

DyGen: A Constant-Time Kernel Generator for Dynamic-Shape Neural Networks

Yuhan Kang^{1,*}, Wenrui Zhang^{2,*}, Dong Chen², Yang Shi^{1,†}, Jianchao Yang^{1,†}, Zeyu Xue¹, Jing Feng¹, Mei Wen¹

¹Key Laboratory of Advanced Microprocessor Chips and Systems, College of Computer Science, National University of Defense Technology, Changsha 410073, China

Emails: {kangyuhan, shiyang14, yangjianchao, xuezeyu, fengjing22, meiwen}@nudt.edu.cn

²Huawei Technologies, Beijing 100004, China

Emails: {zhangwenrui, chendong}@huawei.com

*These authors contributed equally to this work. † Corresponding author

Abstract—In recent years, dynamic-shape neural networks have been widely adopted in intelligent applications, such as Mixture-of-Experts based large language models and computer vision tasks. However, in dynamic scenarios, operator shapes are determined at runtime. This leads to prohibitively expensive compilation times for existing static compilers, as they must search across a vast optimization space to identify the best configuration. To address the need for efficient optimization of dynamic-shape neural networks, we present DyGen (Dynamic-shape Kernel Generator)—a lightweight, two-stage compiler plug-in on GPU platforms. In the offline stage, DyGen employs deliberately crafted pruning rules to construct a compact candidate configuration set for the target hardware, then select the configuration of the high-performance kernel to train a configuration generation model. During the online stage, dynamic operator information is directly fed into the generator, which can quickly produce efficient kernel configurations without the need for costly search. Compared to state-of-the-art tensor compilers, DyGen improves inference performance by an average of 36%, while significantly reducing generation overhead from 9 seconds to 0.3 seconds.

Index Terms—Dynamic-Shape Neural Networks, Kernel Generation

I. INTRODUCTION

High-performance kernels form the foundation of intelligent applications such as deep neural networks (DNNs). In these applications, the core computations primarily rely on tensor operators such as convolution and general matrix multiplication (GEMM), whose execution efficiency largely determines overall system performance [1]–[4].

However, developing high-performance kernels typically requires deep expertise in both low-level hardware architectures and complex parallel programming models [5]. Consequently, developers have long relied on vendor-specific, hand-tuned operator libraries. For example, oneDNN [6] for x86 CPUs, and cuBLAS [7] and CUTLASS [8] for NVIDIA GPUs. While these libraries deliver excellent performance, they suffer from high development costs and limited portability, making it difficult to keep pace with the rapid evolution of hardware architectures.

To lower this barrier, a variety of tensor compilers have emerged in recent years (e.g., TVM [9], Triton [10]). These frameworks abstract away the operator development process and optimize tensor computations by searching across large design spaces of loop tiling configurations to determine the

best implementation for a given shape. Such compilers have significantly improved developer productivity and demonstrated strong results in static-shape networks.

Nonetheless, with the rise of large language models (LLMs) and emerging intelligent applications, dynamic-shape neural networks are increasingly becoming the focus of both research and practice. In these scenarios, operator input and output dimensions are often unknown until runtime.

For example, dynamic shapes arise in NLP (variable-length sequences), computer vision (diverse image resolutions), and GNNs (varying nodes and edges). These dynamic characteristics require tensor compilers to rapidly generate high-performance kernels for shape-specific operators at runtime.

Current solutions within tensor compilers fall into two main categories: autotune-based and heuristic-based search. The former relies on template-solving strategies, where developers predefine candidate configurations, and the system iteratively tests them at runtime to select the optimal parameters. The latter employs heuristic algorithms, inferring reasonable scheduling parameters at runtime based on input sizes and decision rules, without depending on fixed configurations. Unfortunately, they still suffer from critical limitations. Template-based search is highly dependent on the quality and coverage of developer-provided candidates—insufficient coverage risks missing optimal solutions, while excessive candidates can dramatically increase runtime search and compilation costs. Heuristic-based methods, on the other hand, often rely on coarse rules that fail to accurately model critical GPU features such as memory hierarchy and access patterns, leading to suboptimal schedules. As shown in Table I, for large-scale or highly dynamic workloads, existing approaches struggle to balance compilation efficiency and runtime performance, making this issue a key bottleneck in high-performance operator generation.

TABLE I
COMPARISON OF DIFFERENT APPROACHES (MEASURED ON 4090).

Approach	Compilation	Compile Time	Hardware-aware	Performance
Autotune-based	~ 10 ¹ configs	~ 10 s	None	Worst
Heuristic-based	~ 10 ² configs	~ 10 ² s	Low	Medium
DyGen	1 config	< 1 s	High	Best

To address the challenge of rapidly generating high-performance kernels for dynamic shape operators, we propose

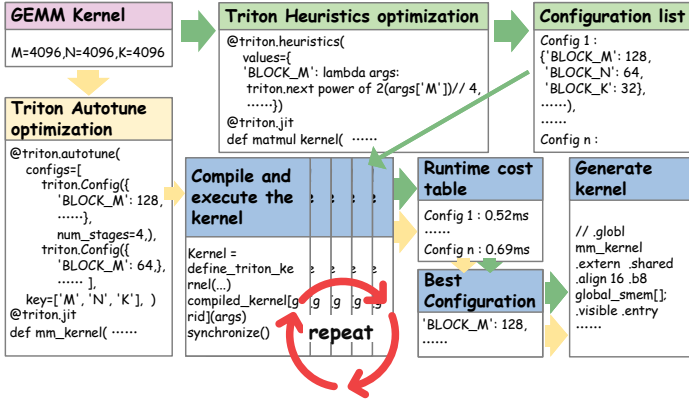


Fig. 1. Optimization Process in Static Tensor Compilers: Autotuning and Heuristic Techniques for Fixed Tensor Shapes.

a lightweight hardware-aware plug-in for existing tensor compilers. The core idea is to encapsulate the relationship between operator shapes and their optimal configurations into a set of polynomials through offline learning. This approach entirely bypasses the search process during online inference, enabling constant-time kernel generation.

The main contributions can be summarized as follows:

- **Efficient automatic kernel parameter selection:** We introduce a hardware-aware strategy that leverages offline learning to capture the mapping between operator sizes and optimal configurations, enabling low-latency configuration prediction in dynamic-shape neural networks.
- **Hardware-aware search space construction and dataset generation:** By combining operator classification with GPU architecture constraints, we filter and partition the solution space across diverse input sizes, reducing offline training costs while improving cross-hardware adaptability.
- **System implementation and cross-hardware validation:** We implement a complete system based on Triton and evaluate it on NVIDIA H100 and RTX 4090 GPUs. Experiments demonstrate that our method significantly reduces compilation and search overhead while maintaining kernel performance.
- **Performance advantages:** Our method achieves millisecond-level configuration prediction latency, high adaptability, and strong cross-hardware portability. When integrated with just-in-time compilation, it substantially improves inference throughput.

II. BACKGROUND AND RELATED WORK

A. Dynamic-shape Neural Network Models

Real-world applications often exhibit dynamic characteristics, necessitating dynamic-shape neural networks to handle more complex tasks [11], [12]. Representative scenarios include:

a) *Dynamic sequence lengths in NLP and LLMs:* Natural language models handle inputs of varying lengths, which causes fluctuating tensor shapes. Advanced strategies like bucketing and grouped padding help maintain dynamic shapes and reduce redundant computation. This issue is more pronounced during

the prefill stage in LLMs, where input lengths vary significantly. Dynamic batching and asynchronous requests further add variability to tensor shapes [13]–[15].

b) *Dynamic image resolutions:* In computer vision, varying input resolutions lead to tensor shape variation. Models like Faster R-CNN support dynamic inputs and use advanced pooling to process images of different resolutions effectively [16]–[18].

B. Static Tensor Compilers

Static tensor compilers like Triton and TVM use “auto-scheduling” techniques to explore the optimization space by searching for optimal loop tiling and kernel configurations for fixed tensor shapes. However, these techniques rely on static tensor shapes, making them unsuitable for dynamic workloads where shapes change at runtime. To address this, compilers like Triton provide optimization mechanisms such as autotuning (which benchmarks multiple candidate configurations to find the best one) and heuristics (which generate configuration spaces based on empirical rules), as shown in Fig. 1. While autotuning can identify the best configuration within a candidate set, the overhead increases as the set grows, and a smaller set may lead to suboptimal performance. Heuristic methods expand the search space but may overlook complex hardware characteristics, resulting in suboptimal performance and higher overhead. Both methods struggle to effectively handle dynamic shapes and large-scale scenarios, highlighting the need for new optimization techniques.

C. Dynamic Shape Compilers

With the growing demand for dynamic shape support, several solutions have emerged to optimize dynamic tensor operations. These compilers aim to efficiently handle dynamic tensors by combining features such as symbolic shape representation, dynamic memory allocation, and runtime shape-based kernel scheduling.

For example, Nimble [19] introduces a virtual machine (VM) based runtime that decouples the platform-independent control logic and platform-dependent kernel implementation. This VM allows dynamic models to run efficiently across diverse hardware platforms. DietCode [20] enhances auto-scheduling for dynamic tensor workloads by developing a shape-generic search space and cost model. However, Nimble and DietCode both rely on pre-compiled kernels or schedules for known shapes, which may lead to sub-optimal performance when confronted with unknown or highly variable shapes. Nimble also suffers from overheads introduced by symbolic shapes, which can negatively impact performance, especially in smaller workloads.

The recent cutting-edge work, MikPoly [21], focuses on optimizing dynamic shape tensor operators using a two-stage approach, where pre-optimized microkernels are dynamically composed at runtime. This method has shown promising results, but its efficiency heavily relies on accurate modeling and prediction of the microkernel performance, which can be quite challenging for developers without hardware expertise. Additionally, the runtime dynamic selection and composition

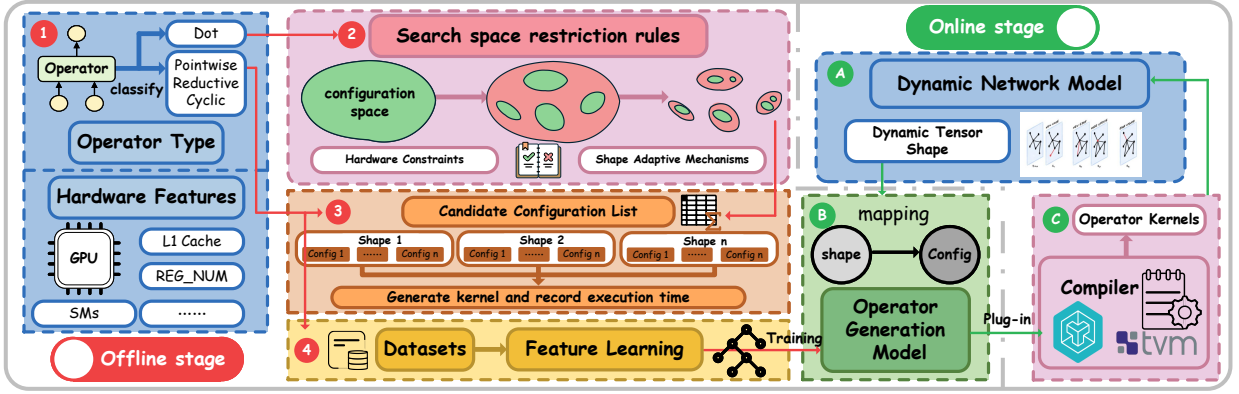


Fig. 2. Workflow of DyGen: Offline Kernel Generation and Online Dynamic Inference.

of microkernels introduce extra overhead. As a result, when applied to large-scale models, highly variable dynamic workloads, or diverse architectures, it may face significant challenges.

III. THE DYGEN DESIGN

Fig. 2 illustrates the overall workflow of DyGen, which consists of two main stages: offline operator kernel generation model training (the left part) and online dynamic inference (the right part). During the offline stage, a configuration generation model for operators is constructed by learning the configurations of high-performance operators on specific hardware. During online inference, this model is utilized to generate configuration parameters for dynamic operators in constant time. We describe each stage as follows.

A. Offline Stage: Operator Kernel Generation Model Training

In the offline stage, DyGen accelerates kernel generation for dynamic shape neural networks by constructing a candidate configuration set for each operator and training a configuration generation model. Initially, the system explores a comprehensive configuration space for each operator by iterating through a list of parameters, generating multiple candidate configurations. Then, DyGen filters out invalid or inefficient configurations using pruning rules based on hardware characteristics and the operator shape, ensuring that the generated configuration set is compact and efficient.

The goal of this stage is to establish an efficient kernel generation model that can quickly respond to dynamic shape changes in the online stage. Specifically, we provide a phased explanation, as illustrated in Fig. 2.

1) *Operator Classification*: Operators are classified into categories like pointwise operators, reductive operators, etc., based on their computational characteristics and memory access patterns. This taxonomy not only provides a unified perspective for characterizing diverse computational patterns but also provides structured guidance for compiler optimization and parameter tuning. The detailed classification criteria and representative examples are summarized in Table II.

The first three categories are simple in logic and have low hardware dependency, with configurations mainly based on tensor shapes and handled by fixed strategies. In contrast, Dot operators like GEMM and convolution are compute-intensive

TABLE II
CLASSIFICATION OF OPERATORS IN DYGEN.

Type	Characteristics	Examples
Pointwise	Element-wise independent ops	ReLU, Sigmoid, Add
Reductive	Aggregation on one dimension	Sum, Max, Softmax
Cyclic	Block accumulation with loop unrolling	NonZero, LayerNorm
Dot	Sub-block matrix multiplication	GEMM, Conv2D, Attention

and dominate runtime. These operators have a much larger configuration space, requiring fine-grained search and pruning strategies. Detailed filtering rules for Dot operators are discussed in the next subsection.

2) *Filtering Rules*: Dot operators (e.g., GEMM, convolution) are defined by multi-level nested loops, where performance is sensitive to configuration parameters like tile sizes, threads per block, pipeline depth, and split factors. A brute-force search for the optimal configuration leads to an exponential growth of the search space, represented by:

$$S = T_M \times T_N \times T_K \times N_{threads} \times N_{stages} \times S_{split}, \quad (1)$$

where T_M, T_N, T_K denote tile sizes along the respective dimensions, $N_{threads}$ the number of threads per block, N_{stages} the number of pipeline stages, and S_{split} the split factor along the reduction dimension. For large tensor sizes (e.g., $512 \times 512 \times 512$ GEMM), this search space becomes infeasible. Therefore, pruning the search space based on hardware constraints is essential.

Key hardware constraints include:

- **Thread Occupancy Rule**: Each GPU Streaming Multi-processor (SM) imposes strict limits on the number of concurrently resident threads, making thread occupancy the first feasibility check for any kernel configuration. For a given configuration, the number of threads per block must not exceed the SM's maximum thread capacity T_{max} :

$$threads_per_block = N_{threads} \times warp_size \leq T_{max}. \quad (2)$$

- **Shared Memory Rule**: Shared memory serves as a high-speed cache for intra-block data reuse. For GEMM, each thread block loads sub-blocks of the input matrices A and B into shared memory and employs multi-stage pipelining to hide global memory latency. A configuration is considered feasible only if the shared memory requirement does not exceed the SM's limit S_{max} :

$$SharedMem = (T_M + T_N) \times T_K \times byte \times N_{stages} \leq S_{max}. \quad (3)$$

- **Register Usage Rule:** Registers are private to each thread and crucial for instruction-level parallelism. Each thread requires registers for:
 - (1) Accumulator registers: Accumulator registers to hold partial results:

$$R_{acc} = \frac{T_M \times T_N}{threads_per_block}$$

- (2) Load-buffer registers: for caching shared memory loads across pipeline stages:

$$R_{loadbuf} = \frac{N_{stages} \times ((T_M + T_N) \times T_K)}{threads_per_block}$$

- (3) Overhead registers: for loop counters, address offsets, thread indices, and temporaries (typically a small constant, e.g., 20).

The total per-thread register requirement is then:

$$R_{thread} = R_{acc} + R_{loadbuf} + R_{overhead}$$

and the total per-block requirement must satisfy:

$$R_{block} = R_{thread} \times threads_per_block \leq R_{max}. \quad (4)$$

- **Operational Intensity Constraint Rule:** To ensure high compute utilization and avoid memory-bound performance, configurations are pruned using a tile-level performance model that estimates the ratio of compute operations to memory accesses. For a tile of size $T_M \times T_N \times T_K$:

$$Operational_intensity = \frac{T_M \times T_N \times T_K}{T_M \times T_K + T_N \times T_K}. \quad (5)$$

Specifically, if T_K is too small, memory traffic dominates; if $T_M \times T_N$ is too small, parallelism on specialized units (e.g., Tensor Cores) cannot be fully exploited. Tiles with operational intensity outside an empirically determined feasible range are discarded.

Even after applying these constraints, the remaining search space remains large. To further refine it, we introduce adaptive mechanisms based on empirical observations of matrix shapes (M, N, K) , exploiting memory locality, parallelism, and workload balance:

- **Cache-friendly Adaptation (Block Swizzling):** The performance of large-scale GEMM is often limited by the reuse rate of the L2 cache. To address this, $GROUP_M$ is used to partition the computation grid, ensuring that blocks within a group share the same A sub-block during the N iteration. This improves cache locality and reduces global memory traffic.
- **Pipeline Depth Adaptation:** As K increases, memory latency becomes the primary bottleneck. There is a non-linear relationship between K and the pipeline depth N_{stages} (e.g., $K \in [512, 8192]$ corresponds to $N_{stages} \in [2, 6]$). By adjusting the number of pipeline stages, latency is reduced and performance is optimized.
- **Thread parallelism adaptation:** The number of threads per block ($N_{threads}$) determines compute utilization. By

Polynomial Approximation Function for Configuration Prediction

```
def gen_num_stages(shape_detail_M, shape_detail_N, shape_detail_K):
    res = 0.1407*np.log(shape_detail_K) - 0.1193*np.log(shape_detail_M)
        - 0.2223*np.log(shape_detail_N) - 1.377e-5*shape_detail_K
        + 1.978e-5*shape_detail_M + 1.968e-5*shape_detail_N + 5.073
    candidates = [2, 3, 4, 5]
    return min(candidates, key=lambda x: abs(x - res))
```

Fig. 3. Examples of polynomial-approximated prediction functions learned for GEMM operator.

dynamically selecting the number of threads $N_{threads}$, the configuration search space is reduced.

- **Thread Block Adaptation (K_{split} Parallelism):** Assigning the entire K dimension to a single block reduces parallelism and leads to imbalance. Splitting K into sub-problems (K_{split}) allows concurrent execution, improving utilization and balancing the workload.

By combining these strategies—Block Swizzling, adaptive pipeline depth, thread parallelism optimization, and K_{split} parallelism—our approach achieves efficient and highly adaptive operator selection across a wide range of matrix sizes and hardware configurations.

3) *Dataset Construction and Training:* After generating a candidate configuration list, the compiler generates candidate kernels and benchmarks them on GPU hardware. The top- K performing configurations for each operator shape are collected to form the training dataset.

DyGen then learns a mapping from operator shape features to scheduling parameters. We adopt random forests as the base learner due to their nonlinear fitting ability and minimal feature engineering requirements. To enable constant-time inference, predictions are approximated by polynomial regression, yielding compact closed-form expressions (Fig. 3).

At inference time, given (M, N, K) , these functions directly compute parameters such as BLOCK_M/N/K, num_warps, and num_stages. Although random forests are used here, other learners (e.g., GBDT, neural networks) can also be adopted, as long as their outputs are approximated offline to obtain deployable closed-form functions.

B. Online Stage: Dynamic Inference

During online inference, when the system encounters a new tensor shape, it is fed into the generation model. The model, taking into account the operator’s dimensional features and hardware constraints, outputs the corresponding scheduling parameters in constant time. The compiler then uses these parameters to generate the operator kernel on-the-fly, which can be efficiently executed on the target hardware.

In summary, the online stage leverages a **Model Prediction + Just-In-Time Compilation** workflow, effectively avoiding costly online search and tuning, and enabling low-overhead, high-performance operator generation in dynamic inference scenarios.

IV. EVALUATION

A. Experimental Setting

a) Infrastructure: DyGen was evaluated on two NVIDIA hardware platforms running CentOS 7: H100 GPU and 4090 GPU with CUTLASS (v4.1.0), CUDA Toolkit (v12.1), Triton(v3.0.0), PyTorch(v2.4.1) [22] and vLLM(v0.10.2) [23].

b) Baseline: We compare DyGen against three state-of-the-art baselines: (1) Hand-tuned vendor libraries: cuBLAS and CUTLASS, which provide highly optimized implementations for fixed shapes; (2) General-purpose tensor compilers: Triton and PyTorch, which support automatic kernel generation for arbitrary workloads; (3) Dynamic code generation frameworks: MikPoly, representing prior efforts in runtime kernel specialization. To ensure fairness, all experiments include GPU warm-up, and execution times are averaged over 1,000 runs.

c) Metrics: We report results along three dimensions: (1) Operator-level performance – kernel execution time and speedup over baselines for GEMM and other representative operators; (2) Compiler overhead – size of the configuration search space and dataset construction cost compared to traditional tensor compilers; (3) End-to-end inference – overall latency and throughput on LLMs workloads.

d) Applications: We evaluate DyGen on a broad set of dynamic-shape workloads. First, Table III summarizes the GEMM benchmarks, including 386 cases from DeepBench [24] and 686 realistic test cases extracted from representative neural networks. These cover both Transformer-based models (e.g., Llama [25], Qwen [26], BERT [27], DistilBERT [28], RoBERTa [29], ALBERT [30]) and CNNs (e.g., AlexNet [31], ResNet [32]), ensuring coverage of diverse operator dimensions. Next, we perform end-to-end evaluation on the BERT-base model under dynamic sequence lengths with a batch size of 32. Finally, to assess scalability in real-world serving scenarios, we integrate DyGen into the vLLM framework and measure end-to-end performance under varying batch sizes and request patterns.

TABLE III
BENCHMARKED GEMM WITH DYNAMIC SHAPES.

Category	M	N	K	Test Cases
DeepBench	[7,10752]	[1,28000]	[1,4096]	386
Realistic Neural Network Matrices	[1,65536]	[1,65536]	[1,65536]	686

B. Performance Results

1) Performance Optimization of Dynamic-Shape Operators: Fig. 4 presents the performance comparison between DyGen-generated kernels and various compilers and operator libraries. The x-axis denotes the parameter size of each operator, while the y-axis represents the speedup relative to a baseline. DyGen is able to quickly generate efficient kernels on top of existing compilers, effectively optimizing dynamic-shape operators.

- (a) Compares DyGen’s kernel acceleration with Triton on the RTX 4090 and H100 GPUs. On the RTX 4090, DyGen

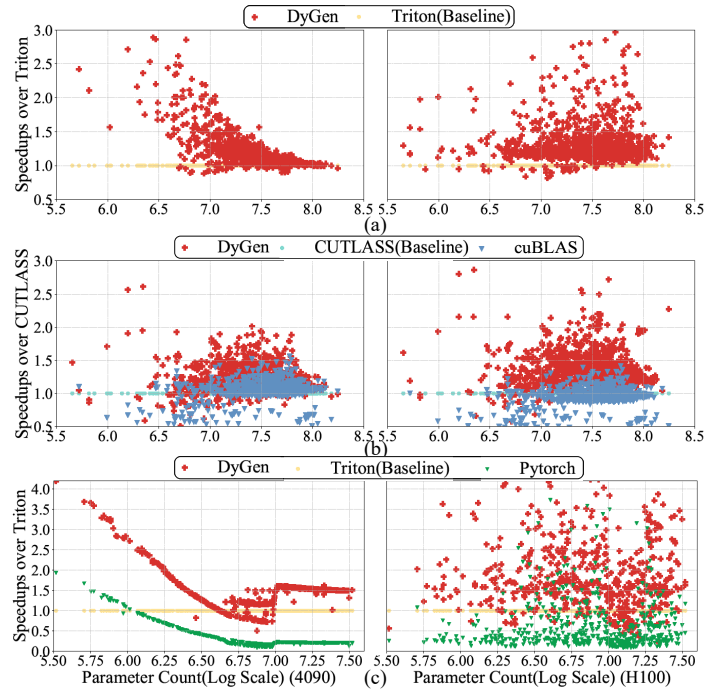


Fig. 4. Performance Comparison of DyGen-Generated Kernels Against Compilers and Libraries.

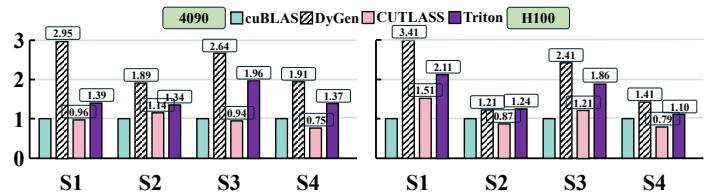


Fig. 5. DyGen Speedup Across Various Matrix Sizes in Transformer Workloads.

achieves an average GEMM speedup of $1.35\times$, with a maximum of $7.46\times$. On the H100, the average speedup is $1.36\times$, with a maximum of $8.39\times$.

- (b) Compares DyGen with the cuBLAS and CUTLASS libraries. On the RTX 4090, DyGen achieves an average GEMM speedup of $1.22\times$ over cuBLAS (up to $2.92\times$) and $1.51\times$ over CUTLASS (up to $4.17\times$). On the H100, the average speedup reaches $1.41\times$ over cuBLAS (up to $3.61\times$) and $1.62\times$ over CUTLASS (up to $5.29\times$).
- (c) Demonstrates the applicability of DyGen to other types of operators, using the NonZero operator as an example. Execution time is compared with Triton and PyTorch. Notably, for small-dimensional inputs, the NonZero operator behaves as a reductive operator, while for larger dimensions it exhibits characteristics of a cyclic-reductive operator. On the RTX 4090, DyGen achieves an average speedup of $1.46\times$ over Triton (up to $4.18\times$) and $5.79\times$ over PyTorch (up to $14.34\times$). On the H100, the average speedup reaches $1.87\times$ over Triton (up to $4.26\times$) and $5.16\times$ over PyTorch (up to $9.70\times$).

Figure 5 demonstrates the speedup of DyGen for four representative matrix sizes in Transformer workloads, highlighting

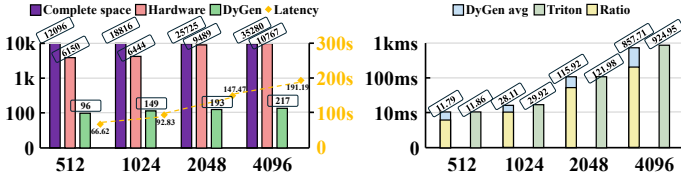


Fig. 6. Effectiveness of DyGen’s Filtering Rules in Reducing Search Space and Dataset Construction Time.

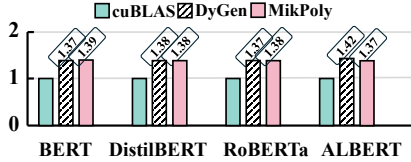


Fig. 7. End-to-End Evaluation of DyGen on BERT-family Models with Dynamic Sequence Lengths.

its efficiency across different scenarios. Additionally, we have recorded the compilation time overheads for both DyGen and Triton optimization strategies. Under Triton’s autotuning strategy, the average time for compiling 18 times is 8,977 seconds, while the heuristic strategy requires an average of 243 compilations, taking approximately 4 minutes. In contrast, DyGen only requires a single compilation, with an average time of 0.312 seconds, and the compilation time remains consistently close to the mean across runs.

Overall, these results indicate that DyGen can effectively optimize dynamic-shape operators, achieving significant performance improvements across different hardware platforms and operator types.

2) *Effectiveness of the Filtering Rules:* Fig. 6 (left chart) compares the configuration search space and dataset construction time for GEMM operators with different dimensions (M , K , N) under various filtering strategies. Exhaustive search generates tens of thousands of candidate configurations, resulting in high dataset construction costs. Hardware-aware filtering reduces this to a few thousand, but it’s still impractical. In contrast, DyGen’s rule-based filtering reduces the candidate space by nearly two orders of magnitude (96–217 configurations), cutting construction time from hundreds of seconds to under 200, while maintaining model training diversity. Using DyGen’s filtering rules, we built datasets on an RTX 4090 by sweeping M , K , and N from 512 to 8192 in 512-step increments, completing the entire process, including training, in under a day. These results highlight DyGen’s efficiency in accelerating dataset construction.

The right chart further illustrates the performance advantage of DyGen. We executed all configurations filtered by DyGen and calculated their average execution time, which was then compared against the fastest execution time achieved by Triton. The results show that DyGen’s average execution time is lower than Triton’s best time. Moreover, in over 80% of cases, DyGen-generated configurations outperform Triton’s fastest configuration. This further confirms the effectiveness of DyGen in eliminating redundant configurations and delivering high-performance kernels.

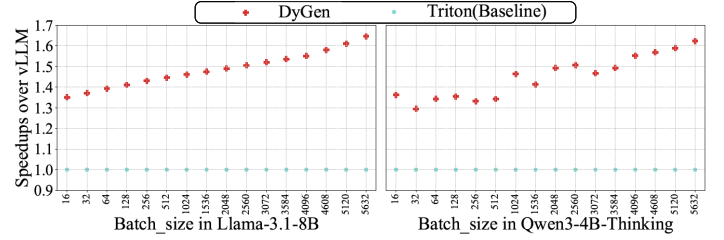


Fig. 8. DyGen Performance in Large-Scale LLM Serving: Speedup in vLLM Framework with Dynamic Batching.

3) *End-to-End Model Evaluation:* To further verify the practicality of DyGen, we conduct end-to-end evaluations on both NLP inference tasks and LLM serving scenarios.

We first evaluate DyGen on BERT-family models (BERT, DistilBERT, RoBERTa, ALBERT) with dynamic sequence lengths, fixing the batch size at 32. For comparison, we use cuBLAS as a benchmark (as it outperforms CUTLASS in most cases). As shown in Fig. 7, DyGen consistently delivers higher runtime performance, achieving average speedups of 1.37×, 1.38×, 1.37×, and 1.42× across the four tasks, thereby significantly reducing end-to-end inference latency. Compared to the state-of-the-art MikPoly (1.39×, 1.38×, 1.36×, and 1.37× speedup over cuBLAS, as reported in its paper), DyGen achieves comparable or even slightly better performance.

To assess DyGen in LLMs online serving scenarios, we integrate Triton into the vLLM framework, a popular and efficient LLM inference system with dynamic batching support. As shown in Fig. 8, we fix the sequence length to 10 and measure latency across different batch sizes (ranging from 16 to 5632). On LLaMA-3.1-8B and Qwen-3-4B, DyGen achieves average speedups of 1.37× and 1.42×, respectively.

DyGen demonstrates effectiveness in both micro-benchmarks and real-world serving, delivering system-level performance gains. Its constant-time kernel generation and hardware-aware optimizations provide a scalable solution for large-scale inference, particularly as models grow. Unlike MikPoly’s dynamic microkernel composition that incurs overhead, DyGen maintains consistent performance under variable workloads. As a compiler plug-in, it integrates seamlessly with emerging compilers, benefiting from future advancements.

V. CONCLUSION

In this paper, we introduced DyGen, a dynamic-shape kernel generator compiler plug-in that addresses the challenge of efficiently optimizing neural network operators with varying shapes. DyGen leverages an innovative two-stage approach, combining offline kernel generation with real-time dynamic inference, significantly reducing the overhead of traditional compilers. Through a hardware-aware model that predicts optimal kernel configurations in constant time, DyGen achieves significant performance improvements across various tasks tested on the GPU. Our experiments show that DyGen outperforms state-of-the-art compilers and libraries, demonstrating its effectiveness in both operator optimization and real-world applications. Future work will focus on extending DyGen’s capabilities for even more diverse operator types and further improving its integration with emerging hardware accelerators.

REFERENCES

- [1] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf
- [2] Y. Mansour, A. Kaissar, and S. Ansari, "Review on recent matrix multiplication optimization using deep learning," in *International Conference on Intelligent and Fuzzy Systems*. Springer, 2024, pp. 359–371.
- [3] K. Lu, Y. Wang, Y. Guo, and et al., "Mt-3000: A heterogeneous multizone processor for hpc," *CCF Transactions on High Performance Computing*, 2022.
- [4] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 3, p. 17, 2017.
- [5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE micro*, vol. 28, no. 4, pp. 13–27, 2008.
- [6] oneDNN Contributors, "oneAPI Deep Neural Network Library (oneDNN)." [Online]. Available: <https://github.com/uxlfoundation/oneDNN>
- [7] NVIDIA, "cublas: Nvidia cuda basic linear algebra subprograms library," 2023, accessed: 2025-08-20. [Online]. Available: <https://developer.nvidia.com/cublas>
- [8] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS," Jan. 2023. [Online]. Available: <https://github.com/NVIDIA/cutlass>
- [9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. 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)*, 2018, pp. 578–594.
- [10] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.
- [11] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," 2021. [Online]. Available: <https://arxiv.org/abs/2102.04906>
- [12] Y. Zheng, L. Yi, and Z. Wei, "A survey of dynamic graph neural networks," 2024. [Online]. Available: <https://arxiv.org/abs/2404.18211>
- [13] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/yu>
- [14] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," 2023. [Online]. Available: <https://arxiv.org/abs/2309.06180>
- [15] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2305.05920>
- [16] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: <http://arxiv.org/abs/1504.08083>
- [17] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.
- [18] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.
- [19] H. Shen, J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, and Y. Wang, "Nimble: Efficiently compiling dynamic neural networks for model inference," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 208–222, 2021.
- [20] B. Zheng, Z. Jiang, C. H. Yu, H. Shen, J. Fromm, Y. Liu, Y. Wang, L. Ceze, T. Chen, and G. Pekhimenko, "Dietcode: Automatic optimization for dynamic tensor programs," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 848–863, 2022.
- [21] F. Yu, G. Li, J. Zhao, H. Cui, X. Feng, and J. Xue, "Optimizing dynamic-shape neural networks on accelerators via on-the-fly micro-kernel polymerization," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 797–812.
- [22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: <https://arxiv.org/abs/1912.01703>
- [23] vLLM Project, "vllm: A high-performance language model serving framework," 2023, accessed: 2025-08-15. [Online]. Available: <https://github.com/vllm-project/vllm>
- [24] B. Research, "Deepbench: Benchmarking deep learning performance," 2023, accessed: 2025-08-20. [Online]. Available: <https://github.com/baidu-research/DeepBench>
- [25] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: <https://arxiv.org/abs/2302.13971>
- [26] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu, "Qwen technical report," 2023. [Online]. Available: <https://arxiv.org/abs/2309.16609>
- [27] M. V. Koroteev, "Bert: a review of applications in natural language processing and understanding," *arXiv preprint arXiv:2103.11943*, 2021.
- [28] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.
- [29] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
- [30] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," 2020. [Online]. Available: <https://arxiv.org/abs/1909.11942>
- [31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.
- [32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: <https://arxiv.org/abs/1512.03385>