A Sample-Free Compilation Framework for Efficient Dynamic Tensor Computation

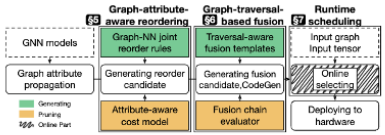
Yangjie Zhou, Honglin Zhu, Qian Qiu, Weihao Cui, Zihan Liu, Peng Chen, Mohamed Wahib, Cong Guo, Siyuan Feng, Jintao Meng, Haidong Lan, Jingwen Leng, Yun Lin, Jin Song Dong, Wenxi Zhu, Minwen Deng.

SC 2025

My Research: Full-Stack System Support for ML

End-to-End System Support for ML: From Hardware to Framework

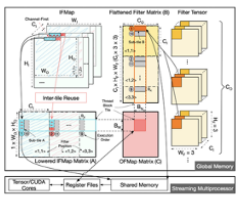
Scalable, Efficient, and Adaptive Execution



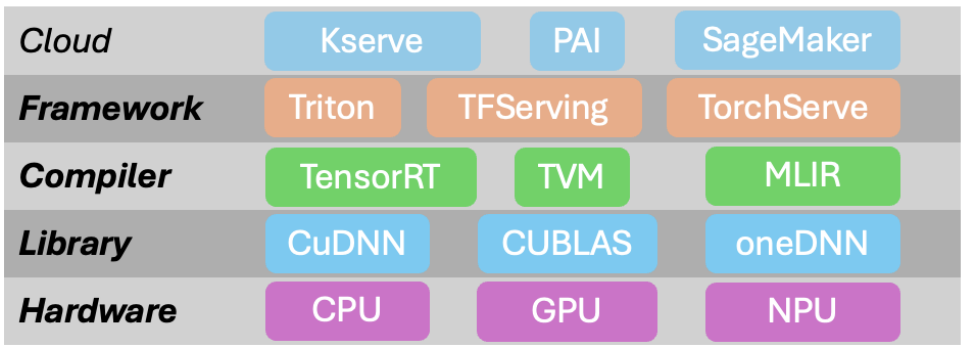
Input-Adaptive Trans. for GNN[ASPLOS 25]

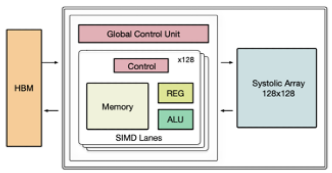
Unified Graph Operator Abstraction [ASPLOS 23]

Adaptive Sub-Graph Kernel [CF 23]



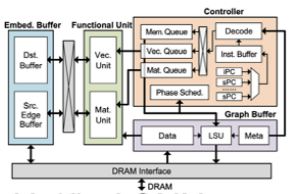
Implicit Convolution Algorithm [IISWC 21]



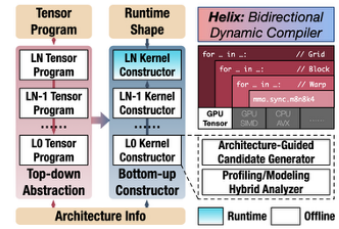


TPU Simulator [DAC20 poster]

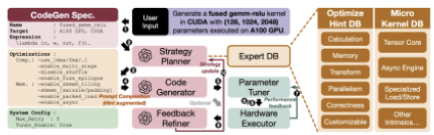
GPU-Systolic Array Integration[DAC20]



Unified GNN Accelerator [TCAD' 25]



Dynamic Shape Opt. [SC' 25]



LLM for Cuda Code Gen. [Submit to ASPLOS' 26]

Covering architecture, kernel optimization, compiler/framework infra co-design.

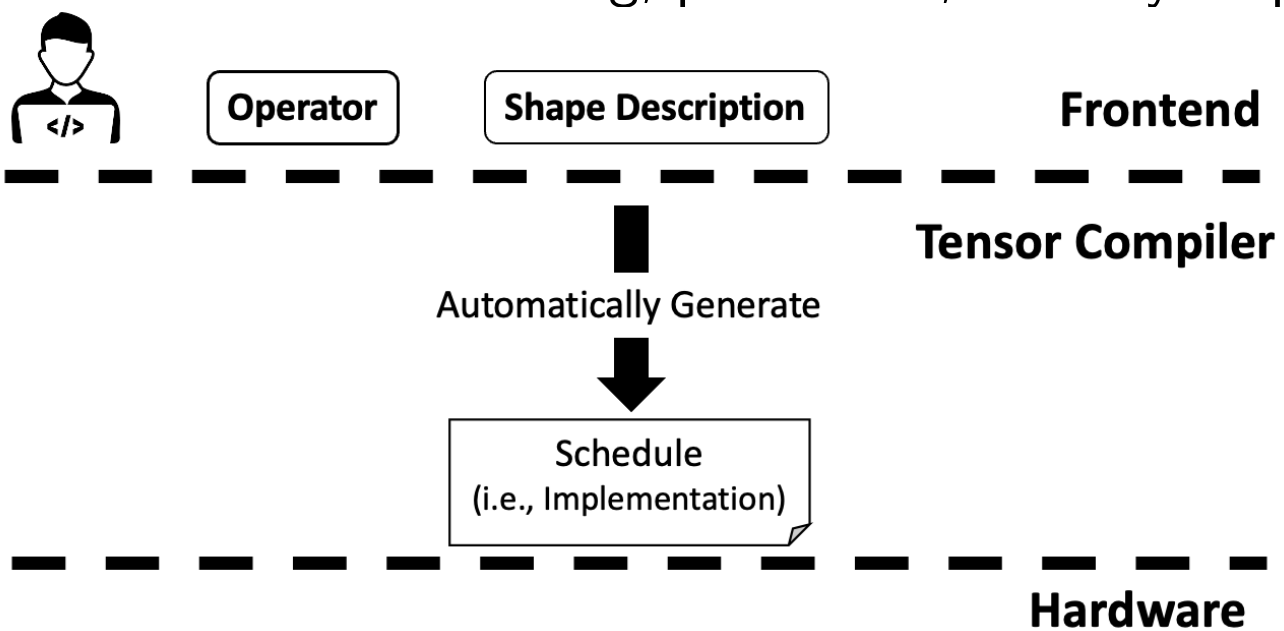
Why Do We Need Compilers for Tensor Programs?

Tensor programs are performance-critical:

- Modern DNNs involve millions of tensor ops (e.g., GEMM, Conv).
- Performance affects latency, energy, and deployment cost.

Manual kernel optimization is not scalable:

- 100+ hardware backends: CPU/GPU/ASIC.
- 10K+ operator variants: shape, layout, type...
- 1B+ optimization choices: tiling, parallelism, memory mapping...



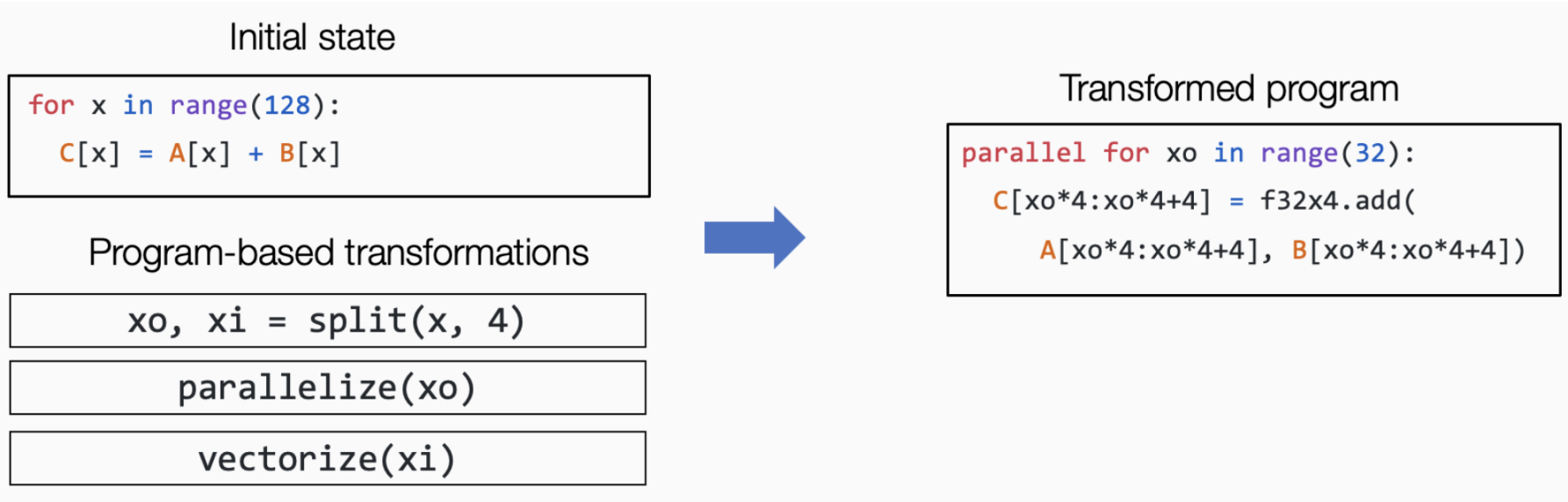
Tensor compiler automatically explores possible schedules (strategy implementation) and generates optimized kernels.

How Do Compilers Optimize Tensor Programs?

Core optimization tasks of tensor compilers:

- **Loop transformation:** decide unrolling, parallelization, and vectorization.
- **Tiling size configuration:** balance performance with hardware constraints.
- **Low-level implementation:** map schedules to low-level intrinsics.

Example:



Compilers transform tensor programs into hardware-efficient kernels via loop scheduling and low-level code generation.

Compilers: Advantages and Remaining Challenges

Aspect	Handwritten Kernels	Tensor Compiler
Loop trans.	Expert-crafted heuristics	Auto. scheduling search
Tiling config	Hand-tuned for regular shapes	Auto. config search
Low-level impl.	Expert hand-tuned intrinsics	Relies on limited intrinsics; lacks fine-grained control

Advantages of Tensor Compilers:

- **Portability:** adapt quickly across hardware and operators.
- **Shape generalization:** efficient kernels for irregular shapes.

Dynamic-shape Challenge:

- Even for static shapes, compilation takes hours ($\sim$5h per conv op).
- When shapes vary at runtime, search space **explodes**, making offline/online tuning infeasible.

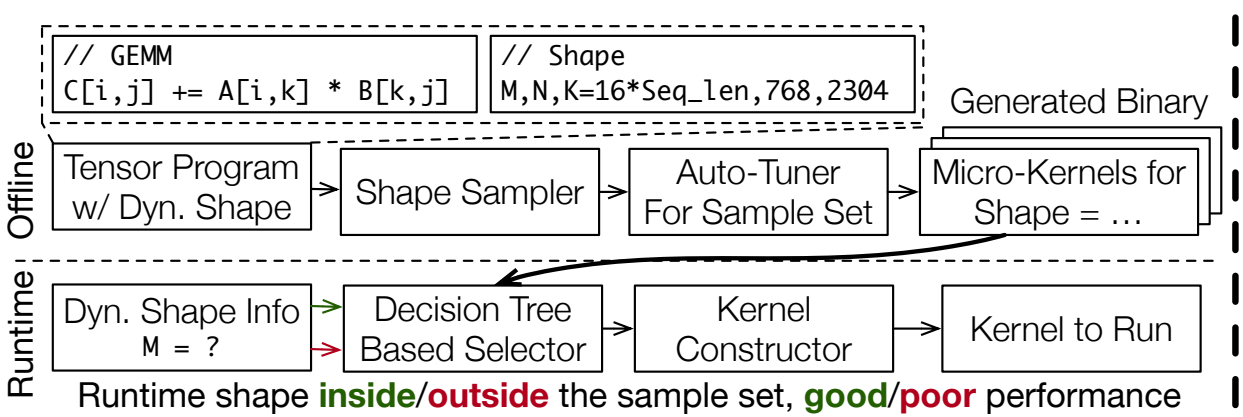
Key Question:

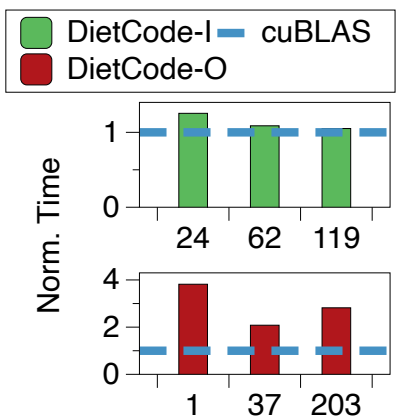
How can we deliver **high-performance kernels for diverse shapes within limited compilation time**?

Existing Dynamic Shape Compilation

Sampling-based Method (DietCode[MLSys'22], BladeDISC[SIGMOD'24])

- **Offline Sample-Based Tuning:**
  - Sample a fixed set of dynamic shapes.
  - Auto-tune each shape to generate optimized micro-kernels.
- **Runtime Selection:**
  - Use decision trees to select kernels at runtime.
  - Fall back to kernel construction for unseen shapes.





Key Limitation

Sample-based tuning only optimizes a narrow subset of possible shapes.
When runtime shapes fall outside this set, performance drops significantly.
Moreover, the compilation cost of tuning grows with the sample set.

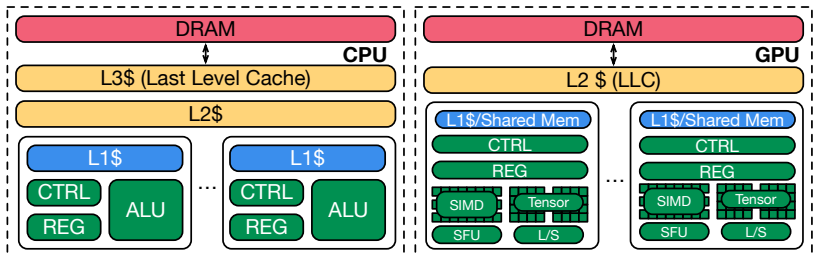
Key Insight: Hardware-Guided Compilation

Limitations of Existing Sample-based Compilers:

- Rely on expensive sample-based search, infeasible under large sampling sets.
- Focus solely on dynamic shapes, ignoring other information.

Observation 1: While shapes are unknown, hardware is *fixed and hierarchical*.

- Memory hierarchy & compute units.
- Resource constraints (e.g., registers).
- Instruction usage (e.g., MMA).



CPU/GPU architectures diagram.

Observation 2: Tensor program execution itself exhibits a *stacked / hierarchical structure*. Performance drops sharply once resource usage exceeds hardware limits.

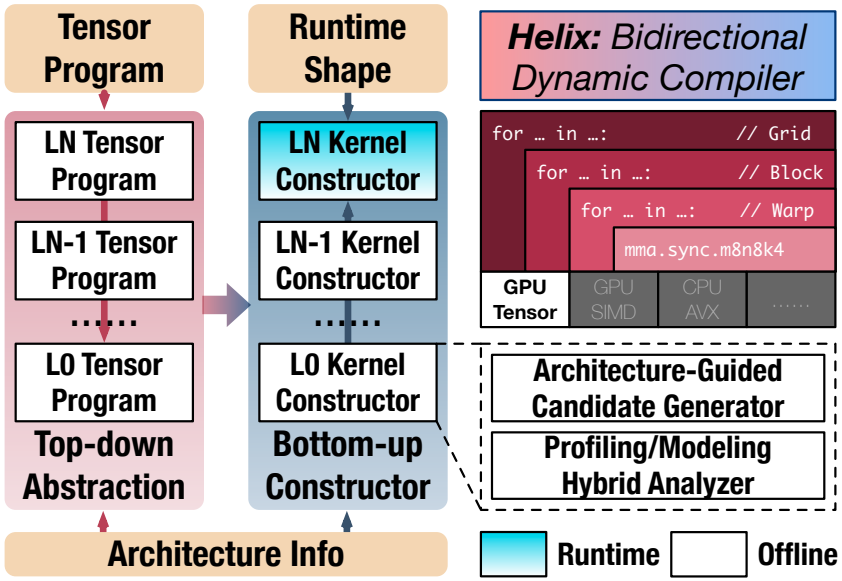
Design Shift

Hardware hierarchy provides a natural structure to guide compilation.
We can transform a global shape-explosive search into **layer-wise, hardware -constrained** subproblems.

Helix Overview: Bidirectional Compilation Framework

Core Design: Helix organizes compilation into **shape-agnostic abstraction** and **hardware-driven** construction to tackle dynamic shapes challenge.

- **Top-Down Abstraction**: Decomposes programs into hierarchical, shape-agnostic representations.
- **Bottom-Up Construction**: Builds optimized code from hardware-aligned micro-kernels upward.
- **Hybrid Analyzer**: Combines profiling and analytical modeling to guide hardware-aware kernel construction.



Helix system overview.

System Insight

Helix's layered design ensures:

– **High-level layers**: support dynamic shape generality.
– **Low-level layers**: exploit hardware intrinsics for peak performance.
– **Layered structure**: controls compilation overhead.

Top-Down Abstraction: Template for Loop Transformation

Goal: Construct a unified, general abstraction to enable hierarchical, top-down tensor program decomposition.

Unified Abstraction

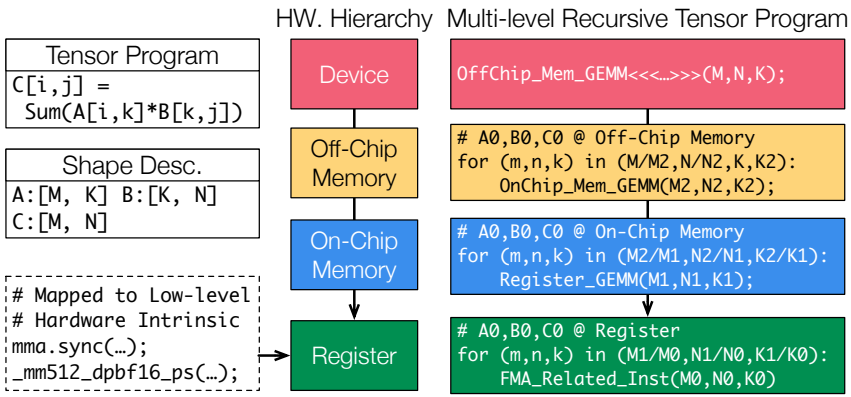
- **Parallelize** temporal/parallel loops
- **Invoke** load/compute/store primitives
- **Recurse** into the next level

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)
```

GEMM Mapping Example

- **Device**: end-to-end GEMM
- **Off-chip**: off-chip micro-kernel
- **On-chip**: on-chip micro-kernel
- **Register**: register micro-kernel



Insight:

- Serves as a reusable template for loop transformations, reducing compilation overhead and avoiding costly search in traditional tensor compilers.

Bottom-Up Kernel Generation: Layerwise High-perf. Impl.

Goal: Build valid high-performance candidate sets at each layer by generating and filtering tiling options under hardware constraints.

Layer-wise Workflow

Layer 0 (base layer):

- Start from manual kernel optimization.
- Guarantee to satisfy hardware resource and ISA constraints.

Layer $L > 0$ (recursive construction):

- Initialize candidates under current-layer hardware constraints.
- Filter candidates by shape-aligned composability with the previous layer.

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
```

Insight:

- Manually optimized base micro-kernels ensure hardware-feasible high-perf.
- Recursive filtering ensures higher-layer legality via composability.

Performance Analyzer: Hierarchical Cost Propagation

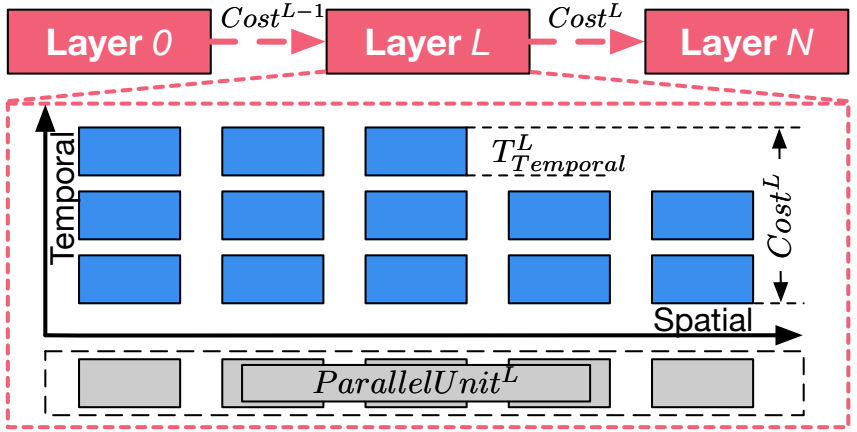
Goal: Estimate execution latency of tiling candidates via hierarchical cost models with layer-wise propagation.

Layer-0 Compute Cost:

$$T^0_{compute} = \begin{cases} \frac{FLOPs^0}{F_{CPU}}, & \text{CPU} \\ \frac{FLOPs^0}{F_{TC}}, & \text{Tensor Core} \\ \frac{FLOPs^0}{F_{CUDA}}, & \text{CUDA Core} \end{cases}$$

Recursive Compute Cost:

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



Layer-wise costs propagation.

Temporal Execution Cost:

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

Layer-Wise Execution Time:

$$T^L_{exec} = \left\lceil \frac{N^L_{\text{ParIter}}}{|\text{ParallelUnit}^L|} \right\rceil \cdot T^L_{temporal}$$

Insight: By combining recursive abstraction with hierarchical cost modeling, Helix achieves end-to-end performance prediction, guiding the selection of high-performance tiling configurations.

Design Summary: Addressing Compiler Limitations

How Helix overcomes compiler challenges under dynamic shapes:

Aspect	**Existing Compilers**	**Helix Design Module**
Loop Trans.	Global schedule search; costly for dynamic shapes	**Top-Down Abstraction:** recursive template reduces search overhead
Tiling Config	Sample-based tuning; infeasible with large shape candidate space	**Hierarchical Cost Propagation:** end-to-end performance prediction enables efficient config selection
Low-level Impl.	Limited intrinsic-level optimization	**Bottom-Up Generation:** layer-wise construction ensures hardware-feasible high-perf candidates

Insight: Helix aligns each module with a specific compiler challenge, turning dynamic-shape compilation from sample-based tuning into structured, hardware-guided optimization.

Evaluation Setup

- **Platforms**: We evaluate Helix on four hardware setups:
  - Intel Xeon 8255c (x86 CPU, FP32)
  - AWS Neoverse N1 (ARM CPU, FP32)
  - NVIDIA A100 (CUDA Core mode, FP32)
  - NVIDIA A100 (Tensor Core mode, FP16)
- **Benchmarks**:
  - *Operator-Level:* 1389 dynamic-shape cases from DeepBench, Transformer, CNN, GNN
  - *Model-Level:* GPT2, BERT, LLAMA2, ResNet, AlexNet, GoogleNet
- **Baselines**:
  - *Vendor Libraries:* cuBLAS, cuDNN, oneDNN, ACL, ONNX, CUTLASS
  - *Dynamic-Shape Compiler:* DietCode (sample-driven SOTA)
- **Evaluation Metrics**:
  - Runtime Execution Latency
  - Offline Compilation Time

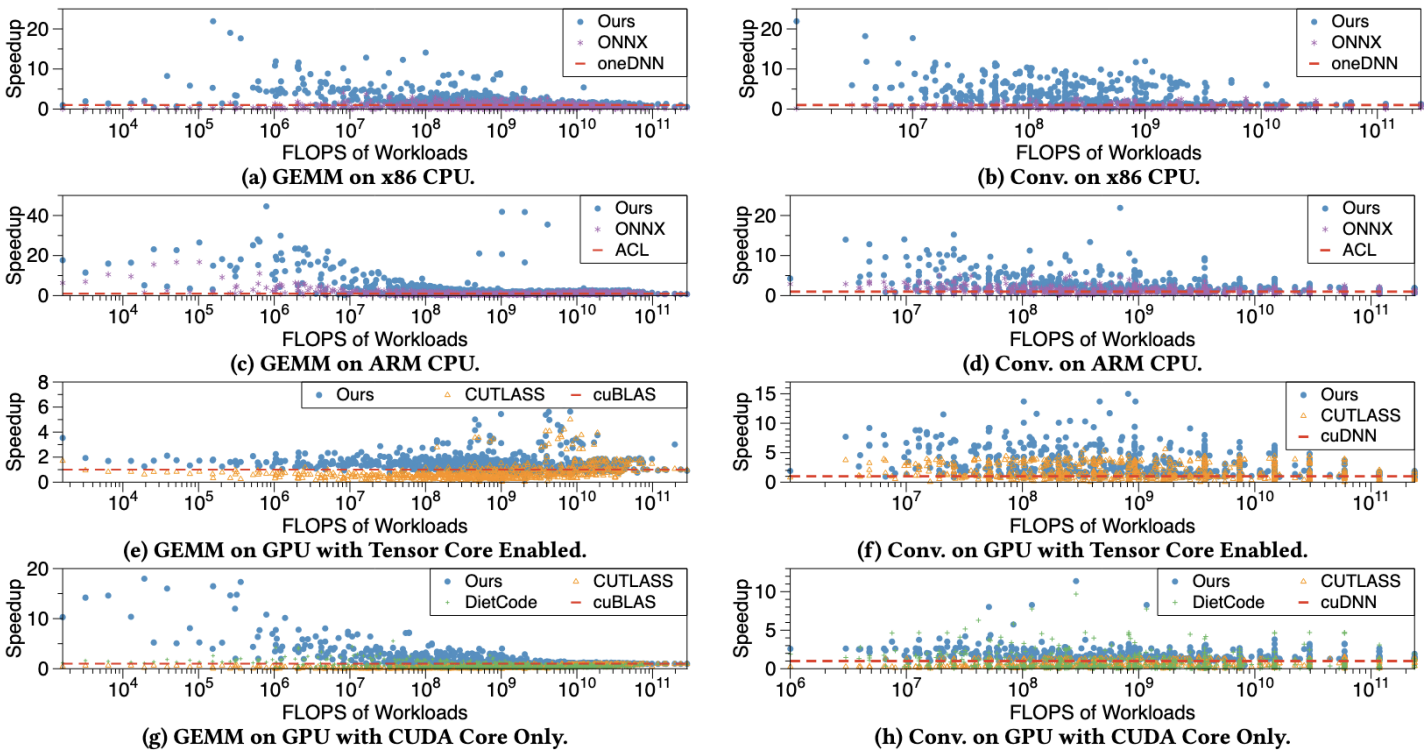
Evaluation Summary

Compilation Time:

- Helix generates and compiles **2332 CUDA Kernel candidates** in **529.6s**.
- DietCode requires **25 hours** for the same workload.
- $\rightarrow$ **174**$\times$ **faster** compilation than DietCode.

Operator-Level Performance:

- Consistently outperforms baselines on CPU and GPU.
- **6.00**$\times$ speedup on GPU (CUDA Core), and **5.37**$\times$ on x86 CPU.
- Over **90%** win-rate in most configurations.

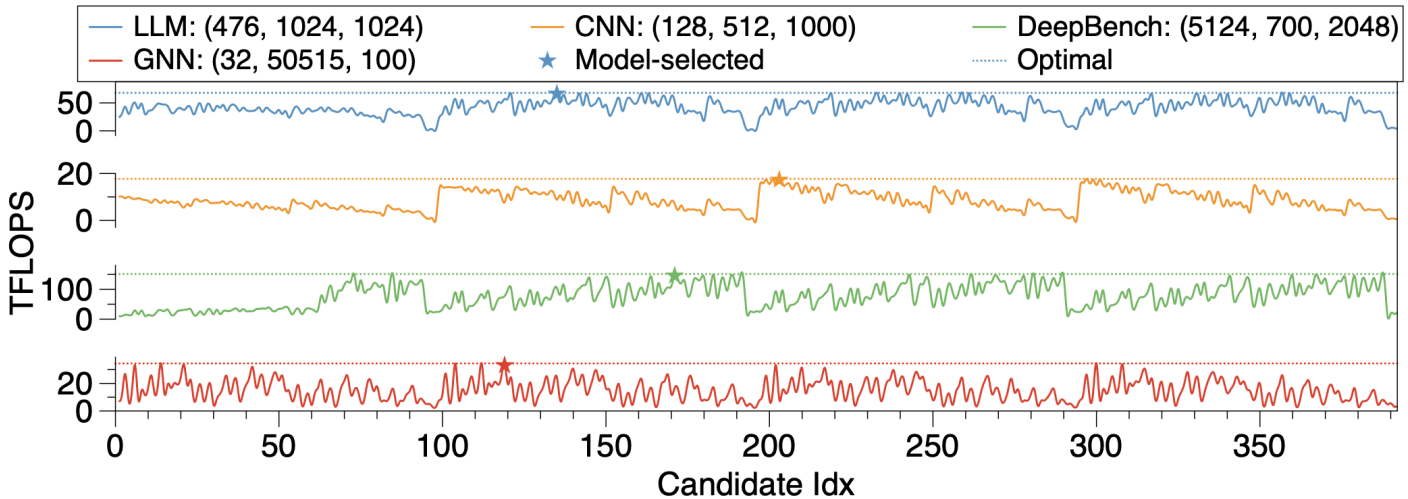


(a) GEMM on x86 CPU. (b) Conv. on x86 CPU.
(c) GEMM on ARM CPU. (d) Conv. on ARM CPU.
(e) GEMM on GPU with Tensor Core Enabled. (f) Conv. on GPU with Tensor Core Enabled.
(g) GEMM on GPU with CUDA Core Only. (h) Conv. on GPU with CUDA Core Only.

Model Accuracy and Runtime Efficiency

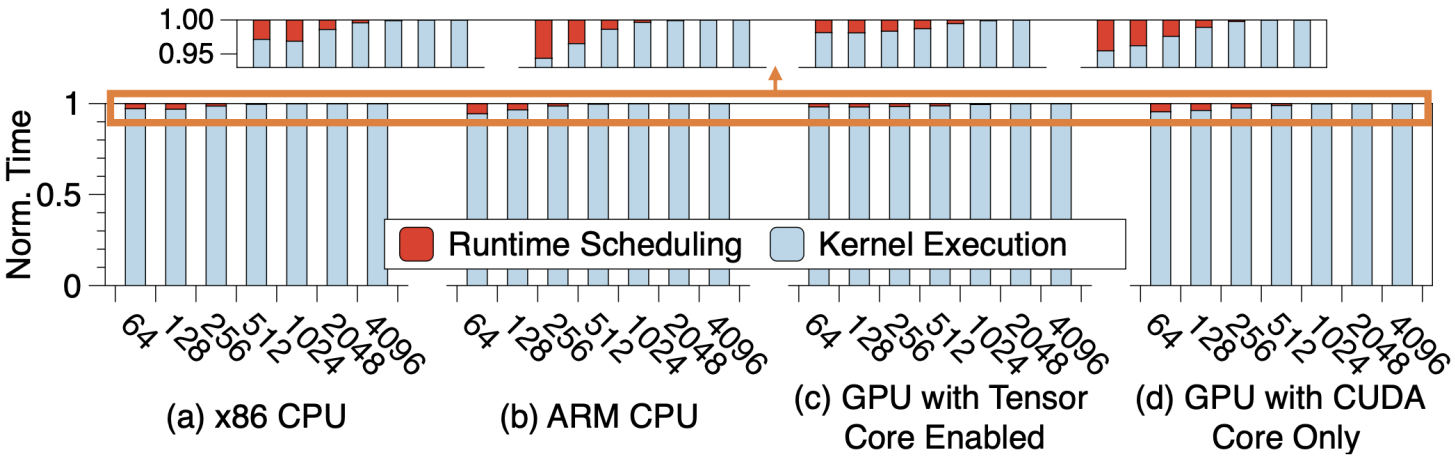
Accurate Model Selection:

- Selected strategies(stars) closely match the optimal(dotted lines) across diverse workloads.



Runtime Overhead:

- Across CPU and GPU, runtime scheduling takes less than 5% of total execution time.



Conclusion

Key Takeaways of Helix

- **Sample-Free Compilation:** Avoid per-shape sample-based tuning by decoupling shape logic, reducing compile time and engineering effort.
- **Architecture-Guided Codegen:** Leverage hardware information, build optimized kernels without end-to-end profiling.
- **Shape Generalization:** Achieve high performance across diverse inputs without manual tuning or sampling.

Comparative Summary:

Classification	**Sample-Free**	**Codegen Method**	**Tuning Support**	**Tuning Overhead**
Handwritten Libraries	✓	Hand-Tuned	No	N/A
Sample-Driven Compilers	×	Sample-Based	Yes	High
Helix (Ours)	✓	Arch-Guided	Yes	Low

Comparison with existing solutions.