TilePilot: A Lightweight Framework for Generating Optimized GPU Kernels

Jack Le Nathan Paek William Li

CS348K Spring 2025

TileLang: A GPU DSL for Structured, Composable Kernel Programming

🧩 Python-based DSL for GPU kernel development

📐 Separates dataflow from scheduling (e.g., memory layout, thread binding)

🔁 Kernels are written using tiles as first-class objects

⚡ Provides low-level primitives for:

- Memory allocation: T.alloc_shared(), T.alloc_fragment()
- Computation: T.gemm(), T.reduce(), T.copy()
- Scheduling: T.Pipelined, T.Parallel, T.annotate_layout()

📦 Compiler handles backend optimizations

Goal: Automatically generate efficient GPU kernels in a new low-resource DSL

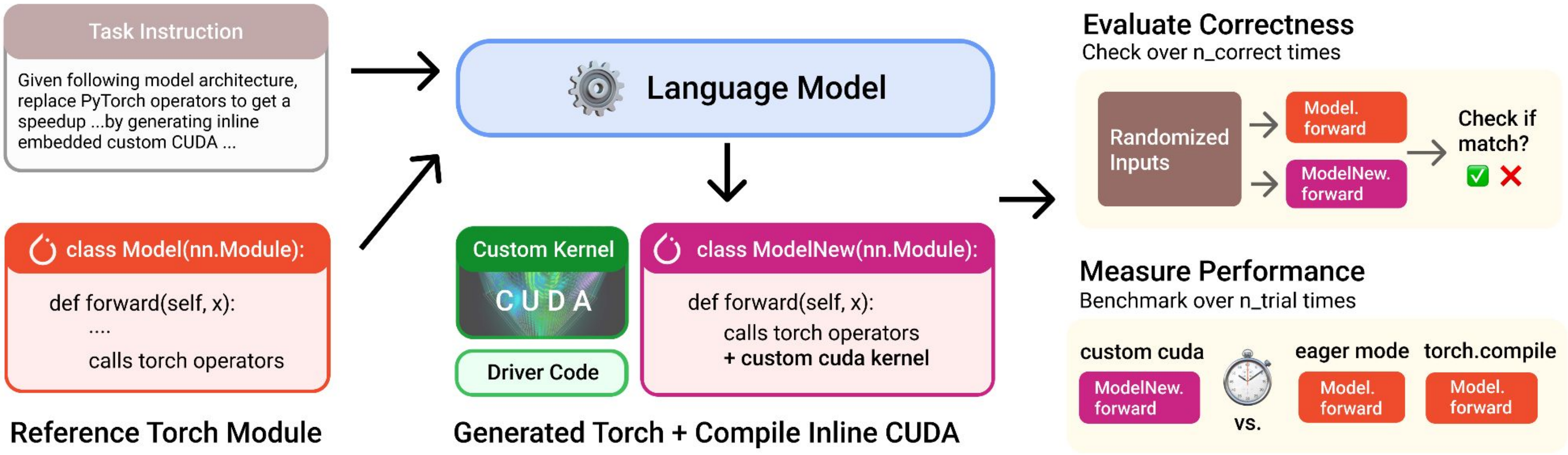
🎯 Input: PyTorch Models containing arbitrary operations

⚙️ Output: Fast, correct TileLang kernels to speed up the model

Must be:

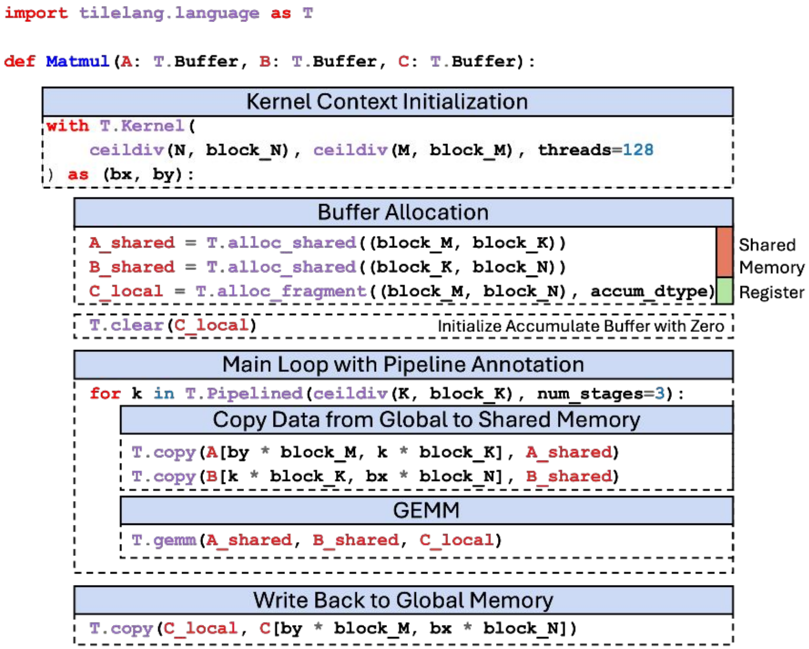
- ✅ Functionally correct
- 🚀 Performance competitive
- 💰 Cheap and easy to generate

KernelBench: Our Benchmark for Functional + Performance Evaluation



Source: scalingintelligence.stanford.edu/blogs/kernelbench

Challenge: TileLang is new, under-documented, and performance-critical



Source: github.com/tile-ai/tilelang

Insight: Kernel generation is often pattern recombination

💡 Most GPU optimizations aren't novel — they're reused patterns

- Coalesced Memory Access
- Shared Memory Tiling
- Thread Divergence Avoidance

Hypothesis: if we could retrieve relevant working examples and adapt them, we can teach a model a new language with very limited training data

TilePilot: Progressive Learning with RAG

Seed Stage

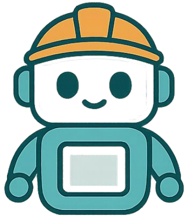
✍️ 37 handwritten TileLang kernels

Bootstrapping Stage

🔍 Retrieve and adapt similar kernels

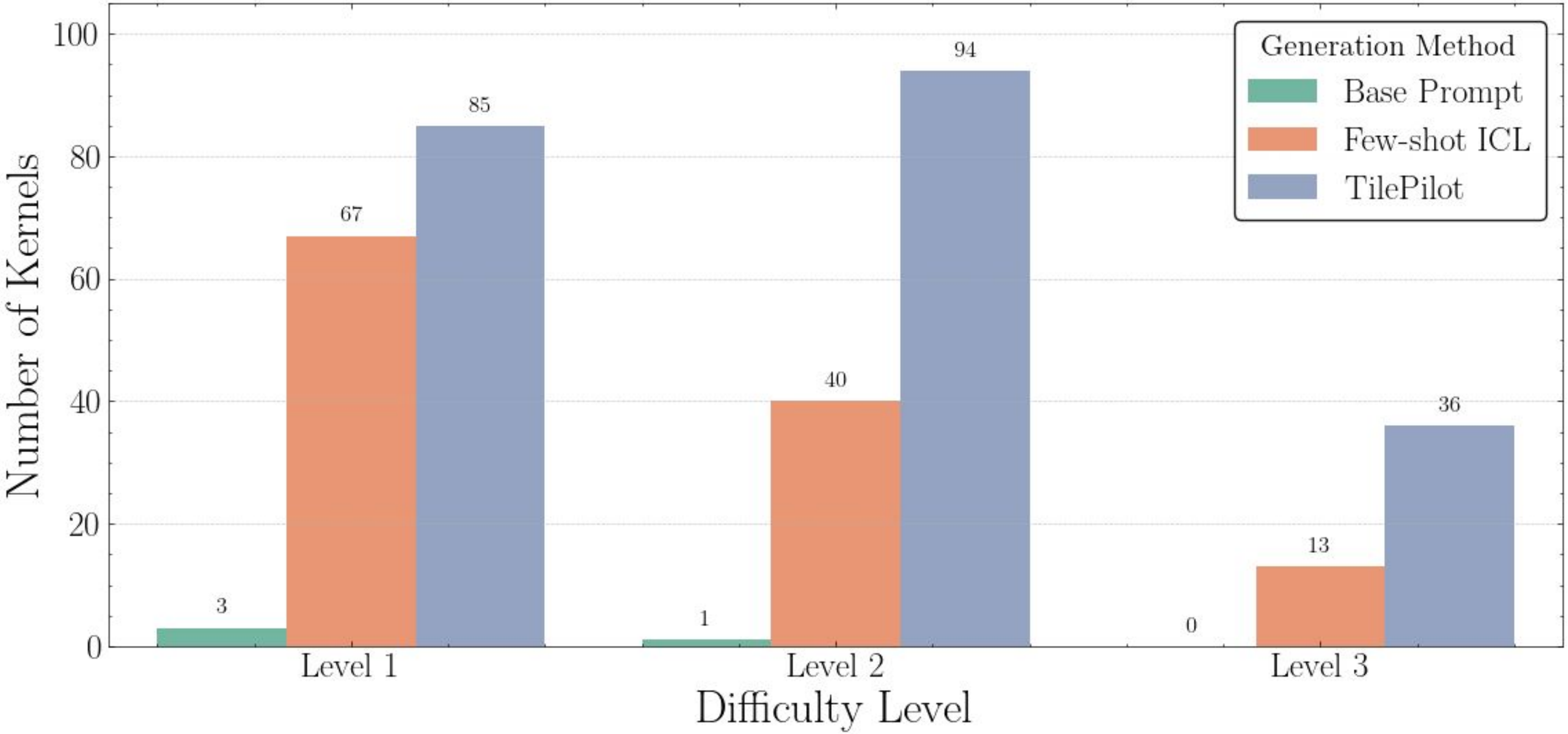
Optimization Stage

🔧 Fix slow or incorrect kernels

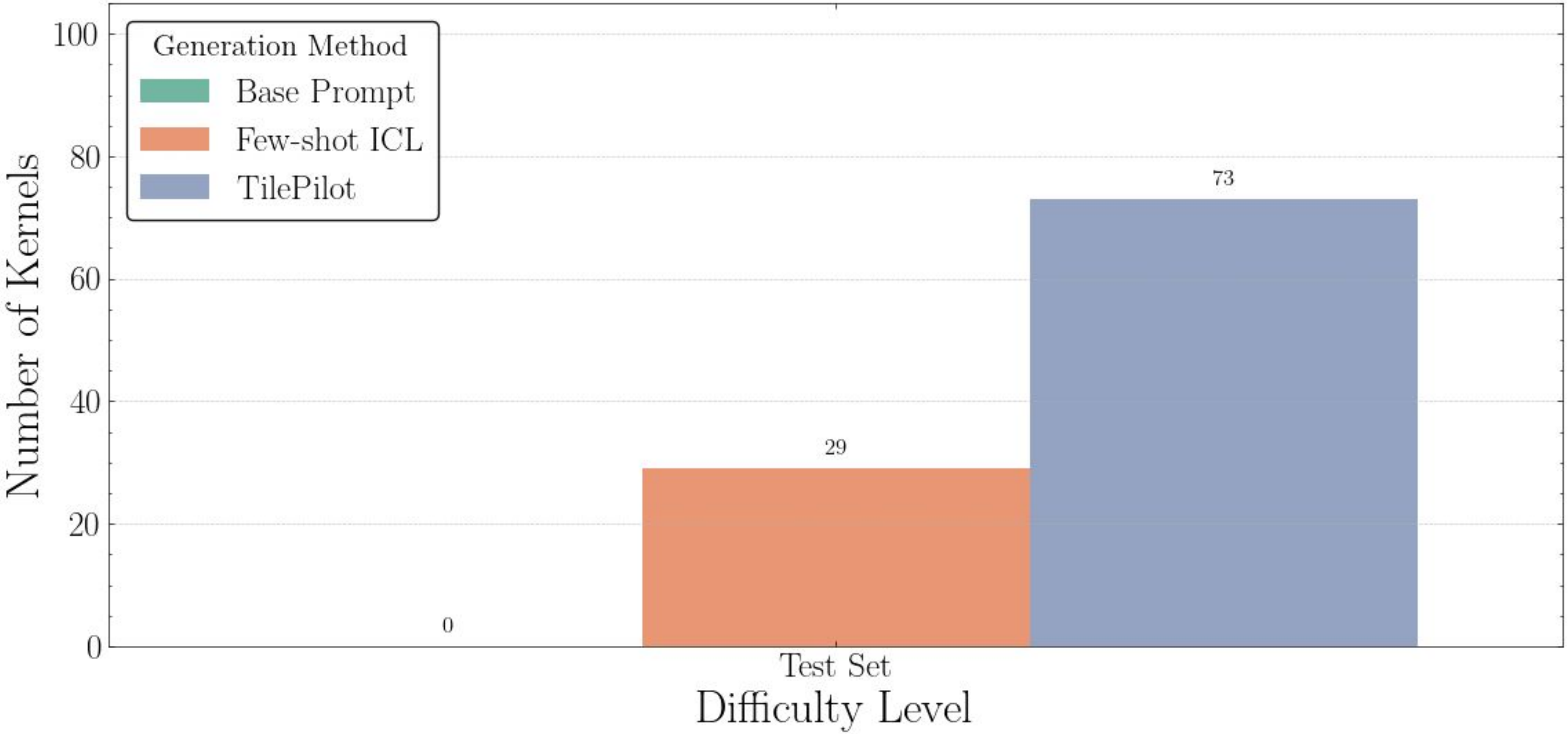


Results

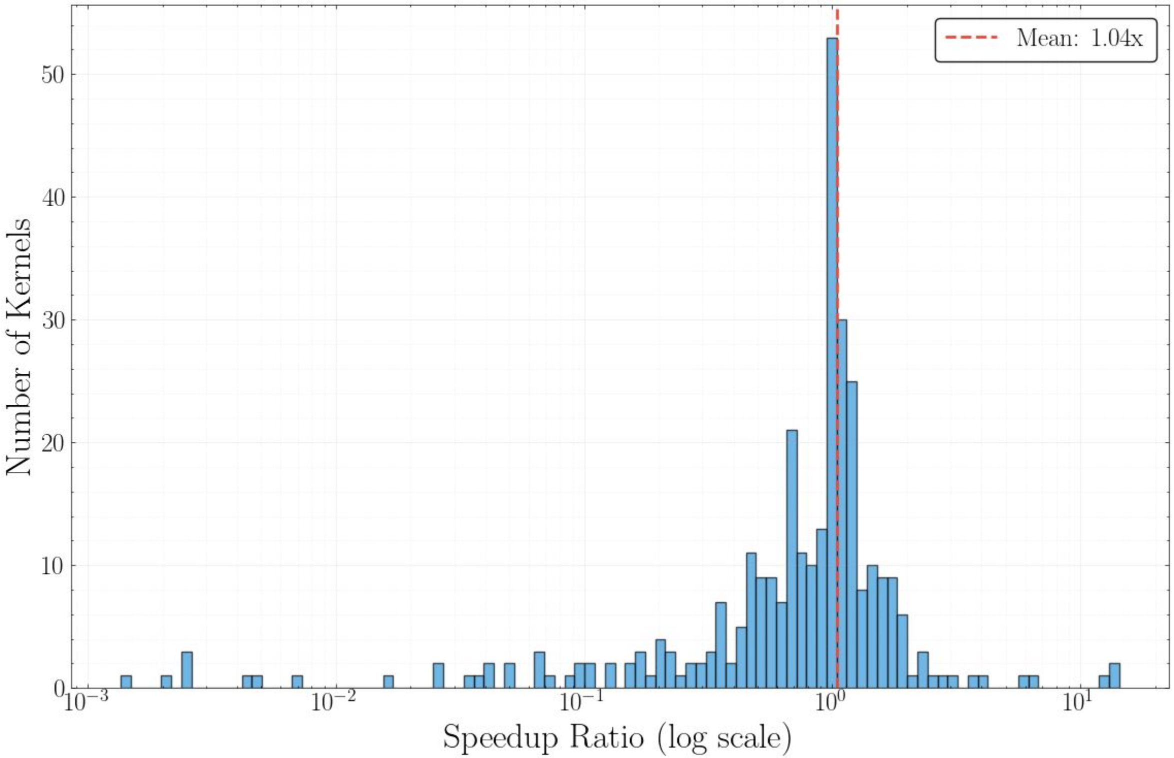
Correctness: TilePilot achieved 79% more successful generations vs. few-shot prompting



Correctness: TilePilot has greater zero-shot generalization on a held out test set

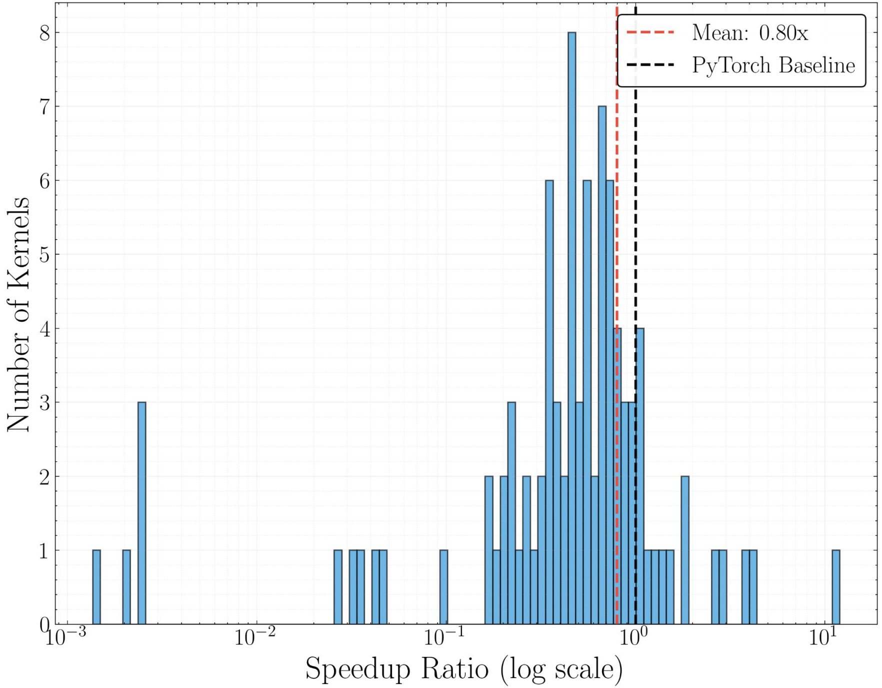


Performance: Many kernels outperform PyTorch baselines

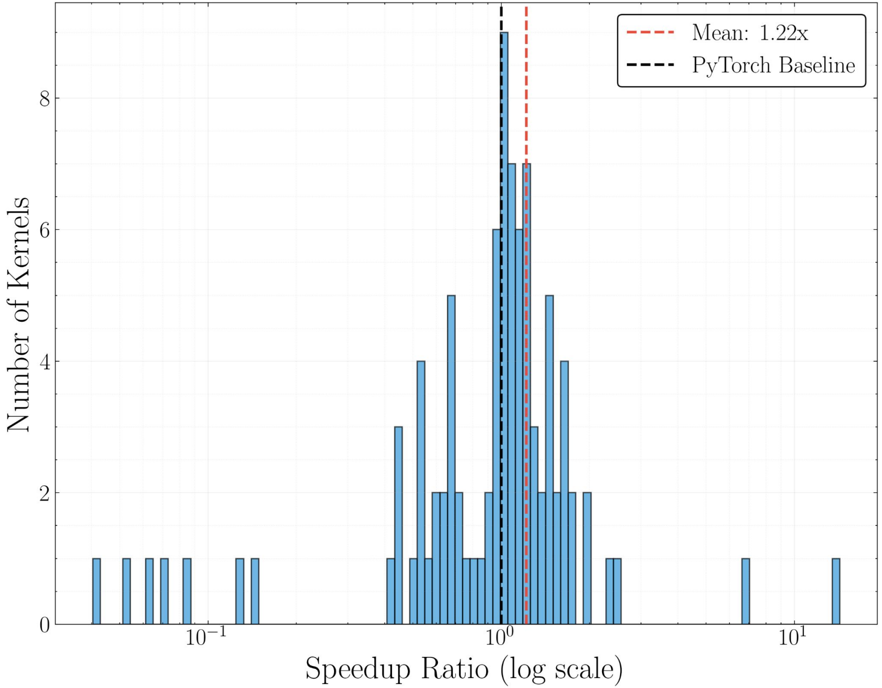


Speedup Ratios across All Generated Kernels

Performance: Many kernels outperform PyTorch baselines

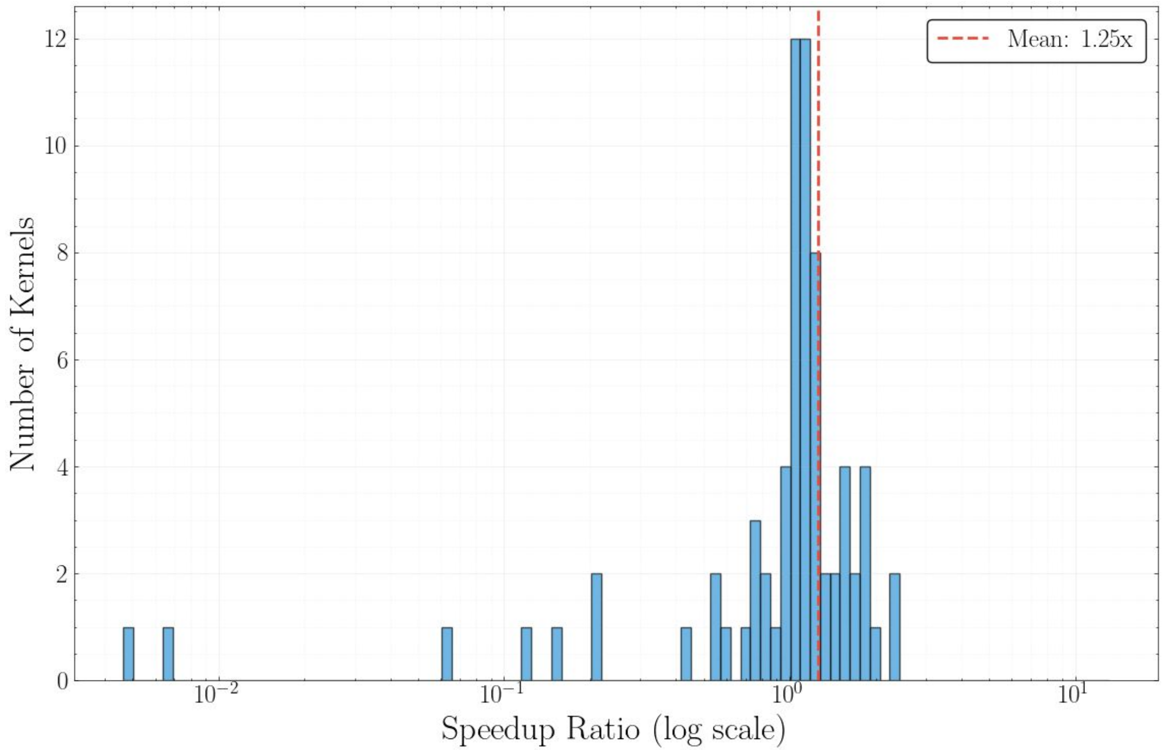


Level 1 Speedup Ratios



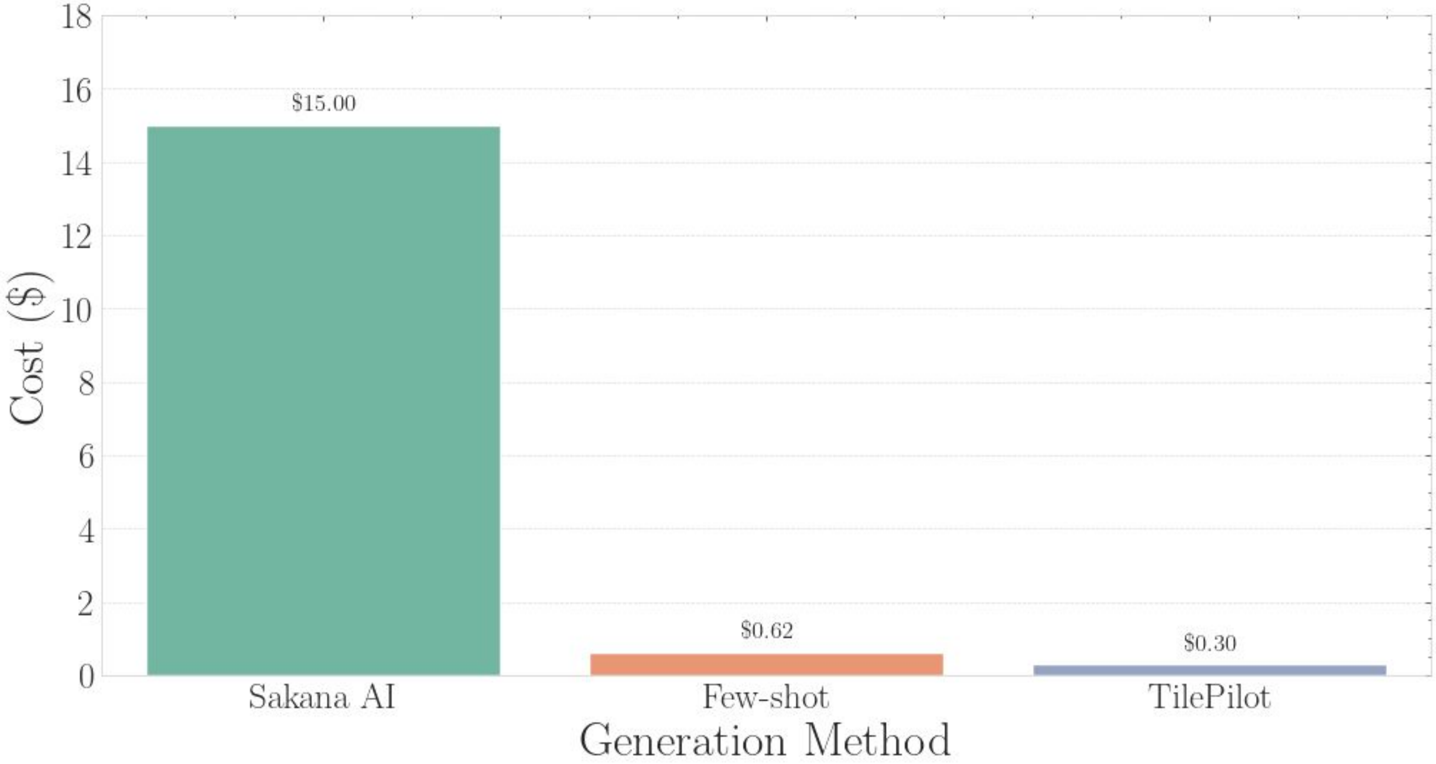
Level 2 Speedup Ratios

Performance: Many kernels outperform PyTorch baselines



Speedup Ratios on Held Out Test Set

Cost: TilePilot reduces kernel generation cost by 50x



TilePilot matches or beats CUDA baselines — despite no training data for TileLang

Comparison between TilePilot (TileLang) and KernelBench leaderboard (CUDA).

	TilePilot			Leaderboard		
Level	**Total**	**Mean**	**Speedup > 1.0x**	**Total**	**Mean**	**Speedup > 1.0x**
Level 1	83	0.80x	18%	65	0.88x	20%
Level 2	82	1.22x	62%	65	1.18x	52%
Level 3	29	0.85x	28%	26	0.82x	15%

Case Study: TilePilot can act like an LLM Compiler, discovering algorithmic simplifications

```python
class Model(nn.Module):
    ...
    def forward(self, x):
        x = torch.matmul(x, self.weight.T)  # Gemm
        x = torch.sum(x, dim=1, keepdim=True) # Sum
        x = x * self.scaling_factor  # Scaling
        return x
```

$$\mathrm{sum}(X \cdot W^T, \mathrm{dim} = 1) = X \cdot \mathrm{sum}(W, \mathrm{dim} = 0)$$

Case Study: TilePilot can act like an LLM Compiler, discovering algorithmic simplifications

```python
# Each thread processes one row of the input matrix X
with T.Kernel(T.ceildiv(batch_size, block_size), threads=block_size) as bx:
    tx = T.get_thread_binding(0)
    row = bx * block_size + tx
    if row < batch_size:
        # Initialize accumulator for this row
        acc = T.alloc_local((1,), accum_dtype)
        T.clear(acc)

        # This kernel computes X * sum(W, dim=0) efficiently by:
        # 1. Wsum is pre-computed as sum(W, dim=0) before kernel launch
        # 2. For each row i of X, compute dot product with Wsum
        # 3. This gives us sum(X[i,:] * sum(W, dim=0))
        # 4. Which is equivalent to sum(X[i,:] * W[:,:], dim=1)
        # 5. This avoids the full matrix multiplication X * W.T
        for k in T.serial(input_size):
            acc[0] += X[row, k] * Wsum[k]

        # Apply scaling factor (which includes the division by 2)
        # Final computation: (sum(X[row,:] * Wsum[:]) * scale_const)
        val = acc[0] * scale_const
        Out[row, 0] = T.Cast(dtype, val)
```

Where We Struggled, and Where We're Going Next

🧠 Level 1 is deceptively hard

🔄 TileLang's autotuner is too experimental – for now

📚 Level 3 needs advanced patterns (multi-kernel fusion), need more examples

⚠️ KernelBench can be unreliable → small input sizes crashes TMA

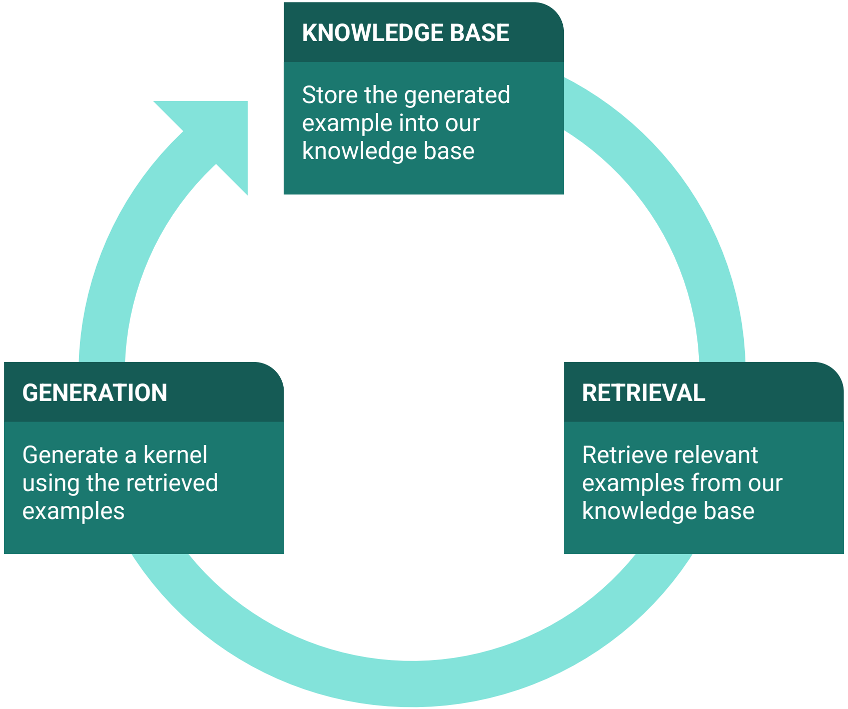
TilePilot enables fast, low-cost, compiler-like LLM kernel generation

💡 GPU kernels share

common structures, and

optimization patterns are

well defined



Training with TilePilot: Enabling Efficient Model Training 🚀

We use TilePilot to show that you don't need a massive model or dataset to get strong results for training models.

Start with 200 high-quality kernels

🧑‍💻 For each kernel, TracePilot generates a step-by-step reasoning trace:

🔍 Explains the "why" behind each optimization

🛠️ Details mapping, fusion, and performance choices

🧩 Captures expert logic, not just code

📚 Result:

A dataset of expert reasoning traces paired with kernels

🤖 Use for Training:

Train models to not just copy code, but to reason and optimize like an expert