A Sample-Free Compilation Framework for Efficient Dynamic Tensor Computation

Yangjie Zhou
Tencent
Shenzhen, China
National University of Singapore
Singapore, Singapore
yjzhou96@gmail.com

Honglin Zhu
Tencent
Shenzhen, China
honglinzhu@tencent.com

Qian Qiu
Tencent
Shenzhen, China
qianqiu@tencent.com

Weihao Cui
Shanghai Jiao Tong University
Shanghai, China
weihao@sjtu.edu.cn

Zihan Liu
Shanghai Jiao Tong University
Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, China
altair.liu@sjtu.edu.cn

Peng Chen
RIKEN Center for Computational Science
Kobe, Japan
peng.chen@riken.jp

Mohamed Wahib
RIKEN Center for Computational Science
Kobe, Japan
mohamed.attia@riken.jp

Cong Guo
Shanghai Jiao Tong University
Shanghai, China
guocong@sjtu.edu.cn

Siyuan Feng
Shanghai Jiao Tong University
Shanghai, China
hzfengsy@sjtu.edu.cn

Jintao Meng
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
Shenzhen, China
jt.meng@siat.ac.cn

Haidong Lan
Taichi Graphics
Shenzhen, China
haidonglan@taichi.graphics

Jingwen Leng
Shanghai Jiao Tong University
Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, China
leng-jw@cs.sjtu.edu.cn

Yun Lin
Shanghai Jiao Tong University
Shanghai, China
lin_yun@sjtu.edu.cn

Jin Song Dong
National University of Singapore
Singapore, Singapore
dcsdjs@nus.edu.sg

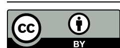
Wenxi Zhu
Tencent
Shenzhen, China
wenxizhu@tencent.com

Minwen Deng
Tencent
Shenzhen, China
danierdeng@tencent.com

Abstract

Dynamic-shape tensor computation poses challenges for shape-specific compilation due to variable input dimensions. Existing compilers rely on shape samples, incurring high tuning costs and performance degradation on unseen inputs. We present `Helix`, a dynamic tensor compilation framework with sample-free compilation and architecture-guided optimization to achieve both compilation efficiency and shape-general performance. To avoid shape sampling, `Helix` constructs shape-agnostic compilation by decomposing computations across architectural layers. A bidirectional strategy combines top-down abstraction to align tensor computations with architectural hierarchies, and bottom-up kernel construction to build efficient execution strategies from reusable, architecture-aligned micro-kernels. A hybrid analyzer ensures accuracy through profiling at lower architectural levels, and achieves scalability through architecture-informed modeling at higher levels and runtime. This hierarchical design eliminates shape-specific tuning and enables

Yangjie Zhou and Honglin Zhu contributed equally to this work. Wenxi Zhu and Minwen Deng are the corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759779

shape-adaptive execution. Evaluations conducted on x86 CPUs, ARM CPUs, and NVIDIA GPUs demonstrate that Helix reduces compilation time by 174× over the existing compilers and delivers 2.26× and 3.29× execution speedups over vendor libraries and dynamic-shape compilers, respectively.

CCS Concepts

• **Computing methodologies → Parallel computing methodologies**; • **Software and its engineering → Dynamic compilers**.

Keywords

Dynamic Tensor Computation, Deep Neural Networks, Compilation Optimization

ACM Reference Format:

Yangjie Zhou, Honglin Zhu, Qian Qiu, Weihao Cui, Zihan Liu, Peng Chen, Mohamed Wahib, Cong Guo, Siyuan Feng, Jintao Meng, Haidong Lan, Jingwen Leng, Yun Lin, Jin Song Dong, Wenxi Zhu, Minwen Deng. 2025. A Sample-Free Compilation Framework for Efficient Dynamic Tensor Computation. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3712285.3759779

1 Introduction

Efficient optimization of tensor programs is crucial in accelerating Deep Neural Networks (DNNs) [27] and large language models (LLMs) [61]. Traditional compilers [7, 16, 38, 63] have primarily focused on optimizing **static-shape** DNNs, where tensor computations involve fixed-shape inputs and outputs at runtime. Recently, the emergence of **dynamic-shape** DNNs, capable of handling variable input shapes during runtime, has introduced new challenges [21]. For example, Transformer-based models [49] often adopt variable sequence lengths, necessitating dynamic-shape tensor computation. As such, efficiently managing these variations is crucial for maintaining performance.

The core challenge in optimizing dynamic-shape tensor programs is to achieve efficient offline code generation while ensuring high-performance execution at runtime [21]. A practical solution must construct a shape-generalized execution strategy within a tractable compilation budget, as the vast shape space makes exhaustive enumeration computationally prohibitive. At runtime, it must select and invoke an efficient execution strategy with minimal dispatch overhead, ensuring high performance even for input shapes unseen during offline compilation while avoiding costly online re-optimization. Bridging this gap between offline code generation and runtime execution is fundamental to enabling scalable and high-performance dynamic-shape tensor computation.

Existing approaches to optimizing dynamic-shape tensor programs fall into two main categories: vendor-provided libraries and sample-driven compilers. Vendor libraries, such as oneDNN [1] for CPUs [11] and cuBLAS [33] for GPUs [32], rely on manually optimized implementations tailored for specific tensor operations. While these solutions can deliver high performance for specific workloads, they lack flexibility and fail to generalize across diverse and unpredictable input shapes. Sample-driven compilers [62, 66] aim to improve adaptability by optimizing kernels for a predefined set of shape samples. However, this reliance on a fixed sample set inherently limits generalization, causing performance to degrade significantly for shapes not included in the sample set, with slowdowns of up to 4× (§2.2). Furthermore, covering a wide shape space incurs prohibitive compilation costs, making these methods impractical in settings with highly variable input distributions.

Table 1: Comparison of existing solutions with Helix.

Classification	Sample-Free	Codegen Method	Tuning Overhead	Performance Generalization
Vendor Libraries [1, 3, 9, 14, 33, 45]	✓	Hand-Tuned	N/A (Unsupported)	Labor-Constrained
Sample-driven Compilers [62, 66]	✗	Sample-Based	High	Sample-Bound
Helix (Ours)	✓	Arch-Guided	Low	Shape-Generalizable

We present Helix, an architecture-guided and sample-free dynamic tensor compiler that addresses the dual challenges of compilation efficiency and execution adaptability through bidirectional compilation. Helix combines hierarchical abstraction, multi-level kernel construction, and a hybrid performance analyzer into a recursive workflow that reduces offline overhead and enables efficient, shape-adaptive execution.

To improve compilation efficiency by reducing the search complexity, Helix transforms kernel generation into a structured and architecture-aligned recursive process. It begins with a top-down abstraction that decomposes tensor programs based on the memory hierarchy of modern computing architectures [11, 32, 35], where computations are structured across off-chip memory (e.g., DRAM), on-chip shared storage (e.g., L1 cache or shared memory), and register-level execution resources. This decomposition reduces the complexity of the original problem by partitioning the global search space into tractable layer-wise subproblems. At each layer, Helix uses a bottom-up process to construct candidate kernels under local constraints, including register usage, thread configuration, and memory capacity. These constraints define tight feasibility bounds that enable pruning and eliminate the need for costly global tuning. By decoupling kernel construction from runtime shapes, Helix avoids combinatorial explosion and ensures efficient and generalizable compilation across diverse tensor programs.

To improve execution performance while retaining generality, Helix constructs micro-kernels specifically tailored to the architectural hierarchy and adopts a hybrid performance analyzer to guide efficient tiling and kernel construction across layers. At each hierarchy level, Helix constructs a set of low-level micro-kernels, each tailored to minimize padding overhead, enhance data locality, and generalize across varying runtime shapes. These micro-kernels serve as composable building blocks for recursively assembling full execution strategies. Kernel selection at each level is guided by a hybrid analyzer that balances accuracy and cost. It applies profiling at low levels where candidate sets are small, and uses analytical modeling at higher levels for scalability. The analyzer recursively models performance across layers by combining current-level hardware features, such as memory bandwidth, parallelism, and latency, with predicted or profiled results from lower levels. This layered and recursive modeling strategy enables efficient micro-kernel selection throughout compilation and at runtime.

Table 1 summarizes the core differences between Helix and existing dynamic-shape solutions. While vendor libraries rely on

hand-tuned implementations with limited adaptability, and sample-driven compilers require shape-specific tuning to maintain performance, Helix introduces an architecture-guided compilation workflow that eliminates sample dependency, minimizes tuning overhead, and delivers robust performance across diverse shapes. Our comprehensive evaluation includes comparisons with state-of-the-art vendor libraries [1, 3, 9, 14, 33, 45] and the dynamic-shape compiler DietCode [62] across Intel x86 CPUs [11], ARM CPUs [35], and Nvidia GPUs [32]. Helix demonstrates remarkable speed improvements, achieving **2.26×** and **3.29×** faster performance than vendor libraries and DietCode, respectively. Additionally, Helix reduces offline compilation time by an astounding **174×** compared to DietCode. These results underscore Helix's ability to outperform existing solutions across both CPU and GPU platforms.

This paper makes the following contributions:

- We identify the limitations of existing sample-driven dynamic-shape compilers and highlight architecture-guided compilation as a path to eliminate shape-specific tuning while improving both compilation efficiency and execution adaptability.
- We propose a bidirectional compilation framework, integrating top-down hierarchical decomposition with bottom-up, architecture-guided kernel construction. This approach reduces compilation overhead while maintaining adaptability across workloads.
- We introduce a hybrid performance analyzer that recursively combines profiling and modeling to estimate layer-wise kernel performance. It enables accurate cost estimation and kernel selection across architectural levels, supporting both offline pruning and runtime dispatch without online tuning.
- We implement Helix and evaluate it on Intel x86 CPU, Arm CPU, and Nvidia CUDA GPU. Our evaluation confirms a 2~3× speedup on performance with two orders of magnitude less compilation time. It demonstrates Helix's superiority over existing dynamic-shape compilers and manual optimization techniques.

The rest of this paper is organized as follows: Section 2 discusses the background and motivation. Section 3 outlines the proposed framework. Section 4 presents the design strategy. Section 6 describes the implementation. Section 7 reports the evaluation results. Section 8 reviews related work. Finally, Section 9 concludes.

2 Background and Motivation

This section begins with a brief overview of dynamic-shape tensor programs and their practical use cases. We then highlight the limitations of sample-driven optimization and conclude by discussing the opportunities and challenges of architecture-guided approaches.

2.1 Dynamic-Shape Tensor Program

Tensor programs serve as operator-level abstractions widely used in neural network computation [2, 34]. While conventional tensor programs rely on static shape information [7, 16, 63], dynamic-shape tensor programs allow for processing with offline unknown shapes [21], increasingly necessary due to two key factors: *intrinsic data variability* and *system-level execution dynamics*.

Intrinsic data format dynamism arises naturally in many domains. For example, in natural language processing (NLP) [15, 37, 47], varying sequence lengths challenge traditional static-shape assumptions. In computer vision (CV) [18, 22, 58], dynamic image sizes are used for detection and tracking tasks. Similarly, GNN workloads [52, 54, 68] involve graphs with differing numbers of vertices and edges. These cases highlight the broad relevance of dynamic-shape support.

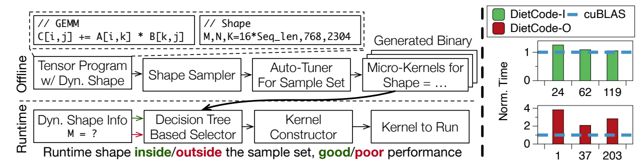


Figure 1: (a) The sample-driven compilation workflow. (b) Comparison of DietCode and cuBLAS across different sequence lengths on A100 GPU. "DietCode-I" and "DietCode-O" refer to input configurations within and outside the tuning sample list, respectively.

System execution and scheduling strategies further underscore the importance of dynamic-shape tensor programs. In real-world scenarios, dynamic adjustments to batch sizes are often required due to varying inputs [10, 13, 60] and optimization techniques such as early-exit [4, 39, 56]. This variability necessitates adaptability in the underlying tensor program.

2.2 Limitations of Sample-Driven Approach

Static-shape compilers [7, 16, 63, 70] are not well-suited for dynamic-shape tensor programs because they assume a fixed, one-to-one correspondence between the generated kernel and shape. This static assumption leads to significant overheads in dynamic scenarios, where shapes are not predetermined and can vary considerably.

Existing optimizations for dynamic-shape tensor programs predominantly follow a sample-driven approach [41, 62, 66], as shown in Figure 1a. In this method, a predefined sample list is used to represent dynamic-shape parameters. This list is combined with the static shape and tensor program to create a shape-generic search space. An auto-tuning module, adapted from static-shape compilers [63], explores this search space, generating optimized micro-kernels for each sample. At runtime, a decision-tree-based selector chooses the appropriate micro-kernel based on the current shape, while the kernel constructor finalizes the process by setting launch parameters and applying padding when necessary to handle unseen shapes not included in the sample list.

However, this approach is significantly limited by its reliance on a predetermined sample list for dynamic-shape parameters, which restricts the compiler's flexibility and overall effectiveness. This shortfall is critical as real-world data often exhibit greater diversity than what predefined samples can address. To empirically validate this limitation, we conduct an experiment targeting the first general matrix multiply (GEMM) operation of the Bert model [15]. In this context, M represents the number of rows in matrix A (the product of batch size and sequence length), N represents the number of columns in matrix B (fixed at 768), and K is the number of columns in matrix A (and rows in matrix B, fixed at 2304). Using DietCode's default sample configuration, we test with a fixed batch size of 16 and sequence lengths varying from 5 to 128 in steps of 19.

As shown in Figure 1b, DietCode exhibits more severe performance degradation compared to cuBLAS [33] when encountering input shapes not included in its sample list. This is due to the absence of specifically optimized micro-kernels for these shapes and

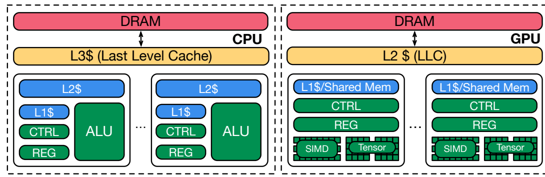


Figure 2: CPU/GPU Diagram.

the increased inefficiency from padding loss. Moreover, modifying the sample list to account for different scenarios necessitates time-consuming re-tuning, further highlighting its inflexibility.

2.3 Architecture-Guided Opportunities

The mismatch between runtime and offline-sampled shapes limits the efficiency of traditional sample-driven methodologies in dynamic-shape compilation, resulting in suboptimal performance and high tuning overhead. More fundamentally, these methods treat the hardware as a black box, relying solely on empirical performance feedback while ignoring the wealth of prior knowledge encoded in the architecture itself.

Figure 2 illustrates the hierarchical structure of mainstream architectures like CPUs [11, 35] and GPUs [32] There exist inherent similarities among these architectures, each of which has a hierarchical structure. This hierarchy is distinctly multi-level, wherein each level comprises a predetermined quantity of computational or storage units. For instance, each CPU Core has its own L1 cache, L2 cache, and arithmetic logic units (ALUs), while all the CPU Cores share the L3 cache and DRAM. As for GPUs, each streaming multiprocessor (SM) has its own computing units (CUDA Cores, Tensor Cores) and L1 Cache, while all SMs share the L2 cache and DRAM. These levels impose resource constraints that directly affect the execution behavior of tensor computation.

To quantify the impact of architectural limits, we collected different configurations for matrix multiplication during Ansor's tuning process [63]. As shown in Figure 3, performance, measured in Floating Point Operations Per Second (FLOPS), drops sharply when resource usage exceeds architectural limits. This consistent trend indicates that architecture-unfriendly configurations are inherently inefficient. Leveraging this insight, we can preemptively prune invalid or suboptimal configurations from the strategy space, eliminating the need for a predefined sample list. Furthermore, the architectural hierarchy provides systematic and structural guidance for pruning at each level.

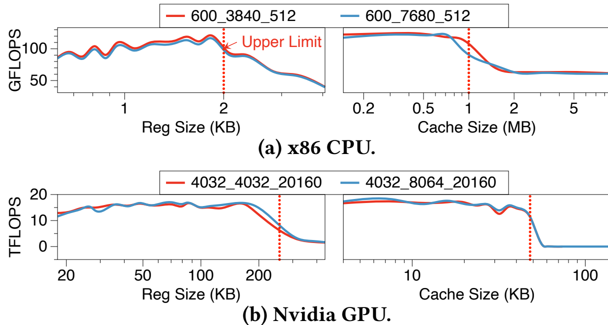


Figure 3: GEMM performance across different hardware resource usages on 8255c CPU and A100 GPU. Legend indicates corresponding GEMM parameters M, N, and K.

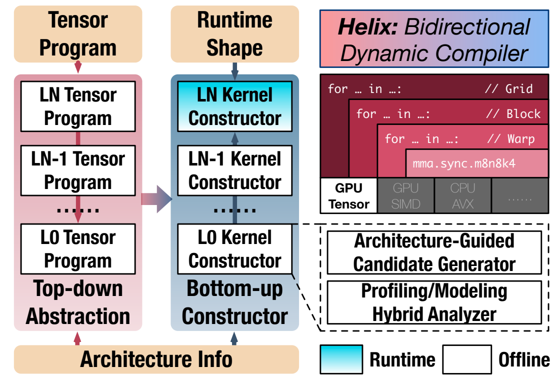


Figure 4: Design overview of Helix.

In summary, architectural hierarchy provides structured guidance for reducing the compilation search space and constructing valid and efficient strategies. This motivated us to design an architecture-guided compilation framework that facilitates highly efficient computational performance while maintaining flexibility.

3 Overview of Helix

Helix is a compilation framework designed to optimize dynamic-shape tensor programs by leveraging architecture-specific information through a bidirectional compilation process that combines top-down abstraction with bottom-up kernel construction. This design enables Helix to efficiently generate kernels that generalize across dynamic input shapes without relying on runtime shape sampling. Figure 4 illustrates the overview of Helix, covering both the offline compilation and runtime adaptation stages.

In the offline stage, Helix first applies a top-down abstraction, systematically decomposing tensor programs into hierarchical levels aligned with the underlying architectural hierarchy. Each level partitions the tensor computation into sub-programs that correspond to architectural components. By incorporating architecture-specific knowledge during this phase, Helix tailors the strategy space to align with the target architecture, ensuring efficient alignment between software and hardware.

Next, Helix refines the abstracted tensor programs into optimized kernels through a bottom-up construction process. A multi-level kernel constructor generates candidate kernels at each hierarchy level, tailoring them to architecture-specific execution units and memory structures. Candidate evaluation is guided by a hybrid analyzer that combines profiling at lower levels, where candidate sets are small and performance characteristics are critical, with predictive modeling at higher levels to maintain scalability. This layered construction process incrementally builds efficient execution strategies while avoiding exhaustive global search.

During the runtime stage, Helix dynamically adapts its pre-compiled micro-kernels based on the actual input shapes. Using the modeling analyzer, the last-level kernel constructor identifies the best-fit configuration, ensuring high-performance execution without incurring runtime compilation overhead.

Together, by systematically embedding architecture-specific knowledge into both offline and runtime stages, Helix consistently delivers robust performance across diverse architectural platforms

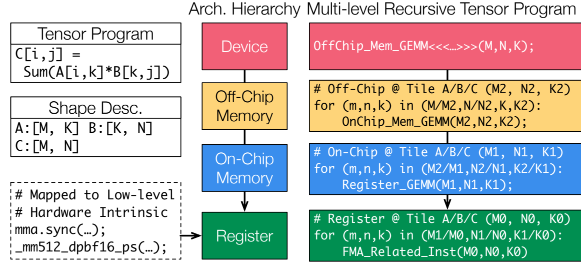


Figure 5: Recursive execution pattern of the *GEMM* operator across architectural hierarchy levels.

Algorithm 1 Unified Recursive Abstraction.

1: **procedure** rKernel(L, PL, TSL, TRL)
2: *// L: Current hierarchical layer*
3: *// PL: Set of parallel loops*
4: *// TRL: Set of temporal reduction loops*
5: *// TNL: Set of temporal non-reduction loops*
6: **for** each parallel loop p in $PL[L]$ **do**
7: **for** each temporal non-reduction loop ts in $TNL[L]$ **do**
8: **for** each temporal reduction loop tr in $TRL[L]$ **do**
9: Load_Func(L, p, ts, tr)
10: rKernel($L-1, PL, TNL, TRL$)
11: Store_Func(L, p, ts)

and dynamic workloads, achieving high execution efficiency while minimizing offline compilation and runtime adaptation overhead.

4 Strategy Space Hierarchization

This section introduces the key of `Helix`'s workflow: strategy space hierarchization via top-down recursive decomposition. Using *GEMM* as a case study, we illustrate the recursive patterns that underlie tensor programs and introduce our unified abstraction, *rKernel*, which generalizes these structures across diverse architectures.

4.1 Top-Down Recursive Notation

We use the example of *GEMM* tensor program to illustrate recursive notation for the tensor program top-down decomposition. *GEMM*, as a classic operator in deep learning, is mathematically defined as $C = A \times B$, where A and B are input matrices and C is output matrix. As shown in Figure 5, the execution of *GEMM* can naturally be represented by recursively decomposing the tensor program into nested subproblems, forming a hierarchy of multi-level loops. Each subproblem corresponds to a distinct architectural level, progressively refining the computation from off-chip memory down to on-chip memory and registers. Specifically, the uppermost level, `OffChip_Mem_GEMM`, manages complete matrix computation in off-chip memory. At the intermediate level, `OnChip_Mem_GEMM` handles smaller matrix blocks in on-chip memory. The innermost recursion, `Register_GEMM`, focuses on individual elements in registers, utilizing specific calculation instructions.

This top-down recursive approach is intuitive and crucial for optimizing tensor programs on modern computing architectures. It decomposes the computation into subproblems and maps them onto distinct architectural levels. This mapping enables each subproblem to be specifically optimized for the memory, compute, and storage characteristics of its corresponding level, providing a clear and structured pathway for independently improving data locality, parallelism, and overall execution efficiency across the hierarchy.

4.2 Unified Abstraction Design

Recognizing the hierarchical nature of tensor program execution, we also acknowledge the variations between different architectures and tensor programs. Specifically, CPUs and GPUs exhibit distinct computation modes and memory access controls, where CPUs are optimized for multi-threaded parallelism, and GPUs excel in Warp, CTA, and Grid-level parallelism [6]. Moreover, different tensor programs, such as *Convolution* and *GEMM*, demonstrate unique loop patterns and execution characteristics. These variations drive the design of our unified abstraction, which provides a universal framework for representing tensor program execution across diverse computing architectures.

Algorithm 1 elaborates on this abstraction. It maintains the layer-wise recursive structure as the core, and enables custom loop mapping and execution stages for various tensor programs. At each hierarchical level, loops are categorized into three sets. The *Parallel Loop Set* manages parallel operations, enabling simultaneous execution across available hardware resources. The *Temporal Loop Sets* handle sequential operations and are divided into two subsets: the *Temporal Reduction Loop Set*, which processes operations involving reduction (e.g., summing values), and the *Temporal Non-reduction Loop Set*, which deals with operations that do not involve reduction. Each level, denoted as level N, abstracts the execution into three stages: *Load*, *rKernel(N-1)*, and *Store*. These stages serve as flexible interfaces, allowing the execution strategy at each level to be tailored according to specific computational requirements.

Table 2 illustrates how this unified abstraction rKernel is implemented across different architectural configurations, highlighting its versatility. Our focus lies in scrutinizing recursive execution patterns among different hierarchies. For CPUs [11, 35], at the lowest level (L0), *rKernel* enables direct data transfer from Global memory or CacheBuffer to Registers, depending on specific computational needs. A "CacheBuffer" is defined as a memory buffer, sized within the L2 cache limits, ensuring efficient data caching and access [1]. Additionally, store operations at this level also reflect this adaptability, offering the choice of transferring data back to Global memory or CacheBuffer. As we progress to level L1, the abstraction provides options for either transferring data from Global memory to CacheBuffer or performing no operation, signifying a versatile approach to data handling. The highest level (L2) in CPUs focuses on

Table 2: Complete representation for different architectures and levels via *rKernel* abstraction. '-' refers to 'No Parallel Binding' in the 'Parallel Binding' column and 'No Operation' elsewhere.

Arch.	Level	Parallel Binding	Load	Lower Level rKernel	Store
CPU	0	-	CacheBuf → Reg **or** GlobalMem→ Reg	ALU Calc.	Reg → CacheBuf **or** Reg → GlobalMem
	1	Thread	GlobalMem → CacheBuf **or** -	L1 rKernel	CacheBuf → GlobalMem **or** -
	2	Process	-	L2 rKernel	-
GPU	0	Warp	SharedMem→ Reg **or** GlobalMem→ Reg	Cuda/Tensor Core Calc.	Reg →SharedMem **or** Reg → GlobalMem
	1	CTA	GlobalMem → SharedMem **or** -	L0 rKernel	SharedMem → Global **or** -
	2	Grid	-	L1 rKernel	-

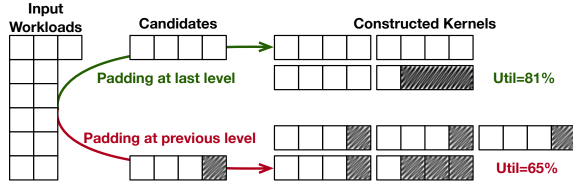


Figure 6: Comparison of different padding strategies and their impact on resource utilization efficiency.

the multi-thread mechanism at the process level, capitalizing on the CPU's capabilities for multi-core parallel processing [19].

For GPUs [32], including both CUDA Cores and Tensor Cores, *rKernel* adapts to different operational requirements. At L0, there's an option for loading data either from Global memory or Shared memory to Registers, and similarly, storing data either back to Global memory or to Shared Memory. At the L1 level, similar to CPUs, the abstraction can facilitate data transfer from Global to Shared memory or no operation, enabling efficient resource management. The L2 captures Grid-level operations, enhancing the scalability across the GPU's multiple SM architecture. *rKernel* achieves a hierarchical abstraction of execution patterns that apply universally to various tensor programs and hardware. This approach ensures a tailored strategy space for each architectural hierarchy level and facilitates universal optimizations for dynamic-shape tensor programs.

5 Architecture-Guided Generation

Building on the hierarchical strategy space defined in §4, this section presents the architecture-guided process for kernel generation in `Helix`. We describe a bottom-up construction workflow that incrementally assembles kernels from low-level micro-kernel primitives. A hybrid performance analyzer integrates profiling and modeling to guide kernel selection at each level.

5.1 Bottom-Up Construction Workflow

`Helix` constructs optimized execution strategies through a bottom-up process aligned with the architectural hierarchy. Instead of enumerating kernel parameters across the full strategy space, it incrementally constructs candidates per layer, ensuring both architectural feasibility and cross-layer composability.

To ensure efficient kernel execution, we enforce a layer-wise alignment constraint: shapes at each upper layer must be integer multiples of those at the previous layer. As illustrated in Figure 6, misaligned shapes cause significant padding overhead, especially at higher layers. By enforcing alignment, we confine padding losses to the outermost layer and ensure efficient tiling across the hierarchy.

The candidate construction algorithm is shown in Algorithm 2. The function GenLayerCands is responsible for generating candidate configurations for a given layer L. It first retrieves architecture-specific resource constraints via GetHWInfo, and initializes a set of candidate shapes using InitLayerCands (lines 2–3).

For the base layer ($L = 0$), architecture-specific instruction-level constraints are enforced through FilterByISA (lines 5) to ensure compatibility with the platform's Instruction Set Architecture (ISA). For example, on Intel x86 [11] and ARM CPUs [35], this includes vector width imposed by AVX512 and NEON instructions; on NVIDIA GPUs [32], it accounts for MMA instruction shape restrictions. These constraints ensure compatibility with platform-specific low-level execution units.

Algorithm 2 Candidates Generation Algorithm.

```
1: function GenLayerCands(L)
2:     hwInfo ← GetHWInfo(L)
3:     cands ← InitLayerCands(hwInfo) ▷ Init. candidates under resource limits
4:     if L = 0 then
5:         cands ← FilterByISA(cands)
6:     else
7:         prevCands ← GetPrevLayerCands(L − 1)
8:         cands ← FilterByMultiples(cands, prevCands)
9:     return cands
10: end function
11: function FilterByISA(cands)            ▷ Initialize ISA-valid candidate set
12:     filtered ← ∅
13:     for cand ∈ cands do
14:         if IsCompatible(cand) then                ▷ Ensure ISA compatibility
15:             filtered.add(cand)
16:     return filtered
17: end function
18: function FilterByMultiples(cands, prevCands)     ▷ Composable with prev
19:     filtered ← ∅
20:     crossLayerMap ← an empty map                 ▷ Track composable pairs
21:     for prev ∈ prevCands do
22:         multiples ← GenMultiples(prev, cands)          ▷ Shape-aligned
23:         for multiple ∈ multiples do
24:             filtered.add(multiple)
25:             crossLayerMap[multiple].append(prev)
26:     return filtered, crossLayerMap
27: end function
```

For higher layers ($L > 0$), candidate filtering is guided by alignment with previous layer configurations. The FilterByMultiples function (lines 18–27) retains candidates that are integer multiples of previous layer shape in each dimension. This sieve-inspired filtering process [12, 20] systematically eliminates non-aligned candidates based on divisibility constraints, providing a concise and theoretically grounded method for efficient pruning. This filtering scheme enforces progressive shape alignment, ensuring structural composability and systematic micro-kernel reuse. During this step, we also construct a candidate-to-substructure crossLayerMap, capturing the compositional relationship across layers.

`Helix` incrementally constructs a compact, architecture-aligned candidate space through progressive constraint enforcement and recursive cross-layer alignment. This space incorporates cross-layer structural composability, providing the foundation for efficient and scalable kernel generation in subsequent stages.

5.2 Hybrid Performance Analyzer

Goal of the Analyzer. The analyzer in `Helix` estimates the performance of candidate strategies during both compilation and runtime. Leveraging Algorithm 2, we construct a mapping that captures the recursive relationships between candidates across adjacent levels. Each candidate may correspond to multiple lower-level configurations, requiring cost aggregation over substructures. To support this, the analyzer must efficiently assess hierarchical performance while preserving architectural alignment and cross-layer composability.

Performance Model. To efficiently estimate the cost of each candidate strategy, `Helix` adopts a recursive performance model that reflects the architectural hierarchy and layer-wise scheduling structure. As illustrated in Figure 7, the performance model is recursive,

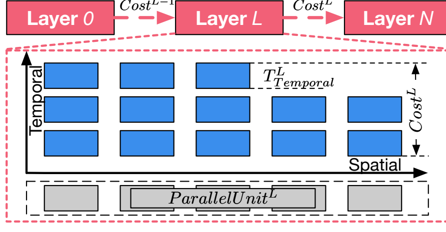


Figure 7: An illustration of execution abstraction and associated performance model.

propagating cost estimates across layers based on kernel decomposition. Within each layer, it captures temporal-spatial interactions by jointly modeling pipeline execution and parallelism patterns. Table 3 summarizes the architectural and modeling parameters used in our analysis.

We begin by modeling tiling feasibility under architectural constraints. Helix explicitly manages memory hierarchy through software tiling. For any candidate tile at level L, its size $TileSize^L$ must not exceed the available memory capacity:

$$TileSize^L \leq Capacity^L \tag{1}$$

This ensures that data fits into the available storage at the current level (e.g., shared memory or register tiles). Given the tiling behavior, the memory transfer latency is determined by both inherent memory latency and bandwidth:

$$T^L_{load/store} = T^L_{Latency} + \frac{TileSize^L}{BW^L} \tag{2}$$

Compute latency is defined recursively. At the base level ($L = 0$), it is computed by dividing the required floating point operations (FLOPs) by the peak throughput of the corresponding execution CPU Cores, Tensor Cores (TC), or CUDA Cores; for higher levels ($L > 0$), it accumulates the execution cost from the previous level:

$$T^0_{compute} = \begin{cases} \frac{FLOPs^0}{F_{CPU}}, & \text{if CPU used} \\ \frac{FLOPs^0}{F_{TC}}, & \text{if Tensor Core used} \\ \frac{FLOPs^0}{F_{CUDA}}, & \text{if only CUDA Core used without Tensor Core} \end{cases} \tag{3}$$

Table 3: Architectural and modeling parameters used in Helix's performance model.

	Definition	Symbol
Architectural	Sustained bandwidth of the memory hierarchy at level L	BW^L
	Fixed access latency for memory operations at level L	T^L_{Latency}
	Available storage capacity of the memory hierarchy at level L	$Capacity^L$
	Number of independent parallel compute units at level L	$\lvert\text{ParallelUnit}^L\rvert$
	Peak compute throughput of CPU Cores	F_{CPU}
	Peak compute throughput of GPU Tensor Cores	F_{TC}
	Peak compute throughput of GPU CUDA Cores	F_{CUDA}
	The time for launching kernels to the compute device	T_{launch}
Modeling	Tile size selected for micro-kernel execution at level L	$TileSize^L$
	Number of floating point operations at level L	$FLOPs^L$
	Number of iterations along temporal loops at level L	N^L_{TmpIter}
	Number of iterations along parallel loops at level L	N^L_{ParIter}
	Total latency for load/store data transfer at level L	$T^L_{load/store}$
	Temporal execution latency with pipelined pattern at level L	T^L_{temporal}
	Scaling factor reflecting parallelism available at level L	F^L_{spatial}
	Modeled execution latency at level L	T^L_{exec}
	Total runtime end-to-end kernel execution time	T_{runtime}

$$T^L_{compute} = \begin{cases} T^0_{compute}, & \text{if } L = 0 \\ T^{L-1}_{exec}, & \text{if } L > 0 \end{cases} \tag{4}$$

We model the temporal execution cost using a pipeline model across N^L_{TmpIter} iterations. The load and compute phases overlap within each iteration, while the store phase follows computation, reflecting the inherent overlap of execution phases, as shown below:

$$T^L_{temporal} = T^L_{Load} + (N^L_{\text{TmpIter}} - 1) \cdot \max(T^L_{Load}, T^L_{Compute}) + T^L_{Compute} + T^L_{Store} \tag{5}$$

To account for architecture-level parallelism, we introduce a spatial scaling factor based on the number of available parallel units at each level:

$$F^L_{spatial} = \left\lceil \frac{N^L_{\text{ParIter}}}{\lvert\text{ParallelUnit}^L\rvert} \right\rceil \tag{6}$$

The total execution latency at level L is then computed as the product of temporal latency and spatial scaling:

$$T^L_{exec} = F^L_{spatial} \cdot T^L_{temporal} \tag{7}$$

Finally, the total runtime end-to-end latency of a candidate strategy includes the kernel launch overhead and the cost at the highest level:

$$T_{runtime} = T_{launch} + T^{L_max}_{exec} \tag{8}$$

This recursive modeling approach enables efficient cost estimation across architectural layers and candidate structures, supporting candidate pruning and selection with minimal overhead.

Hybrid Switching. Our hybrid estimation strategy is motivated by two key observations. First, bottom-up kernel construction generates more candidates at higher levels due to increased loop nesting. Second, performance at lower levels is heavily influenced by unpredictable hardware behavior, such as out-of-order execution [11], which hinders accurate modeling.

To balance accuracy and cost, we apply profiling-based analysis at level $L = 0$ for CPUs, and at levels $L = 0$ and 1 for GPUs, where performance is highly sensitive and the candidate space remains small. At higher levels, we adopt the performance model to capture structural properties effectively. At runtime, performance modeling is consistently adopted to ensure negligible overhead. The effectiveness of this hybrid methodology, in terms of performance and runtime overhead, is further investigated in §7.5.

6 Implementation

This section presents the implementation of Helix, focusing on backend integration and the end-to-end execution workflow.

6.1 Backend Integration

Helix supports x86 CPUs, ARM CPUs, and NVIDIA GPUs through a unified abstraction. At its core is the *rKernel* data structure, which captures per-layer semantics and decouples architecture-specific behaviors, as shown in Listing 1, lines 1-12. Each layer is described by a `layer_meta_info` object. The `layer_depth` field defines its hierarchical position, while a `map<axis, LOOP_TYPE>` assigns loop semantics to each axis. These semantics include parallel (PL), temporal non-reduction (TNL), and temporal reduction (TRL), as shown

Listing 1: GPU GEMM code generation via rKernel.

```
1 // rKernel definition
2 enum ANALYZE_TYPE {profiling, modeling};
3 enum LOOP_TYPE {PL, TNL, TRL};
4 class axis;
5 class layer_meta_info {
6   int    layer_depth;                //Layer index in multi-level hierarchy
7   map<axis, LOOP_TYPE> loop_type;    // Axis-wise loop mapping
8   ANALYZE_TYPE analyzer;
9   func* load_func;                   //Customizable function pointers
10  func* store_func;
11  func* compute_func;
12 };
13 // Input IR, we omit block and grid for brevity
14 layer_meta_info gemm_tc_warp;
15 gemm_tc_warp.set(
16   layer_depth  = 0,
17   loop_type  = {"k0":TRL,"m0":TNL,"n0":TNL},
18   analyzer = profiling ,
19   load_func  = &loadfunc_shared2reg,
20   store_func = &storefunc_reg2shared,
21   compute_func = &compfunc_mma_sync_m16n8k16
22 );
23 // Output generated kernel
24 dim3 grid(M/m_tile_grid, N/n_tile_grid);
25 gemm_tensor_core_grid<<<grid, thread >>>(...) ;
26 __global__ void gemm_tensor_core_grid (...)  {
27   __shared half A_buf[], B_buf[], C_buf[]; // Tile buffers in shared memory
28   for (k2 = 0; k2 < K; k2 += k_tile_grid )
29     for (k1 = 0; k1 < k2; k1 += k_tile_block ) {
30       load_func_L1(A_buf, A);             //CTA-level load: global->shared
31       load_func_L1(B_buf, B);             //CTA-level load: global->shared
32       for (m0 = 0; m0 < m1; m0 += m_tile_warp)
33         for (n0 = 0; n0 < n1; n0 += n_tile_warp) {
34           load_func_L0(A_reg, A_buf);     //Warp-level load: shared->register
35           load_func_L0(B_reg, B_buf);     //Warp-level load: shared->register
36           C_reg = 0;
37           for (k0 = 0; k0 < k1; k0 += k_tile_warp)
38             compute_func_L0(C_reg, A_reg, B_reg);   //mma.sync.m16n8k16
39           store_func_L0(C_buf, C_reg); //Warp-level store: register->shared
40         }
41       store_func_L1(C, C_buf);            //CTA-level store: shared->global
42     }
43 }
```

in Algorithm 1. The `ANALYZE_TYPE` field selects between profiling- and model-based cost estimation. Each layer also specifies three functional bindings: `load_func`, `store_func`, and `compute_func`, which implement specific memory movement and computation logic. These components are composable across layers and support bottom-up kernel construction. As an example, lines 14–22 define a warp-level rKernel layer for GEMM, where `k0`, `m0`, and `n0` are classified as TRL, TNL, and TNL loop types, respectively, and the compute primitive is set to `mma.sync.m16n8k16`. While block and grid layers are omitted, this configuration demonstrates how execution semantics and architecture intrinsics are captured at the warp level.

Once the abstraction is configured, `Helix` applies the code generation process to produce optimized kernels. `Helix` is built on top of TVM [7], using the `tensorize` primitive to define `load`, `store`, and `compute` operations as reusable templates. These templates preserve low-level instruction behavior and enable architecture-specific optimization. During code generation, `Helix` traverses the abstraction hierarchy and lowers each layer into backend-specific instructions. To improve memory efficiency, data layout is customized to support coalesced global access and avoid bank conflicts in shared memory. Lines 23-42 in Listing 1 show a generated CUDA kernel for GEMM, where loop structures and instructions align with the abstraction defined in each layer. All loop structures and data movements are derived from the abstraction, and the kernel is synthesized based on the recursive composition of *rKernel* layers.

Algorithm 3 End-to-end workflow of `Helix`.

Input: input shape **s**
Output: selected kernel k^* and launch parameters p

1: **function** OfflineCompile
2: $N \leftarrow$ number of architecture levels
3: **for** $i = 0$ to $N-1$ **do**
4: $C_i \leftarrow$ GenLayerCands(i) ▷ Defined in Algorithm 2
5: **if** $i > 0$ **then**
6: **if** UseProfiling(i) **then**
7: ConstructImplByProfile(C_i, C_{i-1})
8: **else**
9: ConstructImplByModel(C_i, C_{i-1}) ▷ Defined in Section 5.2
10: **return** C_{N-1}
11: **end function**
12: **function** RuntimeExecute(**s**, $\{C_{L-1}^{(b)}\}_{b\in\mathcal{B}}$)
13: $\mathcal{E} \leftarrow \emptyset$
14: **for** each backend $b \in \mathcal{B}$ **do**
15: $\mathcal{E}_b \leftarrow$ EstimateCostByModel($C_{L-1}^{(b)}$, **s**) ▷ Perf. projection by Eqn 8
16: $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}_b$
17: $k^* \leftarrow$ SelectMinCost($\mathcal{E}$) ▷ Select optimal cand. across backends
18: $p \leftarrow$ InferLaunch(k^*, **s**) ▷ Infer runtime launch config
19: **return** k^*, p
20: **end function**

This abstraction aligns with backend code generation and generalizes across operators and targeted architectures. As the number of architectural levels is typically fixed, supporting new workloads requires only a few `layer_meta_info` instances, making the implementation lightweight and scalable.

6.2 Model-Driven End-to-End Workflow

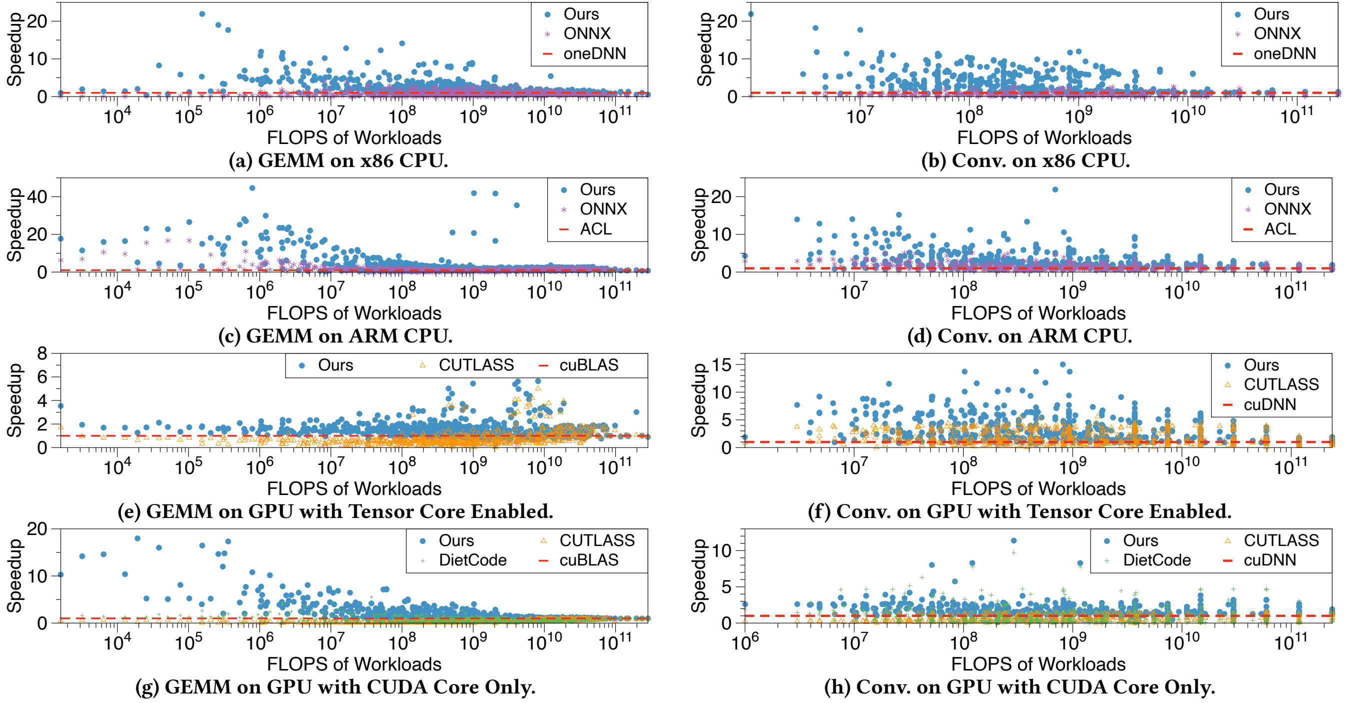
Algorithm 3 illustrates the end-to-end workflow of `Helix`, which is driven by our analytical performance model to estimate the theoretical peak performance. During the offline phase, `Helix` generates architecture-aligned micro-kernel candidates for each architecture level. For each level, ConstructImplByProfile or ConstructImplByModel recursively assembles valid implementations by combining current-level candidates with lower-level paths.

At runtime, given the input shape **s**, `Helix` evaluates all top-level kernel candidates across supported backends using shape-aware performance models (EstimateCostByModel, line 15). It then selects the minimum-cost implementation (SelectMinCost, line 17), and infers backend-specific launch parameters such as thread block configurations (line 18).

This design enables adaptive backend dispatch while addressing the dynamic shapes. For instance, on Nvidia GPUs [32], Tensor Cores require larger MMA instruction alignment, while CUDA Cores allow finer granularity. To leverage such architectural features, `Helix` compiles backend-specific candidate sets offline and performs shape-aware dispatch at runtime. A quantitative evaluation of this adaptive strategy is provided in §7.4.

Table 4: Hardware specifications.

Architecture	Intel x86 CPU	ARM CPU	Nvidia CUDA GPU
Version	Xeon 8255c (48 Cores)	AWS Neoverse N1 (70 Cores)	Ampere A100 (108 SMs)
Storage	Global: 250.53G; L3: 35.75M; L2: 1M/Core; L1: 32K/Core; Reg: 2K/Core	Global: 512G; L3: 32M; L2: 512K/Core; L1: 64K/Core; Reg: 2K/Core	Global: 40G; L2: 40M; Shared Memory: 48K/SM; Reg: 256K/SM
Peak Flops	7344 GFlops	3360 GFlops	CUDA Core: 19.5 TFlops; Tensor Core: 312 TFlops
Software	GCC: 10.2.0 LLVM: 15.0.3	GCC: 10.2.0 LLVM: 15.0.3	Driver Version: 450.156.00 CUDA Version: 11.8 cuDNN: 8.9.7.29



(a) GEMM on x86 CPU.
(b) Conv. on x86 CPU.
(c) GEMM on ARM CPU.
(d) Conv. on ARM CPU.
(e) GEMM on GPU with Tensor Core Enabled.
(f) Conv. on GPU with Tensor Core Enabled.
(g) GEMM on GPU with CUDA Core Only.
(h) Conv. on GPU with CUDA Core Only.

Figure 8: Performance results at operator-level. x86 CPU results are normalized to oneDNN; ARM CPU results are normalized to ACL; GPU results are normalized to cuBLAS for GEMM and cuDNN for convolution, respectively.

7 Evaluation

7.1 Experimental Setup

Platforms. We evaluate `Helix` across three representative platforms: Intel 8255c CPU [11], AWS Neoverse N1 ARM CPU [35], and Nvidia Ampere A100 GPU [32]. For CPUs, all evaluations are conducted using single-precision floating-point (FP32) computations. For GPUs, we conduct evaluations in two modes: *Tensor Core Enabled* mode with half-precision floating-point (FP16) data type, and *CUDA Core Only* mode with FP32 data type. Table 10 details our experimental platforms.

Benchmarks. Our evaluation encompasses two benchmark categories: operator-level and model-level. At the operator level, we collect 1389 different operator configurations from DeepBench [30] and real-world models, covering various tasks like Transformer, CNN, and GNN (see Table 11 and Table 12 for details). At the model level, we assess the performance of three Transformer-based language models (Bert [15], LLAMA-2-13B [47], GPT2 [37]) and three computer vision models (AlexNet [26], ResNet [23], GoogleNet [43]) for end-to-end dynamic-shape neural network evaluation. To mirror real-world scenarios, we randomly sample 60 sequence lengths from the ShareGPT dataset [44], with lengths ranging from 1 to 2k. For CNN models, we configure batch sizes beginning with 1, and then incrementally step from 4 to 64 in multiples of 4.

Baselines. We select various state-of-the-art (SOTA) baselines, categorized into two principal categories. The first category encompasses vendor-provided libraries. For Intel x86 CPUs, we compare GEMM and convolution performance with oneDNN [1] and ONNX Runtime [14]. For ARM CPUs, we compare GEMM and convolution performance with ACL [3] and ONNX Runtime [14]. For NVIDIA GPUs, we utilize cuBLAS [33] for GEMM and cuDNN [9] for convolution. We also evaluate CUTLASS [45] for both tasks. These libraries represent the vendor-optimized SOTA implementations available for their respective platforms. The second category is dynamic-shape compilers, for which we select DietCode [62], the leading open-source SOTA dynamic-shape compiler, as the baseline for comparison. Although BladeDISC [66] also supports dynamic shapes, its original paper indicates that it primarily focuses on model-level optimization and relies on sample-based code generation using CUTLASS [45] for kernel-level operations. However, the essential components responsible for kernel-level optimization are not attainable, limiting the ability to reproduce or fairly compare its results. Similarly, several recent dynamic-shape compilers [29, 59] lack public implementations. Hence, DietCode remains the most suitable publicly available baseline for evaluating dynamic-shape tensor program compilation.

Table 5: Benchmarked GEMM with dynamic shapes.

Category	M	N	K	#Cases
DeepBench [30]	[35, 8448]	[1, 6000]	[128, 500000]	84
Transformer [15, 37, 46, 47]	[1, 2000]	[768, 15360]	[768, 13824]	384
CNN [23, 26, 42, 43]	[1, 128]	[80, 25088]	[10, 4096]	80
GNN [17, 25, 50, 52]	[2708, 1888584]	[2, 121]	[8, 3703]	150

Table 6: Benchmarked Convolution with dynamic shapes.

Category	BS	Fmap	Filter	Cin	Cout	#Cases
DeepBench [30]	[1,16]	[7,700]	[1,20]	[1,2048]	[16,2048]	107
CNN [23, 26, 42, 43]	[1,64]	[4,768]	[1,11]	[3,832]	[16,512]	584

Table 7: Summary of operator-level speedups for Helix compared to various baselines across different setups.

Compute Architecture	Operator	Baseline	Cases with Speedup > 1 (%)	Average Speedup
x86 CPU	GEMM	oneDNN	77.3%	1.82×
		ONNX	88.3%	3.21×
	Conv.	oneDNN	85.8%	2.09×
		ONNX	99.1%	5.37×
ARM CPU	GEMM	ACL	79.9%	1.97×
		ONNX	86.9%	2.02×
	Conv.	ACL	81.3%	1.96×
		ONNX	92.0%	1.82×
GPU (Tensor Core Enabled)	GEMM	cuBLAS	82.9%	1.39×
		CUTLASS	92.8%	2.24×
	Conv.	cuDNN	89.9%	2.33×
		CUTLASS	80.5%	1.70×
GPU (CUDA Core Only)	GEMM	cuBLAS	74.9%	1.50×
		CUTLASS	99.4%	6.00×
		DietCode	95.7%	3.19×
	Conv.	cuDNN	91.1%	1.53×
		CUTLASS	87.8%	2.88×
		DietCode	92.5%	3.39×

Table 8: Speedups of Helix over DietCode for GEMM on GPU across different runtime ranges of M dimension.

96 Test Cases: M ∈ [1,384], N = 768, K = 2304			
Input Range for M	[0, 128)	[128, 256)	[256, 384)
Avg. Speedups	2.8x	1.4x	2.1x

7.2 Dynamic-Shape Tensor Program

We first present the evaluation of the single dynamic-shape tensor program, specifically assessing GEMM and Convolution operators on CPU and GPU platforms. Notably, DietCode is limited to GPU CUDA Cores and requires pre-determination of dynamic shape samples. We leverage the parameters from Table 11 and Table 12 as sample sets for DietCode's offline compilation process.

The evaluation results, shown in Figure 8, demonstrate Helix's performance across various configurations. The x-axis outlines the number of floating-point operations (FLOPs) for each workload, covering all GEMM cases from Table 11 and convolution cases from Table 12, while the y-axis represents the speedups. Helix consistently achieves a generalized performance speedup, demonstrating improvements across different hardware setups, operators, and tensor shapes. To further quantify the effectiveness of Helix, we emphasize two metrics: the percentage of cases where Helix shows performance improvement (defined as cases where the speedup is greater than one) and the average speedup across all cases. Table 7 presents these detailed results. Overall, the evaluation confirms that Helix provides robust and efficient acceleration results.

Additionally, we examine the impact of DietCode's reliance on sample-specific compilation. As shown in Table 8, we configure the M dimension dynamically, sampling and compiling it within the range [128, 256). The results show a performance decline when deviating from this range, highlighting DietCode's limited flexibility.

7.3 Dynamic-Shape Network

To assess the end-to-end performance of Helix on real-world dynamic workloads, we evaluate both language models and CNN models, as shown in Figure 9. We follow the evaluation setup described in §7.1. Specifically, we normalize the performance on x86 CPUs against oneDNN, on ARM CPUs against ACL, and on NVIDIA GPUs against cuBLAS for LLM and cuDNN for CNN, respectively.

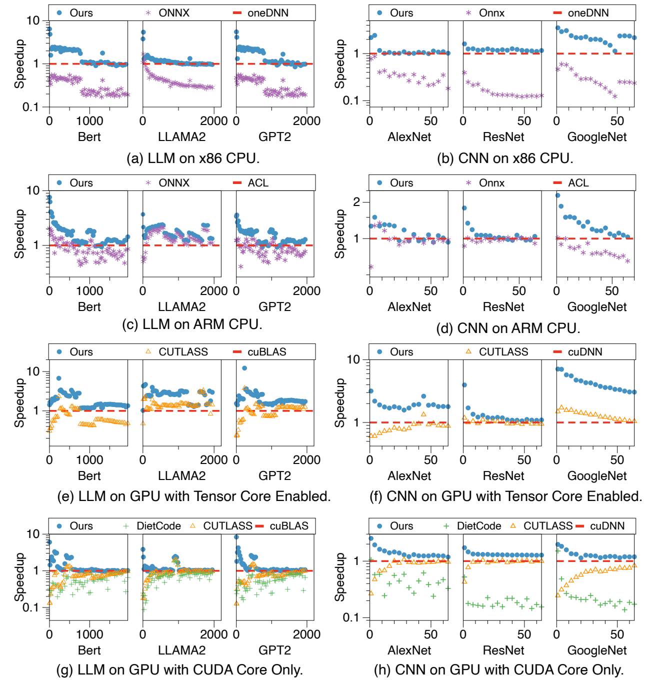


Figure 9: Performance results at model-level. The x-axis represents the sequence length for LLM and batch size for CNN, and the y-axis quantifies the speedups achieved relative to the baselines.

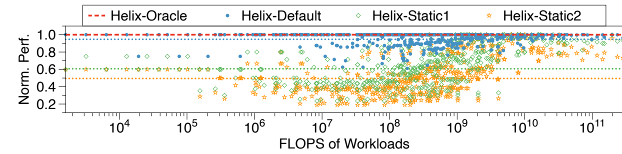


Figure 10: Performance comparison on A100 GPU with *Tensor Core Enabled* mode. The x-axis represents the FLOPS of workloads, and the y-axis shows performance normalized to Helix-*Oracle*. Dashed lines show the normalized average performance across test cases.

Helix demonstrates significant performance improvements across various tasks. Helix achieves notable average speedups of 2.06× for Bert, 1.65× for LLAMA2, and 2.00× for GPT2 across different baselines and architectures configurations. For CNNs, Helix achieves remarkable average speedups of 1.77× for AlexNet, 1.84× for ResNet, and 2, 72× for GoogleNet. In comparison to SOTA solutions, Helix achieves average 1.42×, 2.66×, 1.36×, 1.60×, 1.71×, 1.93× and 3.39× over oneDNN, ONNX Runtime, ACL, cuBLAS, cuDNN, CUTLASS, DietCode, respectively. These results demonstrate that Helix consistently achieves substantial speedups across a wide range of architectures, models, and dynamic workloads, highlighting its generality and practical effectiveness.

7.4 Effectiveness Analysis

Hierarchical Kernel Construction. To validate the effectiveness of Helix's hierarchical kernel construction methodology, we assessed its default configuration against three variants: *Helix-Oracle*, which utilizes Helix as a static-shape compiler with a profiling-based analyzer for all layers in every test case from Table 11; *Helix-Static1*,

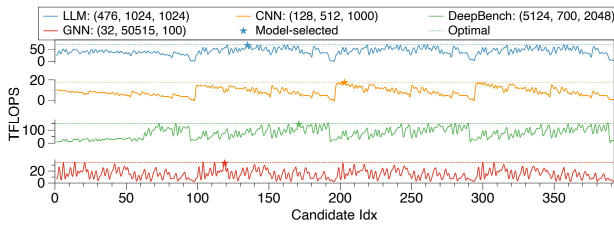


Figure 11: Performance comparison across candidate strategies for diverse GEMM workloads on A100 GPU with Tensor Core. The legend indicates the (M, N, K) parameters. The x-axis shows the candidate index, and the y-axis shows performance (TFLOPS), with stars marking model-selected strategies and dotted lines marking the best performance among candidates.

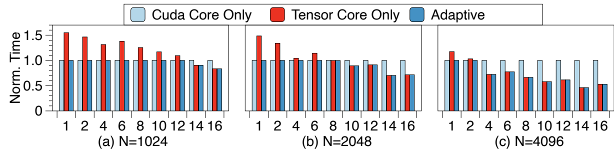


Figure 12: Performance comparison on A100 GPU across CUDA Core Only, Tensor Core Only, and Adaptive modes for GEMM. The x-axis represents the M value, and the y-axis shows the execution time normalized to the CUDA Core Only mode.

which maintains dynamic strategies at the L1 layer and adopts a static configuration for the L0 layer, selecting the most frequently optimal strategy; *Helix-Static2*, which disables dynamic strategy selection at both the L0 and L1 layers, applying the same fixed strategy as *Helix-Static1*. As shown in Figure 10, `Helix` achieves 94.7% of the performance of *Helix-Oracle*, while *Helix-Static1* and *Helix-Static2* reach only 60.7% and 49.5%, respectively. These experimental results underscore the effectiveness of `Helix`'s dynamic code generation and emphasize the importance of maintaining dynamic strategies across varying architecture hierarchies.

Model Accuracy Validation. We evaluate the accuracy of our performance model by comparing model-selected strategies to exhaustively profiled candidate strategies. Experiments are conducted on A100 GPU with *Tensor Core Enabled* mode, covering a set of GEMM configurations from different categories in Table 11. As shown in Figure 11, the model-selected strategies consistently achieve either optimal or near-optimal execution across all workloads. On average, the model-selected strategies achieve *97.8%* of the optimal performance across all evaluated workloads. Additionally, no single static strategy performs well across all cases, underscoring the need for adaptive selection. These findings validate the robustness and generality of our performance model.

Dynamic Backend Dispatch. We investigate the dynamic backend dispatch capability of `Helix` on GPUs using FP16 for the GEMM operator. We test N values of 1024, 2048, and 4096, with K fixed at 1024 and M dynamically adjusted from 1 to 16, across three settings: *CUDA Core Only*, *Tensor Core Only*, and the default *Adaptive*. The results in Figure 12, reveal dynamic backend adaptation opportunities. `Helix` effectively utilizes optimal backend, achieving performance gains of up to 48% and 54% over fixed CUDA and Tensor Core settings, respectively, demonstrating the effectiveness of its backend-adaptive scheduler.

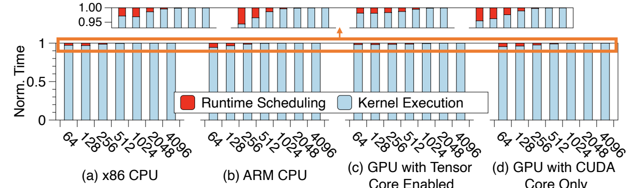


Figure 13: Runtime overhead breakdown of `Helix` in GEMM. The x-axis represents M/N/K parameters, and the y-axis represents normalized execution time.

Table 9: Comparison of `Helix`'s default configuration with the modified analyzer setup. 'E' denotes layers using the profiling method, while unlabeled layers use the modeling method.

Compute Architecture	Analyzer Configuration	Offline Overhead	Execution Performance
x86 CPU	Default (E: L0)	29.3 sec	1×
	Changed (E: L0, L1)	33.0 hour	1.04×
ARM CPU	Default (E: L0)	16.4 sec	1×
	Changed (E: L0, L1)	31.6 hour	1.01×
GPU (*Tensor Core Enabled*)	Default (E: L0, L1)	92.2 sec	1×
	Changed (E: L0)	19.4 sec	0.84×
GPU (*CUDA Core Only*)	Default (E: L0, L1)	529.6 sec	1×
	Changed (E: L0)	39.1 sec	0.63×

7.5 Discussion

Offline Overhead. We first analyze the offline overhead of `Helix`. For diverse tensor shapes, `Helix` requires only a single compilation process, substantially reducing overhead. We conduct GEMM evaluations on x86 CPUs, ARM CPUs, and NVIDIA GPUs under both *Tensor Core Enabled* and *CUDA Core Only* modes. In these settings, `Helix`'s candidate generation algorithm (Algorithm 2) produces 17731, 25925, 392, and 2332 distinct candidates, respectively. The corresponding compilation times are 29.3s, 16.4s, 92.2s, and 529.6s. In comparison, DietCode requires 25 hours for tuning in *CUDA Core Only* mode, using Table 11 as the sample set. Thus, `Helix` achieves a 174× improvement in compilation efficiency over DietCode.

Runtime Overhead. Figure 13 shows a breakdown of `Helix`'s execution, highlighting both runtime scheduling costs and the execution times of final tensor programs for various shapes. The GEMM tests are conducted with M/N/K values ranging from 64 to 4096. This runtime overhead impact is remarkably slight, accounting for only 0.29% of the total execution time, demonstrating the significant runtime efficiency of `Helix`.

Trade-off in Compilation and Execution Performance. We compare the default analyzer configuration with a modified variant, as detailed in Table 9, measuring the compilation time and the runtime performance for all cases in Table 11. On x86 and ARM CPUs, the modified configuration increases offline compilation overhead from seconds to more than 30 hours, while yielding only marginal performance gains of 1.04× and 1.01×, respectively. On GPUs, it slightly reduces compilation time but significantly degrades performance, with execution falling to 0.84× on Tensor Cores and 0.63× on CUDA Cores. These results underscore the trade-off, and `Helix` adopts a balanced configuration to ensure low compilation overhead while preserving high execution efficiency.

8 Related Work

For dynamic-shape tensor programs, vendor-provided libraries, such as cuBLAS [33], cuDNN [9], MKL-DNN [51] and CUTLASS [45]

are extensively utilized in prevalent frameworks, facilitating high-performance tensor operations across diverse hardware platforms. These libraries, tailored to specific target hardware, require substantial engineering efforts. Helix, by introducing a novel unified recursive abstraction, significantly reduces development costs and unifies optimization strategies across different hardware platforms.

Additionally, compilation optimization is a crucial solution for dynamic-shape tensor programs. Existing methods, such as DietCode [62], Nimble [41], and DISC [66], predominantly rely on sample-based compilation approaches. MikPoly [59] builds on CUTLASS [45] and is therefore limited by its fixed kernel templates and tiling patterns, reducing generality across operators. There are also graph-focused compilation efforts [53, 55, 68, 69], which target dynamic and irregular computation patterns by optimizing sparse operators in graph-based models. These approaches often incorporate model-specific assumptions or rely on graph-level semantics, making them less applicable to general tensor programs. Our work, distinguishing itself from existing efforts, uses hardware information as a fundamental element to construct a novel sample-free compilation workflow, thus enabling diverse and general high-performance support.

For optimizing tensor programs with static shapes, various tensor compilers such as AutoTVM [8], FlexTensor [65], Ansor [63], and TensorIR [16] have been proposed. However, these methods are associated with significant compile time overheads. Although efforts like Roller [70] have attempted to optimize the compilation time for static-shape compilers, the time required is still considerably longer than the execution overhead, making them impractical for the online demands of dynamic-shape tensor programs.

Model-level optimization is another important component for end-to-end DNN optimizations. DNNFusion [31], Rammer [28], Chimera [64], and AStitch [67] focus on fusion optimizations in DNNs. TASO [24], Unity [48], XLA [40], JAX [5], TorchDynamo [36], TenSAT [57] explore graph rewriting opportunities. In this paper, our proposed Helix focuses on operator-level for dynamic-shape tensor programs, which is orthogonal to these works. Meanwhile, Helix is designed without inherent limitations for integration with current model-level compilers. We look forward to exploring this area as a part of our future research efforts.

9 Conclusion

In this work, we propose Helix, an architecture-guided and sample-free dynamic-shape tensor program compilation framework. Helix employs bidirectional compilation to ensure high performance with minimal overhead. Experimental results show that Helix achieves average execution speedups of 2.26× over vendor libraries and 3.29× over existing compilers, with 174× lower compilation cost. These results demonstrate Helix's effectiveness and its potential as a standard approach for dynamic-shape optimization.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China (NSFC) Grants (U21B2017 and 62222210) and Shanghai Qi Zhi Institute Innovation Program SQZ202316. This work was also supported in part by the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office (A-8002767-00-00). This work was partly supported by National Science Foundation of China under grant no. 12426303, Shenzhen-HongKong Joint Funding Project (Category A) under grant no. SGDX20230116092056010, the Shenzhen Basic Research Fund under grant no. KQTD20200820113106007, ZDSYS20220422103800001, JCYJ20240813154907010. We would also like to thank the funding support by the Key Laboratory of Quantitative Synthetic Biology, Chinese Academy of Sciences under grant no. CKL075, Shenzhen Key Laboratory of Intelligent Bioinformatics under grant no.ZDSYS20220422103800001. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] 2020. oneAPI Deep Neural Network Library (oneDNN). https://github.com/oneapi-src/oneDNN.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[3] ARM. 2017. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary/

[4] Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. 2023. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. *arXiv preprint arXiv:2310.05424* (2023).

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. 2018. JAX: composable transformations of Python+NumPy programs. *Version 0.2* 5 (2018), 14–24.

[6] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A versatile software systolic execution model for GPU memory-bound kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 53, 81 pages. doi:10.1145/3295500.3356162

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

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

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[10] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy batching: An SLA-aware batching system for cloud machine learning inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 493–506.

[11] George Chrysos. 2014. Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper* 176, 2014 (2014), 43–50.

[12] Alina Carmen Cojocaru and M Ram Murty. 2005. *An introduction to sieve methods and their applications.* Number 66. Cambridge University Press.

[13] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-EntryMulti-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.

[14] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: x.y.z.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction

for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.
[17] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
[18] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.
[19] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 1–25.
[20] Heine Halberstam and Hans Egon Richert. 2013. *Sieve methods*. Courier Corporation.
[21] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 11 (2021), 7436–7456.
[22] Zekun Hao, Yu Liu, Hongwei Qin, Junjie Yan, Xiu Li, and Xiaolin Hu. 2017. Scale-aware face detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6186–6195.
[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[24] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.
[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).
[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[28] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.
[29] Pengyu Mu, Yi Liu, Rui Wang, Guoxiang Liu, Zhonghao Sun, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2023. Haotuner: A hardware adaptive operator auto-tuner for dynamic shape tensor compilers. *IEEE Trans. Comput.* 72, 11 (2023), 3178–3190.
[30] S Narang and G Diamos. 2017. DeepBench: Benchmarking deep learning operations on different hardware.
[31] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.
[32] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.
[33] NVIDIA Corporation. 2025. *NVIDIA cuBLAS Documentation*. https://docs.nvidia.com/cuda/cublas/index.html
[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[35] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. 2020. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro* 40, 2 (2020), 53–62.
[36] PyTorch Contributors. 2022. TorchDynamo. https://pytorch.org/docs/master/dynamo/.
[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[38] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
[39] Haseena Rahmath P, Vishal Srivastava, Kuldeep Chaurasia, Roberto G Pacheco, and Rodrigo S Couto. 2024. Early-exit deep neural network-a comprehensive survey. *Comput. Surveys* 57, 3 (2024), 1–37.
[40] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.
[41] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021), 208–222.
[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.
[44] Sharegpt teams. 2023. Sharegot. https://sharegpt.com/
[45] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. CUTLASS. https://github.com/NVIDIA/cutlass
[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[47] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[48] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.
[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[50] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[51] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.
[52] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 515–531.
[53] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 359–375.
[54] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
[55] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. *Proceedings of Machine Learning and Systems* 4 (2022), 515–528.
[56] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 2246–2251. https://www.aclweb.org/anthology/2020.acl-main.204
[57] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268.
[58] Zerui Yang, Yuhui Xu, Wenrui Dai, and Hongkai Xiong. 2019. Dynamic-stride-net: Deep convolutional neural network with dynamic stride. In *Optoelectronic Imaging and Multimedia Technology VI*, Vol. 11187. SPIE, 42–53.
[59] Feng Yu, Guangli Li, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2024. 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*. 797–812.
[60] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.
[61] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).
[62] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems* 4 (2022), 848–863.
[63] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*.

863–879.
[64] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1113–1126.
[65] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.
[66] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, et al. 2023. BladeDISC: Optimizing Dynamic Shape Machine Learning Workloads via Compiler Approach. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–29.
[67] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, et al. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.
[68] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, et al. 2023. ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 878–891.
[69] Yangjie Zhou, Yaoxu Song, Jingwen Leng, Zihan Liu, Weihao Cui, Zhendong Zhang, Cong Guo, Quan Chen, Li Li, and Minyi Guo. 2023. AdaptGear: Accelerating GNN Training via Adaptive Subgraph-Level Kernels on GPUs. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 52–62.
[70] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

This paper presents an open-source library for A Sample-Free Compilation Framework for Efficient Dynamic Tensor Computation. Our library, Helix, is designed to support a wide range of computation devices: from ARM and X86 CPUs to Nvidia GPUs. The computational artifacts are designed to demonstrate the sample-free compilation of Helix, and the performance advantages of Helix.

Helix is publicly available at: https://doi.org/10.5281/zenodo.16735209.

A.1 Paper's Main Contributions

This paper makes the following contributions:

C_1 We identify the limitations of existing sample-driven dynamic-shape compilers and highlight architecture-guided compilation as a path to eliminate shape-specific tuning while improving both compilation efficiency and execution adaptability.

C_2 We propose a bidirectional compilation framework, integrating top-down hierarchical decomposition with bottom-up, architecture-guided kernel construction. This approach reduces compilation overhead while maintaining adaptability across workloads.

C_3 We introduce a hybrid performance analyzer that recursively combines profiling and modeling to estimate layer-wise kernel performance. It enables accurate cost estimation and kernel selection across architectural levels, supporting both offline pruning and runtime dispatch without online tuning.

C_4 We implement Helix, and evaluate it on Intel x86 CPU, Arm CPU, and Nvidia CUDA GPU. Our evaluation confirms a 2∼3× speedup on performance with two orders of magnitude less compilation time. It demonstrates Helix's superiority over existing techniques.

A.2 Computational Artifacts

The computational artifacts related to this paper are listed as follows.

A_1 scripts/benchmark_op_level_eval.py
A_2 scripts/benchmark_model_level_eval.py
A_3 scripts/Ampere_FP16/benchmark_Ampere_FP16_performance_comparison.py
A_4 scripts/Ampere_FP16/candidate_performance_curve.py
A_5 scripts/Ampere_FP16/TensorCore_CUDACore_Adaptive.py
A_6 scripts/runtime_cost.py
A_7 scripts/changed_profile.py

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1 C_2 C_3 C_4	Section 7 Figure 8 Section 7 Table 7
A_2	C_1 C_2 C_3 C_4	Section 7 Figure 9
A_3	C_2	Section 7 Figure 10
A_4	C_2	Section 7 Figure 11
A_5	C_2	Section 7 Figure 12
A_6	C_2 C_3	Section 7 Figure 13
A_7	C_3	Section 7 Table 9

All evaluations can be fully reproduced by the reader.

A_1) Figure 8 is the operator-level performance comparison between Helix and other state-of-art works on ARM, X86 CPUs and Nvidia A100 GPGPUs. Table 7 is a summary of operator-level speedups for Helix compared to various baselines across different hardware setups.

A_2) Figure 9 is the model-level performance comparison between Helix and other state-of-the-art works on ARM, X86 CPUs, and Nvidia A100 GPGPUs.

A_3) Figure 10 is the experimental results underscore the effectiveness of Helix's dynamic code generation and emphasize the importance of maintaining dynamic strategies across varying architecture hierarchies.

A_4) Figure 11 illustrates the performance comparison results across candidate strategies for diverse GEMM workloads on A100 GPU with Tensor Core.

A_5) Figure 12 illustrates the performance comparison results on A100 GPU with CUDA Core Only, Tensor Core Only, and Adaptive modes for GEMM.

A_6) Figure 13 shows a breakdown of Helix's execution, highlighting both runtime scheduling costs and the execution times of final tensor programs for various shapes.

A_7) Table 9 compares the default analyzer configuration with a modified variant, illustrating the compilation time and the runtime performance for all cases in Table 5.

B Artifact Identification

The identification of the above seven computational artifacts A_i will be provided in the following six subsections.

B.1 Computational Artifact A_1

Relation To Contributions

The operator-level evaluation on execution performance adaptability is assessed using GEMM and Convolution operators.

Expected Results

Helix consistently achieves a generalized performance speedup over other state-of-art works on ARM CPU, x86 CPU, and GPGPU.

Expected Reproduction Time (in Minutes)

The expected reproduction time is approximately 150 minutes on x86 CPU, 180 minutes on ARM CPU, 90 minutes in GPU FP16 mode, and 120 minutes in GPU FP32 mode.

Artifact Setup (incl. Inputs)

Hardware. We evaluate Helix on three representative platforms: Intel 8255c CPU, AWS Neoverse N1 ARM CPU, and Nvidia Ampere A100 GPU. CPU evaluations use single-precision (FP32) computation. GPU evaluations are conducted in two modes: Tensor Core Enabled (FP16) and CUDA Core Only (FP32). The table below summarizes the hardware and software configurations.

Table 10: Hardware specifications.

Architecture	Intel x86 CPU	ARM CPU	Nvidia CUDA GPU
Version	Xeon 8255c (48 Cores)	AWS Neoverse N1 (70 Cores)	Ampere A100 (108 SMs)
Storage	Global: 250.53G; L3: 35.75M; L2: 1M/Core; L1: 32K/Core; Reg: 2K/Core	Global: 512G; L3: 32M; L2: 512K/Core; L1: 64K/Core; Reg: 2K/Core	Global: 40G; L2: 40M; Shared Memory: 48K/SM; Reg: 256K/SM
Peak Flops	7344 GFlops	3360 GFlops	CUDA Core: 19.5 TFlops; Tensor Core: 312 TFlops
Software	GCC: 10.2.0 LLVM: 15.0.3	GCC: 10.2.0 LLVM: 15.0.3	Driver Version: 450.156.00 CUDA Version: 11.8 cuDNN: 8.9.7.29

Datasets / Inputs. We collect 1,389 different operator configurations from DeepBench and real-world models, covering various tasks such as Transformer, CNN, and GNN (see Table 11 and Table 12 for details).

Table 11: Benchmarked GEMM with dynamic shapes.

Category	M	N	K	#Cases
DeepBench	[35, 8448]	[1, 6000]	[128, 500000]	84
Transformer	[1, 2000]	[768, 15360]	[768, 13824]	384
CNN	[1, 128]	[80, 25088]	[10, 4096]	80
GNN	[2708, 1888584]	[2, 121]	[8, 3703]	150

Table 12: Benchmarked Convolution with dynamic shapes.

Category	BS	Fmap	Filter	Cin	Cout	#Cases
DeepBench	[1,16]	[7,700]	[1,20]	[1,2048]	[16,2048]	107
CNN	[1,64]	[4,768]	[1,11]	[3,832]	[16,512]	584

Installation and Deployment. We select various SOTA baselines. For Intel x86 CPUs, we compare GEMM and convolution performance with oneDNN and ONNX Runtime. For ARM CPUs, we compare the performance of GEMM and convolution with ACL and ONNX Runtime. For NVIDIA GPUs, we utilize cuBLAS for GEMM and cuDNN for convolution. We also evaluate CUTLASS for both tasks.

Artifact Execution

To run the operator-level evaluation, use the script we provide with different parameters:

```
python scripts/benchmark_op_level_eval.py -op [. . . ]
        -backend [. . . ] -system [. . . ]
```

The script accepts the following arguments:

- `-op`: specifies the operator type (e.g., `GEMM`, `Conv`).
- `-backend`: selects the hardware backend (e.g., `ARM`, `x86`, `GPU_FP16`, `GPU_FP32`).
- `-system`: sets the system or baseline to evaluate (e.g., `Helix`, `cublas`, `onnxruntime`).

Artifact Analysis (incl. Outputs)

Helix consistently achieves a generalized performance speedup, which demonstrates execution adaptability across different hardware setups, operators, and tensor shapes.

B.2 Computational Artifact A_2

Relation To Contributions

The model-level evaluation is conducted on representative LLM and CNN models.

Expected Results

Helix consistently achieves a generalized performance speedup over other state-of-art works on ARM CPU, x86 CPU, and GPGPU.

Expected Reproduction Time (in Minutes)

The expected reproduction time is approximately 150 minutes on x86 CPU, 180 minutes on ARM CPU, 90 minutes in GPU FP16 mode, and 120 minutes in GPU FP32 mode.

Artifact Setup (incl. Inputs)

Hardware. Same as Computational Artifact A_1.

Datasets / Inputs. We evaluate three Transformer-based language models (BERT, LLAMA-2-13B, GPT-2) and three vision models (AlexNet, ResNet, GoogleNet) to assess model-level performance. To reflect real-world scenarios, we sample 60 sequence lengths (1–2k) from the ShareGPT dataset. For CNNs, we use batch size 1 and then increment it from 4 to 64 in steps of 4.

Installation and Deployment. Same as Computational Artifact A_1.

Artifact Execution

To run the model-level evaluation, use the script we provide with different parameters:

```
python scripts/benchmark_model_level_eval.py -model
      [. . . ] -backend [. . . ] -system [. . . ]
```

The script accepts the following arguments:

- `-model`: specifies the model type (e.g., `LLM`, `CNN`).
- `-backend`: selects the hardware backend (e.g., `ARM`, `x86`, `GPU_FP16`, `GPU_FP32`).
- `-system`: sets the system or baseline to evaluate (e.g., `Helix`, `cublas`, `onnxruntime`).

Artifact Analysis (incl. Outputs)

Same as Computational Artifact A_1.

B.3 Computational Artifact A_3

Relation To Contributions

This artifact validates the effectiveness of Helix's hierarchical kernel construction.

Expected Results

Helix demonstrates effective dynamic code generation that adapts across architectural levels.

Expected Reproduction Time (in Minutes)

300 min (GPU A100).

Artifact Setup (incl. Inputs)

Same as Computational Artifact A_1.

Artifact Execution

To run the evaluation, use the script we provide:

```
python scripts/Ampere_FP16/benchmark_Ampere
_FP16_performance_comparison.py
```

Artifact Analysis (incl. Outputs)

Helix-Default outperforms other configurations, demonstrating the benefit of hierarchical dynamic strategy selection.

B.4 Computational Artifact A_4

Relation To Contributions

This artifact evaluates the effectiveness of Helix's strategy selection by comparing the performance of different candidate strategies on GEMM workloads.

Expected Results

Helix adaptively selects efficient strategies based on workload characteristics.

Expected Reproduction Time (in Minutes)

5 min (GPU A100).

Artifact Setup (incl. Inputs)

Same as Computational Artifact A_1.

Artifact Execution

To run the evaluation, use the script we provide:

```
python
scripts/Ampere_FP16/candidate_performance_curve.py
```

Artifact Analysis (incl. Outputs)

Helix consistently selects the high-performance strategy across dynamic shapes.

B.5 Computational Artifact A_5

Relation To Contributions

This artifact compares the performance of Helix under different GPU execution modes: Tensor Core only, CUDA Core only, and Adaptive.

Expected Results

Helix leverages adaptive execution to dynamically choose the most efficient compute path on modern GPUs.

Expected Reproduction Time (in Minutes)

5 min (GPU A100).

Artifact Setup (incl. Inputs)

Same as Computational Artifact A_1.

Artifact Execution

To run the evaluation, use the script we provide:

```
python
scripts/Ampere_FP16/TensorCore_CUDACore_Adaptive.py
```

Artifact Analysis (incl. Outputs)

Helix's Adaptive mode outperforms static alternatives by utilizing architecture-aware dynamic selection.

B.6 Computational Artifact A_6

Relation To Contributions

This artifact provides a breakdown of Helix's runtime behavior, including runtime scheduling and final kernel execution.

Expected Results

Helix minimizes runtime scheduling overhead while maintaining efficient execution.

Expected Reproduction Time (in Minutes)

The expected reproduction time is approximately 2 minutes on x86 CPU, 2 minutes on ARM CPU, 1 minute in GPU FP16 mode, and 1 minute in GPU FP32 mode.

Artifact Setup (incl. Inputs)

Same as Computational Artifact A_1.

Artifact Execution

To run the evaluation, use the script we provide:

```
python scripts/runtime_cost.py –backend [. . . ]
```

The script accepts the following arguments:

- `–backend`: selects the hardware backend (e.g., `ARM`, `x86`, `GPU_FP16`, `GPU_FP32`).

Artifact Analysis (incl. Outputs)

The results highlight low scheduling cost and efficient kernel execution across various dynamic workloads.

B.7 Computational Artifact A_7

Relation To Contributions

This artifact compares the default and modified configurations to assess the impact on compilation and execution performance.

Expected Results

Helix's hybrid analyzer design enables a favorable trade-off between compile-time and runtime performance.

Expected Reproduction Time (in Minutes)

The expected reproduction time is approximately 2000 minutes on x86 CPU, 2000 minutes on ARM CPU, 10 minutes in GPU FP16 mode, and 10 minutes in GPU FP32 mode.

Artifact Setup (incl. Inputs)

Same as Computational Artifact A_1.

Artifact Execution

To run the evaluation, use the script we provide:

```
python scripts/changed_profile.py -backend [. . . ]
```

The script accepts the following arguments:

- `-backend`: selects the hardware backend (e.g., `ARM`, `x86`, `GPU_FP16`, `GPU_FP32`).

Artifact Analysis (incl. Outputs)

Results validate that Helix's default analyzer achieves better compilation-runtime balance across all test cases.

Artifact Evaluation (AE)

The detailed instructions of how to setup, execute, and analyze the provided artifacts is included in the subsections of the artifact description part above.