the latency induced by phantom read/write.

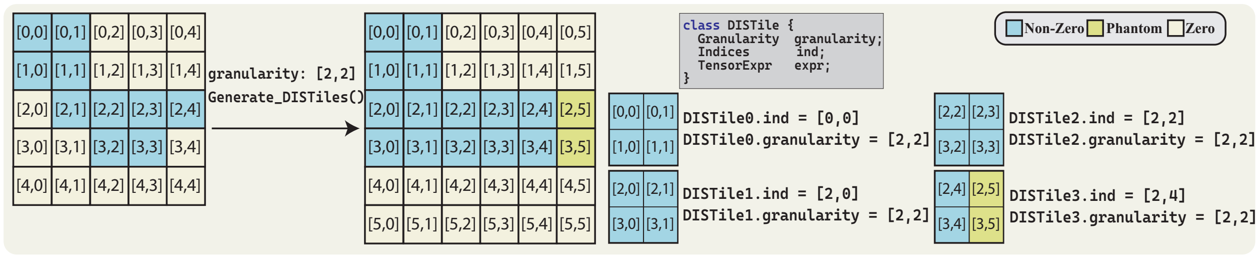
3.2 PolyKernel Auto-Scheduler

The DynaSpa representation described in Section 3.1 can have arbitrary sizes. Each DISTile can also be assigned to compute units in different ways, such as varying the computation order within each DISTile. To find the optimal computation scheme, we introduce the PolyKernel auto-scheduler. It generates a set of candidate computation kernels offline within a limited number of searches.

Polyalgorithm Formulation for Dynamic Sparsity. Given specific hardware, optimal kernel implementations can vary greatly depending on the sparsity ratio and patterns. Therefore, finding a single implementation that efficiently handles all forms of spatial sparsity is nearly impossible. In DynaSpa, we adopt the conventional polyalgorithm approach used in the vendor library. We offer multiple kernel implementations and select the most suitable one during runtime. Therefore, the problem naturally can be partitioned into two parts. Firstly, how to automatically search a set of kernel implementations that can handle all possible cases while achieving the best overall speedup. Secondly, how to dispatch the best kernel implementation at runtime with minimal overhead. We will now focus on the first challenge and explore the second in Section 3.4.

Without loss of generality, let τ represent a code template paired with a set of tuning knobs θ, representing potential schedules (e.g., split, fuse, reorder). The space of all possible schedules is denoted by Θ. We represent the corresponding low-level code as $\tau(\theta)$. Our primary objective is to minimize the actual running cost (e.g., execution time) denoted by $f(\cdot)$ on specific hardware. The analytical expression of $f(\cdot)$ is unknown in advance, but we can make queries by executing experiments on hardware and taking measurements as feedback. Instead of finding an optimal schedule θ^* that minimizes the running cost, the polyalgorithm approach searches for an optimal set of schedules with the following objective:

$$\underset{\mathcal{S}_\theta \subset \Theta}{\arg\min}\, \mathbb{E}_{m \sim \mathcal{M}} \left[\min_{\theta \in \mathcal{S}_\theta} f\big(\tau(\theta), m\big) \right] \quad \text{s.t.} \quad \|\mathcal{S}_\theta\| \leq k, \tag{1}$$



```
class DISTile {
  Granularity  granularity;
  Indices       ind;
  TensorExpr   expr;
}
```

```
DISTile0.ind = [0,0]
DISTile0.granularity = [2,2]
DISTile1.ind = [2,0]
DISTile1.granularity = [2,2]
DISTile2.ind = [2,2]
DISTile2.granularity = [2,2]
DISTile3.ind = [2,4]
DISTile3.granularity = [2,2]
```

Figure 5: DynaSpa representation: DISTile generation w/ phantom read/write

where $\mathcal{S}_\theta \subset \Theta$ is the set of optimal schedules whose cardinality is less than k, and m denotes the input sparsity mask.

Spatial sparsity is a type of structured sparsity that differs from random binary matrices. While the analytical probabilistic density function for the sparsity masks m remains unknown, it can be implicitly determined by sampling sparsity masks that follow the underlying unknown distribution. This is achieved by feeding real-world dataset inputs into a spatially-sparse neural network and collecting sparsity masks from each operator individually as a separate sample set $\mathcal{M}$. The objective function (1) implies that our PolyKernel auto-scheduler wants to find, at most, k kernel implementations for a spatially-sparse operator with the best-expected execution time over all feasible spatial sparsity.

Supermodular Structure in Polyalgorithm. Compared to the original auto-scheduler problem, our PolyKernel auto-scheduler adds complexity by introducing an additional combination optimization on top of the original auto-scheduling problem. Fortunately, we can exploit the structure within the polyalgorithm formulation, which allows us to design an approximate algorithm with performance bound. According to the PolyKernel objective (1), we define a set cost function $g(\mathcal{S})$ as

$$g(\mathcal{S}) = \mathbb{E}_{m\sim\mathcal{M}}\left[\min_{\theta\in\mathcal{S}} f\big(\tau(\theta), m\big)\right], \tag{2}$$

where $\mathcal{S} \subset \Theta$ is a feasible set of schedules.

Therefore, we can prove that the set function $g(\mathcal{S})$ is supermodular by verifying the following inequality. For every feasible schedule set pair $\{\mathcal{S}_1, \mathcal{S}_2\}$, where $\mathcal{S}_1 \subset \mathcal{S}_2 \subset \Theta$, and every single schedule that satisfy $s \in \Theta \setminus \mathcal{S}_2$, we have that

$$g(\mathcal{S}_1) - g(\mathcal{S}_1 \cup \{s\}) \geq g(\mathcal{S}_2) - g(\mathcal{S}_2 \cup \{s\}) \tag{3}$$

regardless of the chosen sparsity mask distribution. The inequality (3) proves that $-g(\mathcal{S})$ is a submodular set function by definition, leading to the conclusion that $g(\mathcal{S})$ is supermodular. Therefore, we can reformulate the original PolyKernel optimization problem (1) as:

$$\operatorname*{arg\,min}_{\mathcal{S}_\theta\subset\Theta} g(\mathcal{S}_\theta) \quad \text{s.t. } \|\mathcal{S}_\theta\| \leq k, \tag{4}$$

which is a supermodular minimization problem or, in equivalent terms, a submodular maximization problem with a cardinality constraint. Although the generic submodular maximization problem is NP-hard even in the unconstrained setting [48], we can find a greedy $1 - 1/e$ approximation algorithm for monotone submodular function subject to a cardinality constraint [49]. The computational complexity of PolyKernel scales linearly with the cardinality of PolyKernel.

Integration with Existing Search Algorithms. An important piece of auto-scheduler that we have not discussed in detail before is the search algorithm. While there are plenty of recent works on developing better search algorithms for auto-scheduler [50–52], they generally follow a standard iterative process:

(1) Begin by initializing the cost function, $f(\cdot)$, using a specific machine learning model.
(2) Using the current cost model $f(\cdot)$ as the energy function, `Explore-and-Exploit` the search space and choose a representative set of schedules Q with their executable.
(3) Run the chosen schedules on the targeted hardware. Measure the kernel execution times and utilize these data as feedback to update cost model $f(\cdot)$.
(4) Repeat step (2) until the cost model converges.

Various search algorithms primarily differ in how they design their `Explore-and-Exploit` functions [50–52].

Our PolyKernel search algorithm is shown in Algorithm 1. To integrate PolyKernel into the existing iterative process, we face two major challenges. Firstly, our optimization objective is based on an expectation over dynamic spatial sparsity instead of a static workload. To approximate the expectation within our set cost function $g(\mathcal{S})$, we use a mini-batch average for stochastic optimization:

$$g(\mathcal{S}) \approx g(\mathcal{S}, \mathcal{M}') = \frac{1}{\|\mathcal{M}'\|} \sum_{m\in\mathcal{M}'} \min_{\theta\in\mathcal{S}} f\big(\tau(\theta), m\big) \tag{5}$$

where $\mathcal{M}'$ is a mini-batch sampled randomly from the complete sparsity mask dataset $\mathcal{M}$. Mini-batch stochastic optimization has been widely adopted in various machine learning model training. As shown in Line 5 of Algorithm 1, we sample a mini-batch sparsity mask for every iteration.

Secondly, we need to embed the greedy supermodular minimization algorithm within the search procedure. The greedy approach provides a simple solution with a nice approximation guarantee for the supermodular minimization problem defined in (4). The solution starts with an empty set $\mathcal{S}_0$ and then repeats the following step for $i \in [1, k]$:

$$\mathcal{S}_i = \mathcal{S}_{i-1} \cup \left\{ \operatorname*{arg\,min}_{s\in\Theta\setminus\mathcal{S}_{i-1}} \Delta_g(s|\mathcal{S}_{i-1}, \mathcal{M}) \right\} \tag{6}$$

$$\Delta_g(s|\mathcal{S}, \mathcal{M}) = g(\mathcal{S} \cup \{s\}, \mathcal{M}) - g(\mathcal{S}, \mathcal{M}). \tag{7}$$

The algorithm enlarges the solution set using a greedy approach that selects the element offering the best marginal gain, as defined in (7). Within Algorithm 1, we integrate concepts of greedy supermodularity in Lines 7 to 10 and 16 to 20. While the original `Explore-and-Exploit` focuses solely on identifying the representation set for a single optimal implementation, our objective is to adapt this process for PolyKernel. Therefore, as shown in Lines 7 to 10, we introduce a for loop, adopting the marginal gain cost function instead of the original one, emulating the greedy approach for supermodular optimization. In Lines 16 to 20, we conduct the greedy

Algorithm 1: PolyKernel Auto-Scheduling

```
1  Input: Transformation space $\Theta$, and sparsity mask set $\mathcal{M}$ ;
2  Output: Selected schedule configuration $\mathcal{S}_\theta = \{\theta_i\}_{i=1}^{k}$ ;
3  $\mathcal{D} \leftarrow \emptyset$ ;
4  while $n\_trials < max\_n\_trial$ do
5      Sample a mini-batch sparsity masks $\mathcal{M}'$ from $\mathcal{M}$ ;
6      $\mathcal{Q} \leftarrow \emptyset$ ;
7      for $i \in [0, k)$ do
8          Explore-and-Exploit the representative schedules $Q_i$ using
             $\Delta_g(\cdot|\mathcal{Q}, \mathcal{M}')$ defined in equation (7) as the energy function ;
9          $\mathcal{Q} \leftarrow \mathcal{Q} \cup Q_i$;
10     end
11     for $q \in \mathcal{Q}$ do
12         $\mathcal{D} \leftarrow \mathcal{D} \cup \{[q, \mathrm{Perf}(q)]\}$ ; /* Run measurement on hardware
             environment */
13     end
14     Update the cost function $f(\cdot)$ using $\mathcal{D}$
15 end
16 $\mathcal{S}_\theta \leftarrow \emptyset$ ;
17 for $i \in [0, k)$ do
18     $d_i = \arg\min_{d \in \mathcal{D} \setminus \mathcal{S}_\theta} g(\mathcal{S}_\theta \cup \{d[0]\}) - g(\mathcal{S}_\theta)$;
19     $\mathcal{S}_\theta \leftarrow \mathcal{S}_\theta \cup \{d_i[0]\}$ ;
20 end
```

supermodularity algorithm again to decide the final PolyKernel output. One thing to notice is that the majority of time consumed by the search algorithm is dedicated to hardware profiling (Lines 11 to 13) [51, 52]. Our PolyKernel algorithm increases the profiling set by a factor of k. Therefore, the time complexity of the PolyKernel search is proportional to the number of PolyKernel implementations.

3.3 Analytical Performance Model

Based on Section 3.1 and 3.2, we are able to generate computation kernels for operators with dynamic sparsity. However, this comes with the trade-off of an expanded search space (i.e., DISTile granularity and PolyKernel). In this section, we will introduce our architecture-aware analytical cost model for search space pruning. Constructing an accurate cost model poses significant challenges. Therefore, we extend memory hierarchy models [53], aiming to estimate the *upper-bound performance* of tensor operations for given GPU architecture and kernel implementation parameters. Specifically, we decompose total latency into data access latency from memory hierarchy and computation latency, checking the performance limiting factors individually. We program the Jetson GPU code using CUDA [54] and the Mobile GPU using OpenCL [55]. Due to the varying software and hardware terminologies employed by different GPU architectures and programming languages, as illustrated in Table 2, we primarily adopt a CUDA-style description for consistency.

Global Memory Latency. Ensuring "coalesced" access patterns is crucial for achieving the best performance in global memory access. When threads in a warp access consecutive memory addresses, the GPU can efficiently combine these accesses into a single transaction, boosting bandwidth utilization and enhancing performance. For example, in Figure 6a, a GPU thread block needs to fetch a 4×8 tile from a 16×16 matrix. This matrix is stored in the GPU's global memory in a row-major format. Given that the global memory transaction size, G_{trans}, is set to 32, one might initially expect that $(4 \times 8)/32 = 1$ global memory transaction would be

Table 2: GPU terminology comparison.

Embedded Jetson/CUDA	Mobile/OpenCL
Streaming Multiprocessor (SM)	Shader Core
Shared Memory	L1 Cache/Local Memory
Thread Block	Work Group
Thread	Work Item

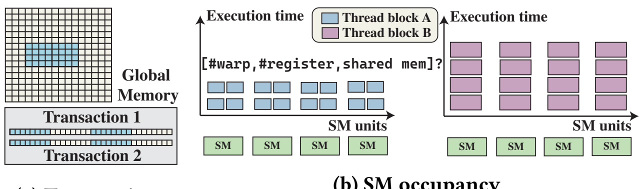


Figure 6: (a) Memory coalescing and calculating # global memory transaction; (b) GPU SM occupancy calculation according to #warp, #register, shared memory consumed by thread blocks and provided by SMs.

sufficient to fetch the tile. However, in reality, we require 2 global memory transactions. This is because the 4×8 tile data is not stored contiguously within the global memory. As a result, the available memory bandwidth is not fully exploited in this scenario.

Therefore, to estimate the global memory latency, we compute the number of global memory transactions of a thread block, denoted as N_{GMT}. The pattern of access to global memory is defined by a specific schedule template from which DynaSpa can directly determine N_{GMT} analytically. It is important to note that the template contains one or more virtual loops. These loops are maintained in the DynaSpa representation. We assume no memory continuity across these virtual loops. The global memory latency T_{global} is thus defined as

$$T_{global} = S_{type} \cdot G_{trans} \cdot N_{GMT} \cdot N_{blk} / B_{global}, \tag{8}$$

where S_{type} is the size data type of input element in bytes (e.g., 4 for a `float` value), B_{global} is the global memory bandwidth of the GPU in bytes per second, and N_{blk} is the number of thread blocks. In addition, B_{global} is provided by device specification or can be measured with profiling tools [7, 56].

Shared Memory Latency. Shared memory is dedicated to each execution unit in a GPU, such as the streaming multiprocessor (SM) in a Jetson GPU. Consequently, shared memory latency is closely tied to how thread blocks are allocated to SMs. As shown in Figure 6b, not every thread block can be processed by SMs simultaneously within a single cycle. Thus, to estimate latency, we break down the problem into two parts: 1) determining the number of cycles required to complete all thread blocks and 2) evaluating the shared memory latency within a single cycle.

To address the first question, several factors play a role in SM occupancy [57]. However, the fundamental approach to determine the number of thread blocks that can run concurrently on a single SM is by examining the ratios between the available capacity of each dedicated resource and the resource requirements of a thread block. Here, we focus on four resource types: shared memory, warps/threads, registers, and thread blocks.

$$\begin{aligned} K &= \min\left\{\left\lfloor\frac{SM_{thrd}}{N_{thrd}}\right\rfloor, \left\lfloor\frac{SM_{shared}}{S_{shared}}\right\rfloor, \left\lfloor\frac{SM_{reg}}{N_{reg}}\right\rfloor, SM_{blk}\right\} \\ C &= \lceil N_{blk}/K \rceil, \end{aligned} \tag{9}$$

```
1  __global__ void op_latency_kernel (float* result) {
2      int k;
3      float j = 1.;
4      float m = 3.;
5      float n = 5.;
6      j = j + 1.*threadIdx.x;
7      m = m + 1.*threadIdx.x;
8      for (k = 0; k <= 1000; k++) {
9          repeat512(n += m*j; m += n*j; j += n*m;); }
10     result[threadIdx.x] = n; }
```

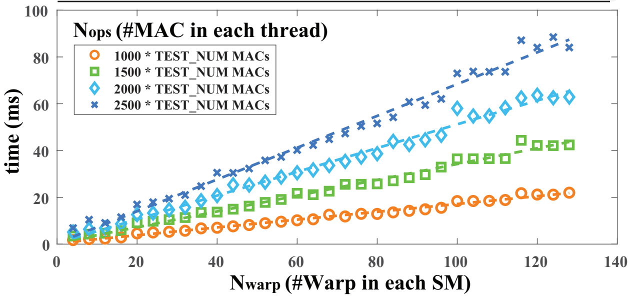


Figure 7: Microbenchmark: $time = \alpha \cdot N_{ops} \cdot N_{warp} + \gamma$.

where K and C denote the number of thread block that can run concurrently on a single SM and the number of cycles required to complete all thread blocks, respectively. SM_{thrd}, SM_{shared}, SM_{reg}, and SM_{blk} denote the capacities of threads, shared memory, registers, and thread blocks that a single SM can support. N_{thrd}, S_{shared}, and N_{reg} denote the number of threads, shared memory size, and number of registers requested by a single thread block. Therefore, we calculate the shared memory latency T_{shared} as:

$$T_{shared} = C \cdot K \cdot S_{shared} \cdot \beta_{bc} / B_{shared}, \tag{10}$$

where B_{shared} is the shared memory bandwidth of the GPU and β_{bc} is the coefficient indicating the degree of shared memory bank conflicts. The access pattern for shared memory is determined statically, allowing us to calculate the number of conflicts prior to kernel generation. For every shared memory load operation, DynaSpa examines the addresses accessed by a warp. It then determines if bank conflicts arise and computes the expected number of cycles for each shared-memory load [56]. If the expected number of bank conflicts is 0, we set β_{bc} to be 1.

Computation Latency. Similar to shared memory latency, computation latency is tied to how thread blocks are allocated among SMs. We can thus reuse the result (9) and focus on the computation latency for a single thread-block cycle. However, the number of warps executed at the same time on a GPU is typically unknown and can be affected by latency hiding due to warp switching [58]. To better understand this, we implemented a micro-benchmark, which is a one-time offline profiling that aims to determine the computation latency as we increase the number of warps and operations carried out in each thread, with the operations being multiply-accumulate (MAC) in our case.

As shown in Figure 7, the relationship among computation latency, the number of warps, and the number of operations in each thread can be approximated by a linear function. In addition, the micro-benchmark, which continually performs the same operation using identical data input, can inadvertently be "optimized" via common subexpression elimination by the NVIDIA nvcc compiler. To address this, we have been careful to avoid compiler-recognizable patterns, as illustrated in Line 9 in Figure 7. Therefore, we calculate the computation latency $T_{compute}$ as:

$$T_{compute} = C \cdot (\alpha \cdot N_{ops} \cdot N_{warp} + \gamma), \tag{11}$$

where N_{warp} denotes the number of warps consumed in a single SM, and N_{ops} denotes the number of MAC operations executed by each thread in a warp. α and γ are linear coefficients learned from the microbenchmark.

Upper-bound Performance Model. One significant benefit of multithreading in GPUs is latency hiding, allowing for the concurrent execution of memory access and computation [58]. While determining the exact degree of overlap due to latency hiding in practice is challenging, DynaSpa focuses solely on establishing an upper-bound performance model used to prune the search space guaranteed to have poor performance. Therefore, we take the assumption that all the latency components can perfectly hide the others, with the total latency being the worst among them.

$$T_{total} = \max \left\{ T_{global}, T_{shared}, T_{compute} \right\}, \tag{12}$$

DynaSpa computes T_{total} for possible combinations in the original search space, which takes at most 6 min for all operators we have encountered. Then, it picks the top 0.1% of the best-performing combinations. This set forms the pruned search space and is fed into our PolyKernel auto-scheduler.

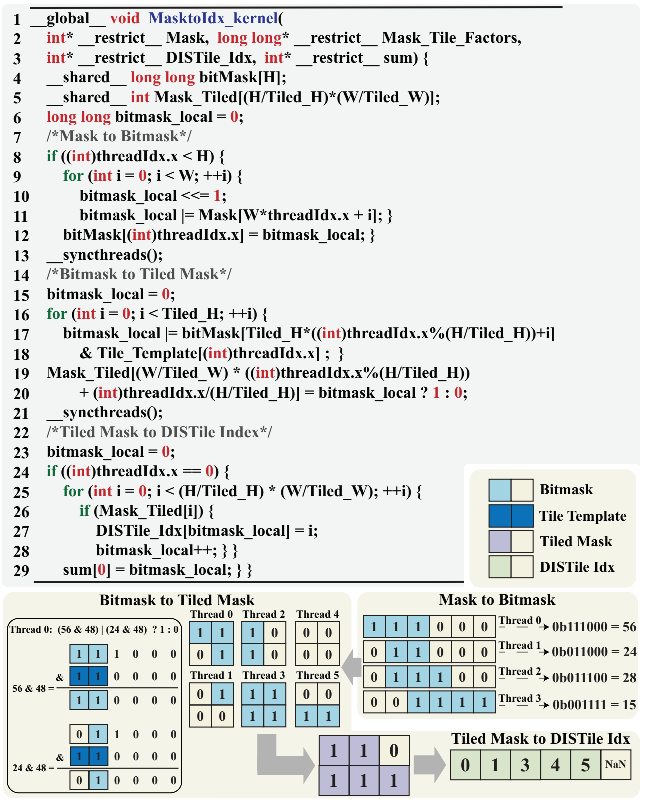
3.4 DynaSpa Runtime

The previous sections present the whole offline process for automated optimization of DNN operators with spatial sparsity. What remains to be discussed is the runtime architecture of DynaSpa, including efficient mask generation, PolyKernel selection, and memory planning.

Bitmask & Bitwise DISTile Generation. Due to the input-dependent spatial-sparsity masks in the workloads, DynaSpa needs to select the most proper GPU kernel from PolyKernel and extract the corresponding `DISTiles` at runtime. Two critical factors govern the selection of the optimal kernel implementation from PolyKernel: DISTile granularity and the total number of DISTiles. In DynaSpa, although PolyKernel is capable of adjusting launch configurations based on the total number of DISTiles, we opt for fixed launch configurations to minimize overhead and avoid potential performance fluctuations. Therefore, the feasibility and execution time of each implementation in PolyKernel can be accurately assessed via offline profiling and runtime table enumeration.

To this end, our runtime calls for an efficient implementation for DISTiles generation when given a granularity selection. We design our DISTile generation process based on bitmasks and bitwise operations. As shown in Figure 8, by leveraging GPU parallelism and highly efficient bitwise operations, we first transform the sparsity mask to the bitmask. Then, the tiled mask is obtained based on the bitmask and tile templates. Here, we carefully organize the thread access pattern to avoid bank conflicts. Finally, the DISTile indexes are obtained accordingly. The whole process can be scaled to multiple thread blocks with different granularity as inputs. After that, we can simply check the profiling table to find the candidates that can accommodate the number of DISTiles for the given input, and then select the best-performing kernel based on the shortest execution time from these candidates.

Memory Planning. Memory planning is important for efficient deep learning inference on GPU. Unlike the training phase, there is



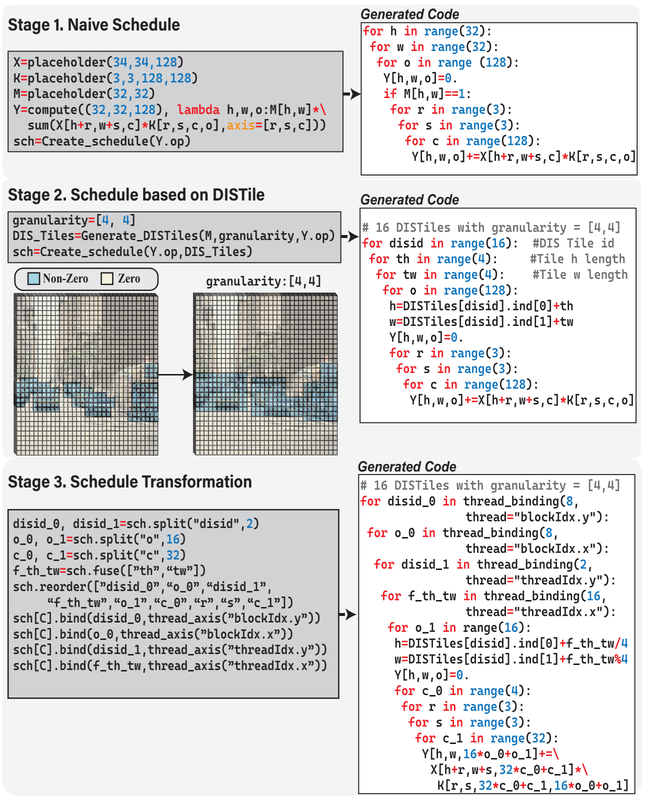
```
__global__ void MasktoIdx_kernel(
  int* __restrict__ Mask, long long* __restrict__ Mask_Tile_Factors,
  int* __restrict__ DISTile_Idx, int* __restrict__ sum) {
  __shared__ long long bitMask[H];
  __shared__ int Mask_Tiled[(H/Tiled_H)*(W/Tiled_W)];
  long long bitmask_local = 0;
  /*Mask to Bitmask*/
  if ((int)threadIdx.x < H) {
    for (int i = 0; i < W; ++i) {
      bitmask_local <<= 1;
      bitmask_local |= Mask[W*threadIdx.x + i]; }
    bitMask[(int)threadIdx.x] = bitmask_local; }
  __syncthreads();
  /*Bitmask to Tiled Mask*/
  bitmask_local = 0;
  for (int i = 0; i < Tiled_H; ++i) {
    bitmask_local |= bitMask[Tiled_H*((int)threadIdx.x%(H/Tiled_H))+i]
      & Tile_Template[(int)threadIdx.x] ;  }
  Mask_Tiled[(W/Tiled_W) * ((int)threadIdx.x%(H/Tiled_H))
      + (int)threadIdx.x/(H/Tiled_H)] = bitmask_local ? 1 : 0;
  __syncthreads();
  /*Tiled Mask to DISTile Index*/
  bitmask_local = 0;
  if ((int)threadIdx.x == 0) {
    for (int i = 0; i < (H/Tiled_H) * (W/Tiled_W); ++i) {
      if (Mask_Tiled[i]) {
        DISTile_Idx[bitmask_local] = i;
        bitmask_local++; } }
    sum[0] = bitmask_local; } }
```

Figure 8: GPU kernel for `Generate_DISTiles` via bitmasks and bitwise operations.

no need to store intermediate layer representations throughout the entire inference process. Utilizing static memory planning allows us to allocate a limited number of output buffers and reuse them across various layers. This approach significantly reduces memory allocation time. Without loss of generality, assume that we have two memory buffers and a neural network with a sequential structure. Tensor operations can be seamlessly executed by using one buffer for input data and the other for output, swapping them as required when going through the neural network. However, we will encounter a problem for a spatially-sparse neural network inference with such a double-buffer strategy. Unlike dense tensor outputs, which overwrite an entire block of memory, spatially-sparse tensor outputs overwrite only a sparse set of memory addresses within a block. This implicitly requires a memory block initialized with 0. Therefore, DynaSpa leverages a triple-buffer strategy. We use one for input, another for output, and asynchronously initialize the third for the next spatially-sparse operation. The initialization latency can be hidden by the computation latency without blocking the inference. To implement this strategy, DynaSpa leverages `clEnqueueFillBuffer` in OpenCL and `cuda stream` in Jetson GPUs.

4 Implementation

We integrate DynaSpa with TVM [27], a code-generation tool. By leveraging TVM APIs, DynaSpa creates tunable computation templates for spatially-sparse operators. Figure 9 illustrates an example



```
X=placeholder(34,34,128)
K=placeholder(3,3,128,128)
M=placeholder(32,32)
Y=compute((32,32,128), lambda h,w,o:M[h,w]*\
  sum(X[h+r,w+s,c]*K[r,s,c,o],axis=[r,s,c]))
sch=Create_schedule(Y.op)
```

```
for h in range(32):
 for w in range(32):
  for o in range (128):
   Y[h,w,o]=0.
   if M[h,w]==1:
    for r in range(3):
     for s in range(3):
      for c in range(128):
       Y[h,w,o]+=X[h+r,w+s,c]*K[r,s,c,o]
```

```
granularity=[4, 4]
DIS_Tiles=Generate_DISTiles(M,granularity,Y.op)
sch=Create_schedule(Y.op,DIS_Tiles)
```

```
# 16 DISTiles with granularity = [4,4]
for disid in range(16):  #DIS Tile id
 for th in range(4):     #Tile h length
  for tw in range(4):    #Tile w length
   for o in range(128):
    h=DISTiles[disid].ind[0]+th
    w=DISTiles[disid].ind[1]+tw
    Y[h,w,o]=0.
    for r in range(3):
     for s in range(3):
      for c in range(128):
       Y[h,w,o]+=X[h+r,w+s,c]*K[r,s,c,o]
```

```
disid_0, disid_1=sch.split("disid",2)
o_0, o_1=sch.split("o",16)
c_0, c_1=sch.split("c",32)
f_th_tw=sch.fuse([”th”,“tw”])
sch.reorder([”disid_0”,“o_0”,“disid_1”,
     “f_th_tw”,“o_1”,“c_0”,“r”,“s”,“c_1”])
sch[C].bind(disid_0,thread_axis(”blockIdx.y”))
sch[C].bind(o_0,thread_axis(”blockIdx.x”))
sch[C].bind(disid_1,thread_axis(”threadIdx.y”))
sch[C].bind(f_th_tw,thread_axis(”threadIdx.x”))
```

```
# 16 DISTiles with granularity = [4,4]
for disid_0 in thread_binding(8,
               thread="blockIdx.y"):
 for o_0 in thread_binding(8,
               thread="blockIdx.x"):
  for disid_1 in thread_binding(2,
               thread="threadIdx.y"):
   for f_th_tw in thread_binding(16,
               thread="threadIdx.x"):
    for o_1 in range(16):
     h=DISTiles[disid].ind[0]+f_th_tw/4
     w=DISTiles[disid].ind[1]+f_th_tw%4
     Y[h,w,o]=0.
     for c_0 in range(4):
      for r in range(3):
       for s in range(3):
        for c_1 in range(32):
         Y[h,w,16*o_0+o_1]+=\
          X[h+r,w+s,32*c_0+c_1]*\
           K[r,s,32*c_0+c_1,16*o_0+o_1]
```

Figure 9: Generated computation scheme sample (pseudo-code in a Python-like syntax) for spatiall-sparse Conv2D.

of generated code for Conv2D. In the first stage, we declare the input tensors. Then, we utilize an index-based lambda expression for tensor computation [27, 28]. The generated code at this stage retains the sparse loop iteration with poor access patterns. In the second stage, our DISTile generation outputs `DIS_Tiles`, encompassing all related DISTile information (i.e., ID, index, and granularity). We then leverage the generated `DIS_Tiles` to compose locally dense computation tiles into a virtual dense loop. Finally, we can apply all existing schedules to our spatially-sparse operation, including loop transformations (e.g., split, fuse), computation management (e.g., compute_at), and parallelism (e.g., bind, vectorize). Throughout this process, DynaSpa automatically maintains the necessary indexing and boundary checking. Changing the parameter values (e.g., granularity, split) in the tunable template can generate different computation patterns. DynaSpa utilizes TVM APIs to incorporate all possible patterns into the search space. After reducing the search space with the analytical performance model, DynaSpa employs the PolyKernel Auto-Scheduler to identify optimal computation candidates.

5 Evaluation

We evaluate the performance of generated spatially-sparse programs at two levels: single operators and entire neural networks.

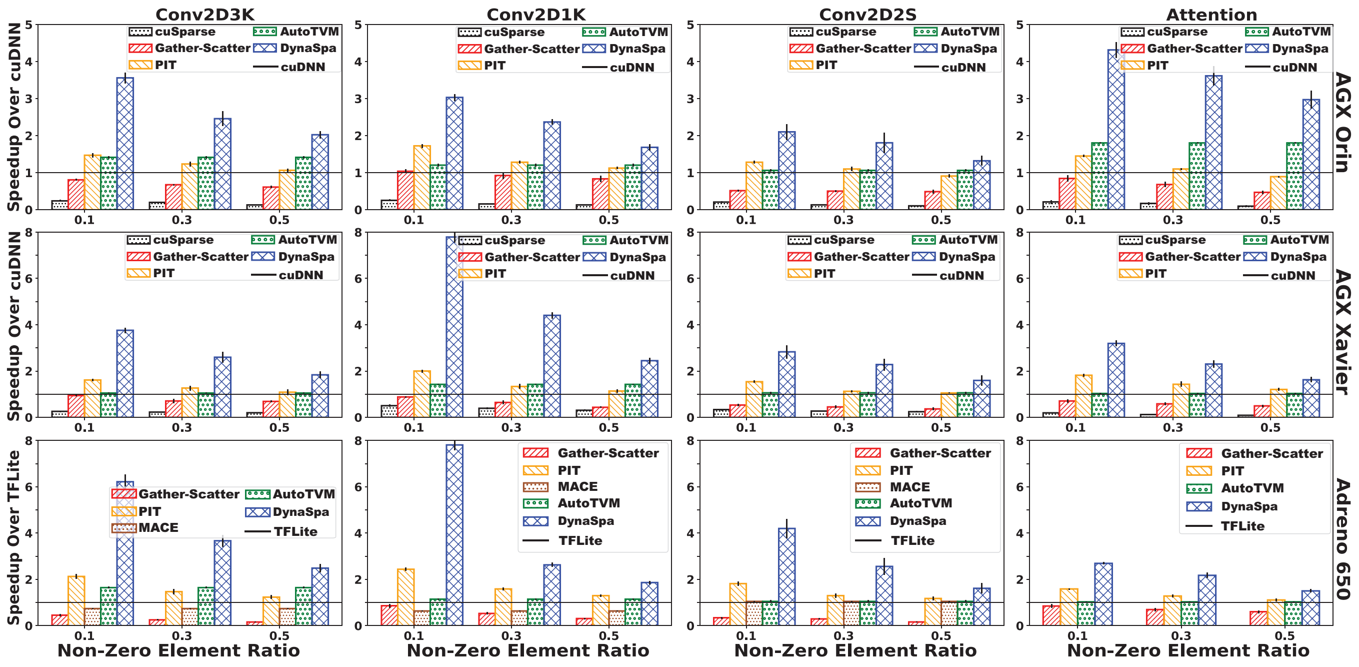


Figure 10: Single operator performance on embedded and mobile GPUs.

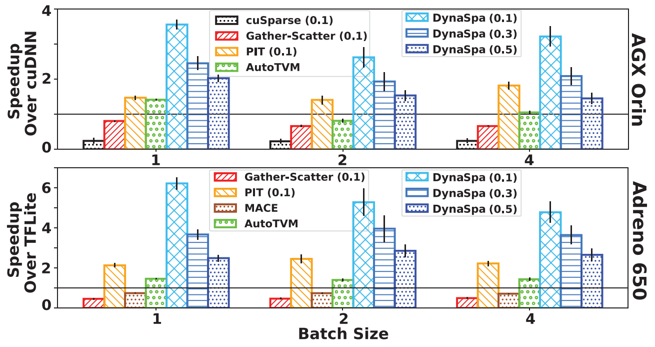


Figure 11: Single operator performance with different batch sizes.

For each level of evaluation, we compare DynaSpa against the state-of-the-art tensor compiler and hardware-specific vendor libraries. All generated tensor programs are evaluated on three hardware platforms: two embedded GPUs (NVIDIA Jetson AGX Orin and NVIDIA Jetson AGX Xavier) and a mobile GPU (an Android platform with a Qualcomm Adreno 650 GPU). We set the number of PolyKernels in DynaSpa to be 6 for all experiments. We use float32 as the data type for all evaluations.

5.1 Single Operator Benchmark

Workloads. We first evaluate DynaSpa on a set of spatially-sparse deep learning operators from three representative DNNs with spatial sparsity: DynConv [16] convolutional model for image recognition, A-ViT [18] vision transformer for image recognition, and SIGE [22] image editing and generation tasks. There are in total 30 types of operators with unique shapes and configurations, where we categorize them into four major types: conv2d with 3×3 kernels (Conv2D3K), conv2d with 1×1 kernels (Conv2D1K), conv2d with 3×3 kernels and stride 2 (Conv2D2S), and batch MatMul in transformer multi-head attention (Attention). These are the four major types of spatially-sparse tensor operators in our single-operator experiments. All three spatially-sparse DNNs can be trained or configured to operate at different levels of density (i.e., the fraction of zero elements in the sparsity mask). For our single-operator experiments, we choose three densities, i.e., 0.1, 0.3, and 0.5. Since densities are expected values that can fluctuate based on inputs, we feed all inputs from their respective dataset and choose the operations with the densities that fall within 0.05 deviation, i.e., $0.1 : [0.05, 0.15]$, $0.3 : [0.25, 0.35]$, and $0.5 : [0.45, 0.55]$. We will see the performance of these spatially-sparse DNNs using their native sparsity masks across layers in the end-to-end network benchmark in Section 5.2.

Baselines. We include different state-of-the-art baselines for embedded and mobile platforms, respectively. For embedded GPUs, we include cuDNN [29], cuSparse [24], AutoTVM [27], PIT [31] and Gather-Scatter [15, 59]. cuDNN is a library for dense DNN operators on Jetson GPUs. cuSparse is the NVIDIA library for sparse tensor operations. AutoTVM is an automated tensor compiler for dense operators. PIT is an automated tensor compiler for dynamic sparse operators. PIT does not support sparsity in the two spatial axes for convolution. To compare performance, we first use the explicit image-to-column algorithm (im2col) [60], a method that converts convolution into matrix multiplication by restructuring the input data, then use PIT. We did not include the time taken for the PIT conversion operation in the evaluation. Gather-Scatter is an implementation that supports spatial sparsity. For Conv operations and Jetson GPUs, we have a specially optimized Gather-Scatter kernel called TorchSparse [59]. For mobile GPU, we include AutoTVM, PIT, TFLite [30], and MACE [61]. TFLite is a library from Google for deploying models on mobile. MACE is a deep-learning inference

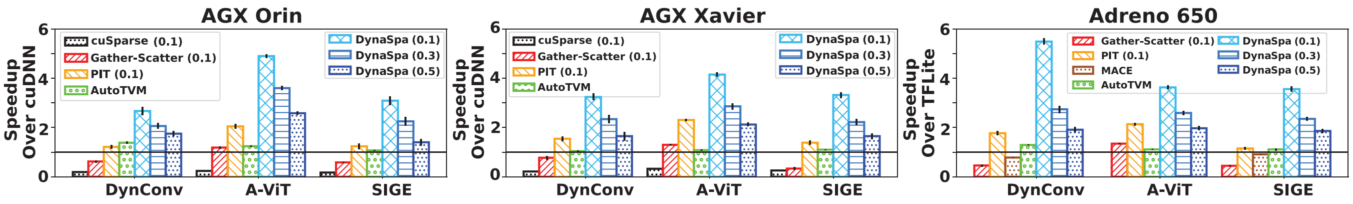


Figure 12: The end-to-end speedup ratios for three spatially sparse DNN workloads on three GPU platforms.

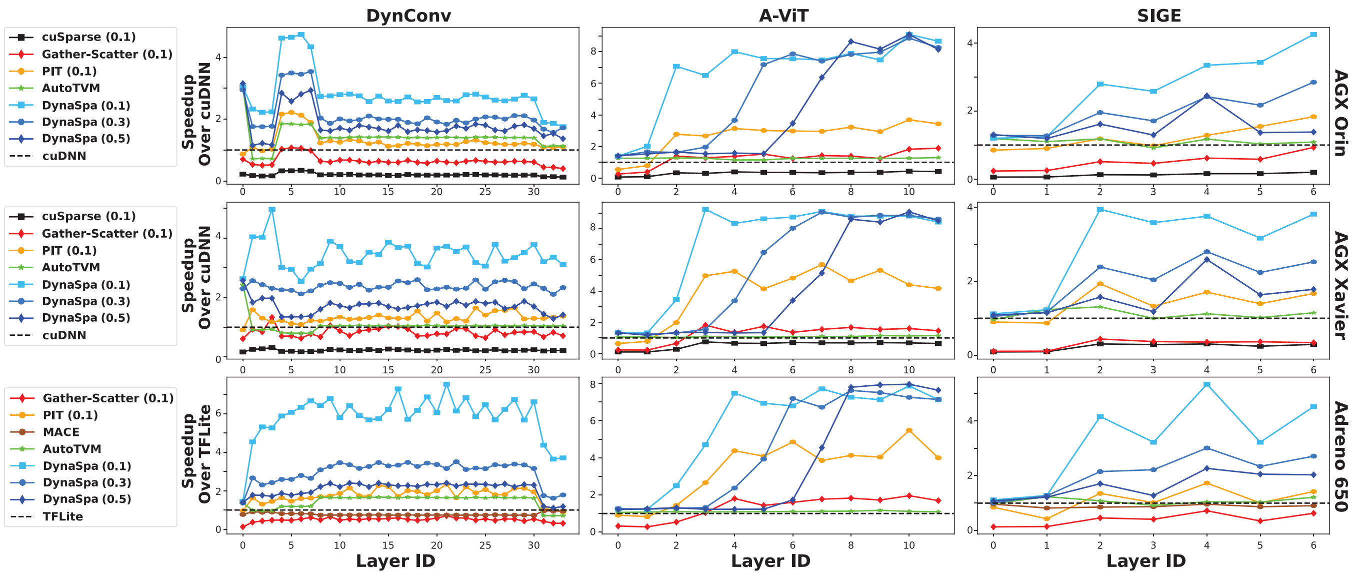


Figure 13: The performance of spatial sparse operators in the order of operator execution for three DNNs.

framework from Xiaomi for mobile. In addition, we evaluate all baselines that deal only with dense operators on the corresponding dense DNN operations.

Performance Metrics. Since the execution times vary greatly within each type of operation, we select one baseline as the standard (cuDNN for embedded GPUs; TFLite for mobile GPU) and compare the speedup ratios of each method against it. We calculate the geometric mean of the speedup ratio of all operators with sparsity masks that fall into the same category. The error bar denotes the standard deviation of the speedup ratio of each operator.

Results. As shown in Figure 10, DynaSpa outperforms all other baselines by a large margin ($\times 1.3 \sim \times 4.4$ speedup on Jetson AGX Orin GPU, $\times 1.6 \sim \times 7.7$ speedup on Jetson AGX Xavier GPU, and $\times 1.5 \sim \times 7.8$ speedup on Adreno mobile GPU) for all spatially-sparse operators on all platforms. Sparse vendor library cuSparse cannot achieve performance levels comparable to cuDNN, even having a 0.1 sparsity density. The reasons are twofold. First, cuSparse is designed to leverage extremely low densities, unfit for spatially sparse DNNs. Second, cuSparse usually requires explicit format transformation, involving a large number of global memory operations, which further degrade the kernel performance. AutoTVM generally outperforms cuDNN and TFLite but does not leverage spatial sparsity. PIT performs similarly to AutoTVM but falls short of DynaSpa's efficiency because its kernels are generated from a limited set of dense kernels, which may not suit the current sparse data.

To the best of our knowledge, Existing vendor libraries on mobile (TFLite) only support static weight sparsity. Moreover, MACE lacks mobile GPU support for attention operations (i.e. batchMatMul) [62]. Gather-Scatter performs worse than cuDNN and AutoTVM in almost all scenarios. In addition, Gather-Scatter has relatively better results on NVIDIA GPUs and Conv operations due to having a manually optimized implementation [59]. Other sparse tensor compilers only support static weight sparsity [35–37], which achieves performance that is comparable to, or even less than, AutoTVM. Therefore, we have chosen not to include them for conciseness. Furthermore, our analysis extends to comparing the performance of Conv2D3K across different batch sizes. Considering the constraints of resource-limited devices, which typically utilize lower batch numbers during inference, our comparison focuses on batch sizes 1, 2, and 4. As illustrated in Figure 11, DynaSpa consistently outperforms all other baselines across these batch sizes.

5.2 End-to-End Network Benchmark

Workloads. We evaluate the end-to-end inference time for the previous three spatially-sparse DNNs: DynConv [16] convolutional model for image recognition, A-ViT [18] vision transformer for image recognition, and SIGE [22] image editing and generation

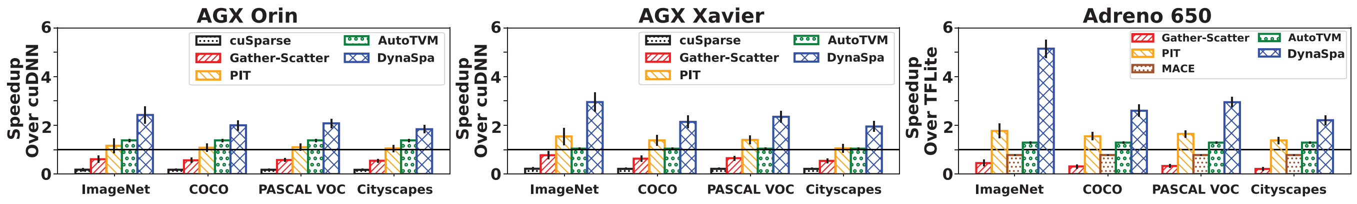


Figure 14: The end-to-end speedup ratios on different datasets.

tasks. It is essential to differentiate between the two cases. In the single-operator benchmark, a sparsity density value of 0.3 is defined as operators having a sparsity density within the interval $[0.25, 0.35]$, thereby excluding all other operators. However, in the end-to-end benchmark, 0.3 indicates that DNNs are trained with an *expected/averaged* sparsity density of 0.3 across all layers. Additionally, we conduct end-to-end experiments on various datasets including ImageNet [63], COCO [64], PASCAL VOC [65], and Cityscapes [66], each representing distinct application scenarios. We process the images from these datasets through DynConv. In each dataset, we select $\min\{1k, 10\%\}$ of the images for offline searching of candidate kernels using DynaSpa, while the remaining are set aside for testing inference acceleration.

Baselines. We include ONNX Runtime with the cuDNN backend, AutoTVM, TFLite, and MACE as baselines. All workloads contain both spatially-sparse and dense operators. When we apply DynaSpa to dense operators, it removes the parts about DISTiles and PolyKernel, which defaults to AutoTVM but with a pruned search space.

Results. As shown in Figure 12, DynaSpa still achieves a significant speedup in terms of end-to-end inference on all GPU platforms, i.e., ×1.4 ∼ ×4.8 speedup for Jetson AGX Orin, ×1.6 ∼ ×4.2 speedup for Jetson AGX Xavier, and ×1.9 ∼ ×5.5 speedup for Adreno 650. Compared to the single-operator benchmark, end-to-end inference usually has a better lower bound but a bit worse upper bound. The reason is due to the non-uniform sparsity levels across layers. As shown in Figure 13, A-ViT and SIGE have an increasing level of sparsity when they go deeper. Figure 14 displays the end-to-end inference speedup across the datasets. DynaSpa achieves ×1.8 ∼ ×2.4 speedup for AGX Orin, ×1.95 ∼ ×2.9 for AGX Xavier, and ×2.2 ∼ ×5.1 for Adreno 650. This result indicates that, in practical inference scenarios, DynaSpa continues to achieve significant acceleration with data from similar contexts, even when encountering such data for the first time. The end-to-end inference also includes all related runtime overhead as well as memory planning strategies. As shown in Figure 12 and 14, thanks to latency hiding in GPU, our triple-buffer strategy does not introduce perceptible overhead. In addition, the speedup pattern of DynaSpa shares similarities with AutoTVM. This is because both of them can enjoy the benefit of automated kernel optimization, especially when a manually tuned GPU kernel is sub-optimal. However, DynaSpa still achieves at least ×1.3 ∼ ×4.6 speedup compared to AutoTVM.

Accuracy Impact of DynaSpa. DynaSpa does not affect the accuracy of dynamically sparse networks. This is because the computation may only involve additional calculations with zeros, but it does not omit any non-zero elements. The extra calculations with zeros do not influence the final results. As shown in Table 3, the accuracy of three different dynamically sparse networks is presented for the original network, the dynamic network, and the dynamic network with DynaSpa applied. As SIGE is a generative network, the accuracy is represented by the Learned Perceptual Image Patch Similarity (LPIPS) metric, comparing the generated images with the ground truth. From the table, it is evident that DynaSpa provides system support for the implementation of spatially sparse networks without altering their accuracy.

Table 3: Accuracy Impact Comparison.

Spatially Sparse Network	Original Accuracy	Dynamic Accuracy	DynaSpa Accuracy	DynaSpa Speedup
DynConv	78.25%	75.71%	75.71%	×2.4 ∼ ×5.1
A-ViT	71.3%	71.0%	71.0%	×3.2 ∼ ×4.6
SIGE	0.409	0.416	0.416	×2.4 ∼ ×3.0

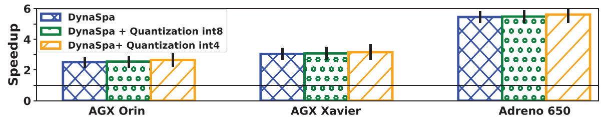


Figure 15: The end-to-end speedup ratios on ImageNet for DynaSpa and DynaSpa+quantization.

Quantization Effects on DynaSpa. Quantization significantly affects the sparsity in neural networks by altering the distribution of weights and activations. However, its impact on mask selection in dynamically sparse networks, such as DynConv, is limited. In DynConv, the mask is selected based on the effective regions of the input, and quantization does not significantly affect the boundaries of these regions. The main benefit of quantization lies in increasing the sparsity of values within the masked regions, making the mask sparser during generation. Figure 15 shows the speed of DynConv on ImageNet when using DynaSpa and DynaSpa with quantization. Here, we focus on the impact of quantization on DynaSpa and do not take into account the deployment speedups of quantization itself. We use Quantization Aware Training (QAT) to train with *int*4 and *int*8 values, where the values are stored in $float32$ format during actual computation to exclude any speed improvements due to quantization deployment. As shown in Figure15, DynaSpa achieves a slight speed improvement when using int4 and int8, indicating that the enhancement in mask sparsity has a limited effect on the overall end-to-end performance.

5.3 Search Time & Overhead

Search Time Reduction. DynaSpa proposes an analytical upper-bound model to reduce the PolyKernel search space. The superior single-operator performance implies that the DynaSpa analytical

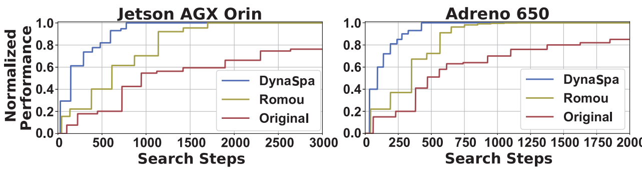


Figure 16: Search steps reduction of DynaSpa. The Y-axis is the performance relative to the best speedup.

model is capable of reducing the search space without compromising the performance. In this section, we evaluate our analytical model in terms of its ability to reduce auto-scheduler search time. As shown in Figure 16, we compare DynaSpa with two baselines: 1) the original search space; 2) the search space pruning method proposed in Romou [7]. We pick the most frequently appeared 3×3 conv operator (i.e., $H, W, C_{in}, C_{out} = [40, 40, 256, 256]$) in DynConv as the test case on two GPU platforms. The original search space is too large to be affordable. Compared to Romou, DynaSpa can further reduce 46% and 58% auto-scheduling time for Jetson and Adreno GPUs, respectively.

DynaSpa Runtime Overhead. Compared to other existing tensor compiler techniques, DynaSpa incurs additional runtime overhead for DISTile generation and PolyKernel dispatch. It is crucial to note that the number of thread blocks exhibits a linear scaling relationship with the number of PolyKernels, as depicted in Figure 8. In all prior experiments, the number of PolyKernels was set to 6, which is below the level of parallelism supported by GPUs. Consequently, the runtime overhead remains nearly constant for each platform, regardless of the sparsity levels and operators. To evaluate the overhead of DynaSpa runtime, we continue to use the most frequently appeared 3×3 conv operator in DynConv as our test case on three GPU platforms. As shown in Table 4, The overhead is primarily composed of three components: the bitmask generation from the spatial mask, the selection of kernel candidates, and the generation of non-zero DISTile Indices. As described in Section 3.4, the bitmask generation can be parallelized by the GPU, accounting for approximately 4% of the total time in the worst-case scenario. The Index generation and kernel selection, which leverage bitwise operations, take less than 2% of the time. Thus, even in the worst scenario (i.e., Adreno 650 with 0.1 density), the runtime overhead remains below 6%. Therefore, the overhead in DynaSpa is acceptable in terms of the achieved speedup.

6 Discussion

While DynaSpa provides significant benefits in reducing computation overhead and optimizing performance for dynamically sparse networks, there are certain limitations that need to be addressed in future work.

From a device perspective, the current Analytical Performance Model we use for predicting performance on different devices is not yet accurate enough. The diversity in hardware architectures and varying characteristics of mobile and edge devices make it challenging to generalize a model that can accurately predict performance across all platforms. Further refinement of the performance model is necessary to ensure better alignment with real-world results.

Table 4: Runtime overhead breakdown.

Overhead Process	Time Ratio	Description
Bitmask Generation	$\approx 4\%$	In parallel
DISTile Idx Generation	$< 1\%$	Bitwise Operation
Kernel Selection	$< 1\%$	Bitwise Operation
Total	$< 6\%$	

From a dynamic perspective, our current focus is primarily on dynamic spatial sparsity. While this has proven effective, we recognize that better support for other types of sparsity, particularly channel-wise sparsity, is required. Channel-wise sparsity offers additional opportunities for optimizing DNNs by reducing redundant computations across channels, and its integration into DynaSpa is an important area for future research. By extending DynaSpa's capabilities to better handle channel-wise sparsity, we can further enhance its applicability to a wider range of network architectures.

7 Conclusion

In this paper, we propose DynaSpa, an automated kernel optimization framework for DNNs with dynamic spatial sparsity. Unlike traditional vendor libraries and tensor compilers, which are only effective under extremely high levels of sparsity, DynaSpa is capable of optimizing tensor computations across all sparsity levels, especially for the moderate range 50% $\sim$ 90%. Equiped with our novel PolyKernel auto-scheduler with run-time dispatch that supports dynamic sparse workload, as well as an analytical model with detailed analysis of performance-limiting factors that helps greatly with search space reduction, DynaSpa is able to effectively exploit the trade-off between computation reduction and memory access, and enable users to apply dense scheduling primitives to spatially-sparse operators. We have conducted thorough experimental evaluations of DynaSpa across a multitude of settings, and observe that it can achieve up to $\times 5.8$ end-to-end speedup across various GPU platforms with no loss in accuracy compared with dynamic networks.

8 Disclaimer

9 Acknowledgements

This work is in part supported by the National Science Foundation grants IIS-2107200 and CNS-2038658.

References

[1] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating {DNN} model porting for {On-Device} inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719, 2021.

[2] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1423–1431. IEEE, 2019.

[3] Hao Wu, Xuejin Tian, Minghao Li, Yunxin Liu, Ganesh Ananthanarayanan, Fengyuan Xu, and Sheng Zhong. Pecam: privacy-enhanced video streaming and analytics via securely-reversible transformation. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 229–241, 2021.

[4] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Yaqin Zhang, and Yunxin Liu. Adaptivenet: Post-deployment neural architecture adaptation for diverse edge environments. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2023.

[5] Kai Huang and Wei Gao. Real-time neural network inference on extremely weak devices: agile offloading with explainable ai. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 200–213, 2022.

[6] Rongjie Yi, Ting Cao, Ao Zhou, Xiao Ma, Shangguang Wang, and Mengwei Xu. Boosting dnn cold inference on devices. 2023.

[7] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *Proceedings of the 28th Annual International Conference on Mobile Computing and Networking*, 2022.

[8] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th conference on embedded networked sensor systems*, pages 476–488, 2020.

[9] Yuyang Leng, Renyuan Liu, Hongpeng Guo, Songqing Chen, and Shuochao Yao. Scaleflow: Efficient deep vision pipeline with closed-loop scale-adaptive inference. In *Proceedings of the 31st ACM International Conference on Multimedia*, pages 1698–1706, 2023.

[10] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[11] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR*, 2016.

[12] 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.

[13] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR*, 2017.

[14] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR*, 2019.

[15] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. Sbnet: Sparse blocks network for fast inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8711–8720, 2018.

[16] Thomas Verelst and Tinne Tuytelaars. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, pages 2320–2329, 2020.

[17] Fanrong Li, Gang Li, Xiangyu He, and Jian Cheng. Dynamic dual gating neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5330–5339, 2021.

[18] Hongxu Yin, Arash Vahdat, Jose M Alvarez, Arun Mallya, Jan Kautz, and Pavlo Molchanov. A-vit: Adaptive tokens for efficient vision transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10809–10818, 2022.

[19] Lingchen Meng, Hengduo Li, Bor-Chun Chen, Shiyi Lan, Zuxuan Wu, Yu-Gang Jiang, and Ser-Nam Lim. Adavit: Adaptive vision transformers for efficient image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12309–12318, 2022.

[20] Yongming Rao, Zuyan Liu, Wenliang Zhao, Jie Zhou, and Jiwen Lu. Dynamic spatial sparsification for efficient vision transformers and convolutional neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[21] 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.

[22] Muyang Li, Ji Lin, Chenlin Meng, Stefano Ermon, Song Han, and Jun-Yan Zhu. Efficient spatially sparse inference for conditional gans and diffusion models. *Advances in Neural Information Processing Systems*, 35:28858–28873, 2022.

[23] Zihao Yu, Haoyang Li, Fangcheng Fu, Xupeng Miao, and Bin Cui. Fisedit: Accelerating text-to-image editing via cache-enabled sparse diffusion inference. *arXiv preprint arXiv:2305.17423*, 2023.

[24] NVIDIA. Nvidia cusparse, 2023. URL https://docs.nvidia.com/cuda/cusparse.

[25] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 672–687. Springer, 2018.

[26] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188, 2014.

[27] 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.

[28] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[29] NVIDIA. Nvidia cudnn, 2024. URL https://developer.nvidia.com/cudnn.

[30] TFLite. Tensorflow lite, 2024. URL https://www.tensorflow.org/lite.

[31] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, et al. Pit: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 331–347, 2023.

[32] 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*, pages 13–20, 2017.

[33] 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*, pages 5533–5543. PMLR, 2020.

[34] 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*, pages 9224–9232, 2018.

[35] 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.

[36] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[37] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[38] 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.

[39] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3: 208–222, 2021.

[40] Kai Zhu, WY Zhao, Zhen Zheng, TY Guo, PZ Zhao, JJ Bai, Jun Yang, XY Liu, LS Diao, and Wei Lin. Disc: A dynamic shape compiler for machine learning workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 89–95, 2021.

[41] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.

[42] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 681–699, 2023.

[43] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems*, 4:848–863, 2022.

[44] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

[45] NVIDIA. Filling the performance gap in convolution implementations for nvidia gpus, GTC Silicon Valley-2019. URL https://developer.nvidia.com/gtc/2019/video/s9218.

[46] Daniel Peter Playne and Ken Hawick. A new algorithm for parallel connected-component labelling on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1217–1230, 2018.

[47] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[48] Andreas Krause and Daniel Golovin. Submodular function maximization. *Tractability*, 3(71-104):3, 2014.

[49] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14:265–294, 1978.

[50] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[51] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. *Advances in Neural Information Processing Systems*, 33:14807–14819, 2020.

[52] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *8th International Conference on Learning Representations, ICLR*, 2020.

[53] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[54] NVIDIA Corporation. *CUDA Toolkit Documentation*, 2023. URL https://developer.nvidia.com/cuda-toolkit.

[55] Khronos OpenCL Working Group. *The OpenCL Specification*, 2023. URL https://www.khronos.org/opencl/.

[56] Design Guide. Cuda c++ best practices guide. 2020.

[57] NVIDIA. Cuda warps and occupancy, 2011. URL https://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf.

[58] Vasily Volkov. *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[59] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. Torchsparse: Efficient point cloud inference engine. *Proceedings of Machine Learning and Systems*, 4:302–315, 2022.

[60] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

[61] XiaoMi. Mobile ai compute engine, 2023. URL https://github.com/XiaoMi/mace.

[62] XiaoMi. Mace operator list, 2023. URL https://mace.readthedocs.io/en/latest/user_guide/op_lists.html.

[63] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[64] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[65] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88:303–338, 2010.

[66] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.