

# TilePilot: A Lightweight Framework for Generating Optimized GPU Kernels

---

Jack Le

Nathan Paek

William Li

CS348K Spring 2025

1

JACK

Hi everyone, we're excited to present our project, TilePilot.

This is a system we built to answer the question: can large language models automatically generate performant GPU kernels, and can do them in a low resource DSL called TileLang.

We'll walk you through the motivation, our approach, and some pretty surprising results.

I'm Jack, and this is Nathan and William.

# 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

2

JACK

TileLang is a DSL to help you write optimized GPU kernels with a Pythonic interface. It treats tiles as first-class objects, which has been shown to be the most performant way to write kernels.

It provides low-level primitives that abstracts the underlying hardware details, making it much easier for engineers to program GPUs.

---

To understand our project, it's important to know what TileLang is. TileLang is a new Python-based domain-specific language that simplifies writing GPU kernels. It is built on top of Apache TVM, which we talked about when we talked about Halide. TileLang decouples what the kernel computes — the dataflow — from how it runs — the scheduling.


Kernels in TileLang are built around *tiles*, which are structured chunks of data assigned to warps or threads. You have explicit control over memory using primitives like `T.alloc_shared()` for fast shared memory or `T.alloc_fragment()` for registers.

On the compute side, TileLang gives you building blocks like `T.copy()` for parallel data movement, `T.gemm()` for matrix multiply, and `T.reduce()` for reductions. For




The goal of the language is to make high-performance kernel programming composable and safe — while letting the compiler handle a lot of the heavy lifting under the hood.

## 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

3

### WILLIAM

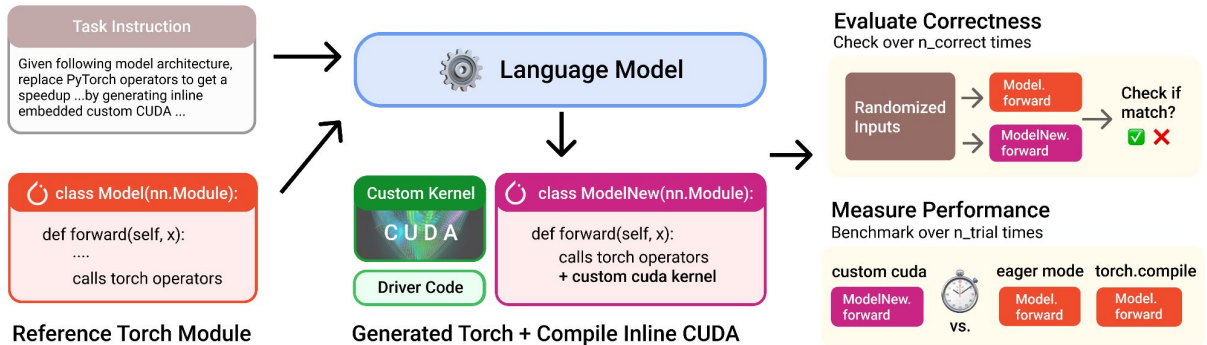
Our high-level goal was to generate TileLang kernels from PyTorch operations — automatically.

The primary research question was: Can retrieval-augmented generation and composition produce functionally correct and performant GPU kernels in a low-resource domain-specific language more efficiently and cost-effectively than brute-force sampling methods?

We defined success via three questions:

1. How often are the generated kernels functionally correct?
2. How do they perform relative to PyTorch and CUDA?
3. And how much does it cost to generate them compared to baseline approaches like brute-force sampling?

# KernelBench: Our Benchmark for Functional + Performance Evaluation



Source: [scalingintelligence.stanford.edu/blogs/kernelbench](https://scalingintelligence.stanford.edu/blogs/kernelbench)

4

## NATHAN

We used KernelBench as our benchmark suite. It contains 250 real GPU kernel tasks across three difficulty levels.

- Level 1 (100 tasks): Single-kernel operators like convolutions and matrix multiplies
- Level 2 (100 tasks): Simple fusion patterns (e.g. conv + bias + ReLU)
- Level 3 (50 tasks): Full ML architectures like VGG or AlexNet

It tests both correctness and speed, and helped us measure how well TilePilot performs across a range of challenges.

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

```
import tilelang.language as T

def Matmul(A: T.Buffer, B: T.Buffer, C: T.Buffer):

    Kernel Context Initialization
    with T.Kernel(
        ceildiv(N, block_N), ceildiv(M, block_M), threads=128
    ) as (bx, by):

        Buffer Allocation
        A_shared = T.alloc_shared((block_M, block_K))
        B_shared = T.alloc_shared((block_K, block_N))
        C_local = T.alloc_fragment((block_M, block_N), accum_dtype)
        T.clear(C_local)
        Initialize Accumulate Buffer with Zero

        Main Loop with Pipeline Annotation
        for k in T.Pipelined(ceildiv(K, block_K), num_stages=3):
            Copy Data from Global to Shared Memory
            T.copy(A[by * block_M, k * block_K], A_shared)
            T.copy(B[k * block_K, bx * block_N], B_shared)

            GEMM
            T.gemm(A_shared, B_shared, C_local)

            Write Back to Global Memory
            T.copy(C_local, C[by * block_M, bx * block_N])
```

Source: [github.com/tile-ai/tilelang](https://github.com/tile-ai/tilelang)

NATHAN

The **main challenge** was that TileLang only launched six weeks ago, so there’s barely any training data, examples, or documentation.

In fact, this figure from their GitHub is one of the only examples of a performant kernel available on the internet.

And unlike general code generation, GPU kernels have tight performance constraints, requiring deep hardware-aware optimizations.

Figure 2 (a) demonstrates how multi-level tiling leverages different memory hierarchies (global, shared, and registers) to optimize bandwidth utilization and reduce latency.

Figure 2 (b) showcases how the Python-like syntax of TileLang allows developers to

reason about performance-critical optimizations within a user-friendly programming model.

## 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

6

WILLIAM

Key insight for our approach: we realized that most GPU kernels reuse the same patterns.


We hypothesized that if we could **retrieve relevant working examples** and adapt them, we could 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



9

JACK

We built TilePilot in three phases.

1. First, we wrote 37 high-quality seed kernels that covered **diverse fundamental operations** like matmuls
2. Then, we implement a RAG system using DSPy that treats our kernels as a **dynamic knowledge base**
  - a. **self-evolving** and **continually growing** database of correct TileLang kernels
3. Finally, we use a structured multi-turn loop where an LLM analyzes potential bottlenecks and optimizes the kernel based on those suggestions.

In this presentation we will focus on the seed and bootstrapping stages.

We explored the optimization stage and **have a working prototype**, but were **not able to get it reliable enough** to justify the costs of running it on our entire dataset.

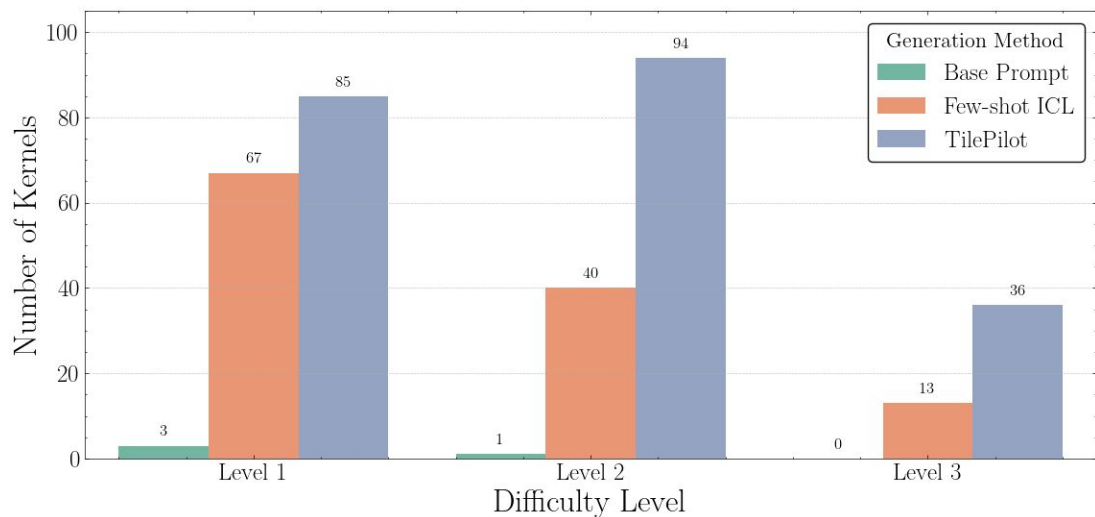
---

- a. provide the initial knowledge base for retrieval and establish patterns for the LLM to learn from.
  - b. each kernel includes detailed annotations explaining the mapping from PyTorch operations to TileLang constructs.
2. Then, we implement a retrieval-augmented generation system using DSPy that treats our growing kernel collection as a dynamic knowledge base. For each generation:
  - a. We analyze the input PyTorch operation to extract key semantic features like operations types and computational patterns.
  - b. We then embed this as a query and retrieves the most semantically similar existing kernels based on cosine similarity in the embedding space.
  - c. These retrieved kernels serve as contextual examples for the LLM.
  - d. The kernel then synthesizes a new kernel by adapting relevant patterns from retrieved examples.
3. Finally, if performance was poor, we used a structured optimization loop to refine the kernel. We employ multi-turn conversations where the LLM analyzes potential bottlenecks and optimizes the kernel based on those suggestions.

In this presentation we will focus on the bootstrapping stage since that was our main contribution. We explored the optimization stage and have a working prototype, but were not able to get it reliable enough to justify the costs of running it on our entire dataset.

# Results

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



9

WILLIAM

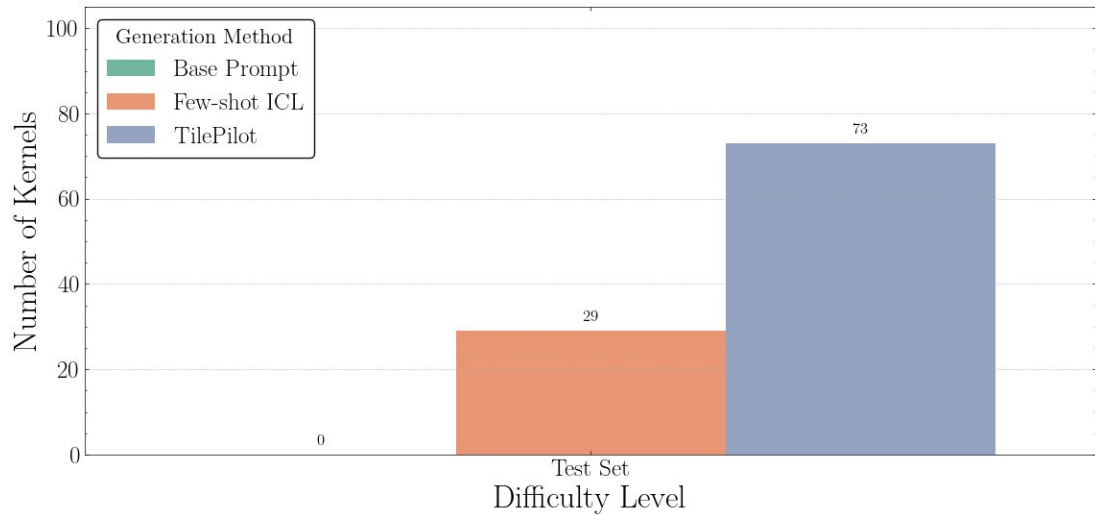
We **tried three methods** to enable LLMs to write kernels in TileLang.

1. Direct generation using OpenAI's o3 with only TileLang documentation as context
2. In-context learning utilizing 5 carefully selected diverse examples
3. Our proposed approach using retrieval-augmented generation with a knowledge base of optimization patterns

In total, few shot prompting was able to generate 120 total kernels. With our system, we were able to generate **215 kernels, representing a 79% increase**.

One interesting note: we saw the biggest gains in Level 2, where most of the problems were related kernel fusion. This shows how our RAG based approach really shines in comparison to standard methods.

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



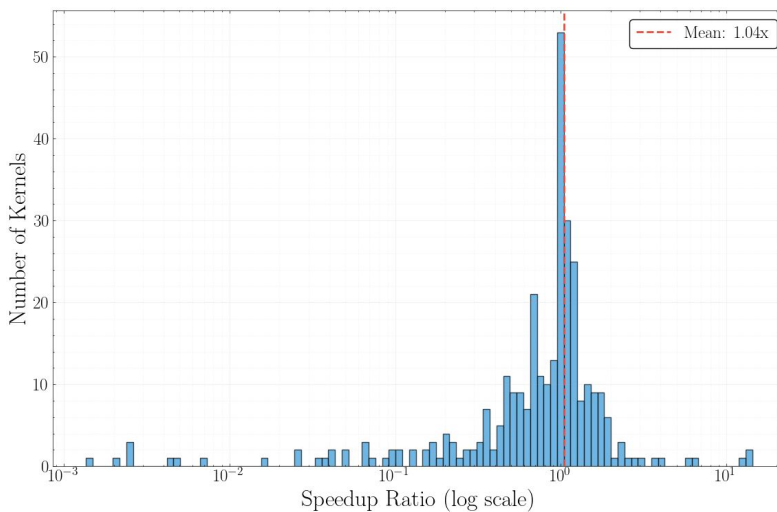
10

WILLIAM

We had a **held out test set** that was not used in RAG or anything else

As you can see, our method is able to **significantly outperform** both other methods

## Performance: Many kernels outperform PyTorch baselines



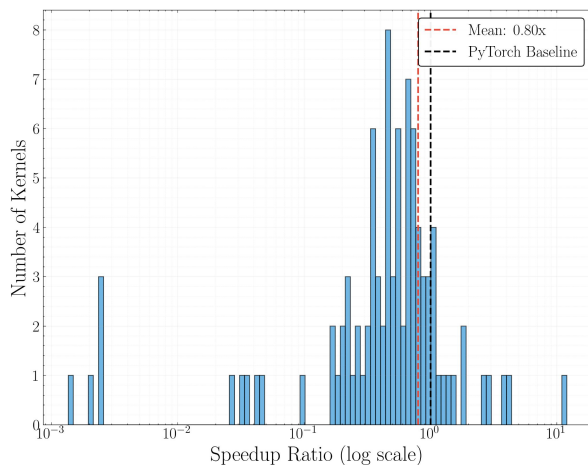
Speedup Ratios across All Generated Kernels

JACK

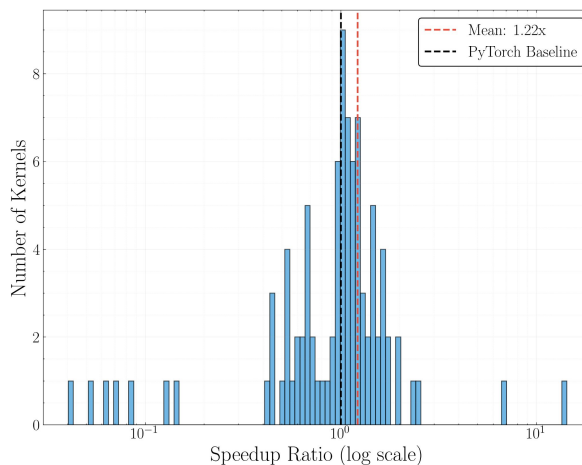
This is a log scale graph of speedup ratios, measured as speedup over the PyTorch baseline, across all generated kernels.

The overall average speedup across all kernels generated by our system is 1.04x, meaning that on average we beat PyTorch.

## Performance: Many kernels outperform PyTorch baselines



Level 1 Speedup Ratios



Level 2 Speedup Ratios

12

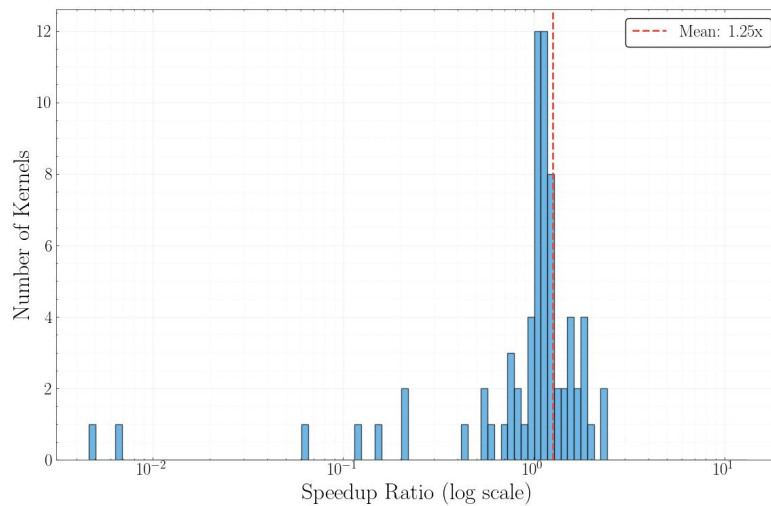
JACK

If we go further in detail into levels 1 and two, we see:

62% of Level 2 kernels beat PyTorch, with a mean speedup of 1.22x and a max of 14x

18% of Level 1 kernels beat PyTorch, with a mean speedup of 0.80x and a max of 12x

## Performance: Many kernels outperform PyTorch baselines



Speedup Ratios on Held Out Test Set

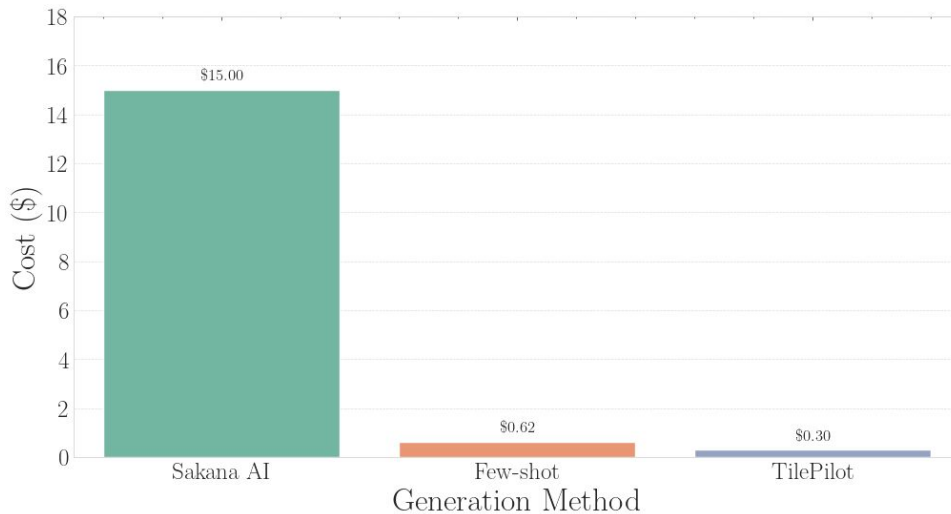
13

JACK

Running on the held out test set, which contains **100 new problems from all levels** that we acquired from the KernelBench authors, we were able to reach a mean speedup of 1.25x, with 50 out of 72 kernels (or 69%) beating the PyTorch baseline.



## Cost: TilePilot reduces kernel generation cost by 50x



14

NATHAN

Our approach achieves **remarkable cost efficiency** compared to other approaches.

- **Writing in DSL for performant kernels**
- **RAG for picking only the most relevant examples, thereby having shorter and more focused prompts**

- 
- Sakana AI's AI CUDA engineer and scientist costs upwards of \$15.00 per successful kernel due to their brute force iterative sampling approach.
  - The naive few-shot ICL approach costs \$0.62 per successful kernel, primarily due to the fact that the prompt needed to be much longer in order to be able to reliably generate a diverse range of kernels.
  - In comparison, thanks to RAG and example reuse, TilePilot costs just 30 cents per successful kernel, while still achieving a higher success rate than the naive few-shot ICL approach.

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

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

Level	TilePilot			Leaderboard		
	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%

WILLIAM

**We run on low resource language, matches and outperforms with leaderboard which uses widely available CUDA**

-----  
We also compared TilePilot to the official KernelBench leaderboard. Their results have models generating CUDA — a mature language with massive training data and decades of optimization research. Ours are in TileLang, a DSL that was publicly released 6 weeks ago and has almost zero representation in LLM training.

Despite that, TilePilot is competitive across the board — and even outperforms the CUDA solutions on Level 2 tasks. We also generated more total kernels across all levels, which shows the practicality and generalizability of our approach.

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

```
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
```

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

16

JACK

**In some cases, TilePilot acted like a compiler.**

This model performs a matrix multiplication followed by a summation.

With some algebraic reasoning, we can replace the matrix multiplication for a cheaper matrix vector multiplication, reducing the time complexity from  $O(N^3)$  to  $O(N^2)$

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

```
# 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)
```

17

JACK

Given this task, the TilePilot model was able to reason about that optimization and implemented the optimized kernel, producing a significant speedup compared to before.

This shows how we have a **continually self improving** dataset of kernels.

## 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

18

JACK

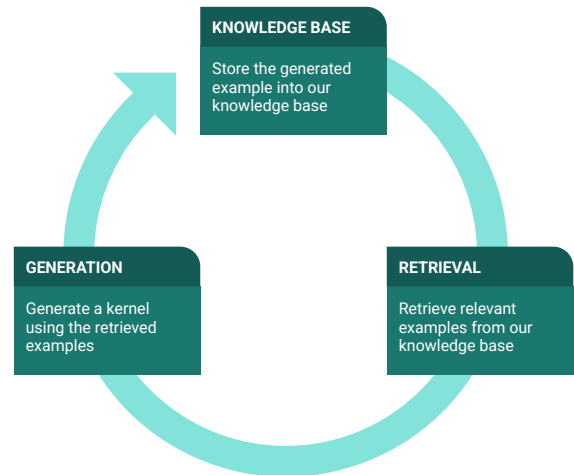
Let's talk about what didn't go well.

Level 1 tasks like matmul are surprisingly hard. These tasks look simple, but PyTorch uses vendor-optimized kernels that have been improved over many years — they're extremely fast, and TileLang will not help improve over them.

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



💡 GPU kernels share  
common structures, and  
optimization patterns are  
well defined



19

JACK

To wrap up: TilePilot shows that LLMs can go beyond brute force — discovering optimizations in a new DSL with high correctness, strong performance, and huge cost savings. We successfully generated kernels for 194 of the 250 problems in KernelBench, and a total of 225 kernels if we include duplicated kernels from multi-turn optimizations.

Thanks for listening — we're happy to take questions!