MatMul (Matrix Multiplication)

Matmul is a compute bound operation because its performance is limited by the GPU’s compute units rather than memory speed. In other words, the GPU spends most of its time doing calculations, not waiting for data.

In matrix multiplication, each output element is produced by taking a row from A and a column from B, multiplying corresponding elements, and summing them.

$$C_{i,j} = \alpha \sum_{k} A_{i,k} \cdot B_{k,j}$$

In our work, we set $\alpha = 1$

Why it’s compute bound:

This approach is compute bound because, once the tiles are loaded, the gpu spends most of its time performing multiplication and additions, called FLOPS.

To quantify this compute-bound performance, we measure TFLOPs(Tera floating point operation per second) using the following metric:

$$TFLOPs = \frac{2\, x\, m\, x\, n\, x\, k}{latency\ [seconds] x 10^{12}}$$

See the code: here

- 2.0 * m * n * k: counts 2 FLOPs per multiply-add in matrix multiplication.
- Dividing by latency gives the achieved TFLOPS.
- Dividing by 10^{12} converts to teraFlops

Triton (Tutorial Version):

In this blog, we use the simplified Triton tutorial implementation, which operates on full rows or full columns. This version is easier to follow because it skips tiling and partial accumulation.

How Triton Matmul works:

Consider Figure 1, to compute 9 blocks in a row of matrix C, a native approach would be to load the entire matrix B (all 81 blocks), and the full corresponding row of blocks from A. Instead, Triton loads a small number of tiles in a smart way to compute 9 output blocks efficiently. For example, in the bottom part of Figure 1, we load 27 blocks from A and 27 blocks from B to compute the 9 blocks of C. By doing this, we avoid 36 unnecessary memory loads, which reduces memory traffic and improves performance.

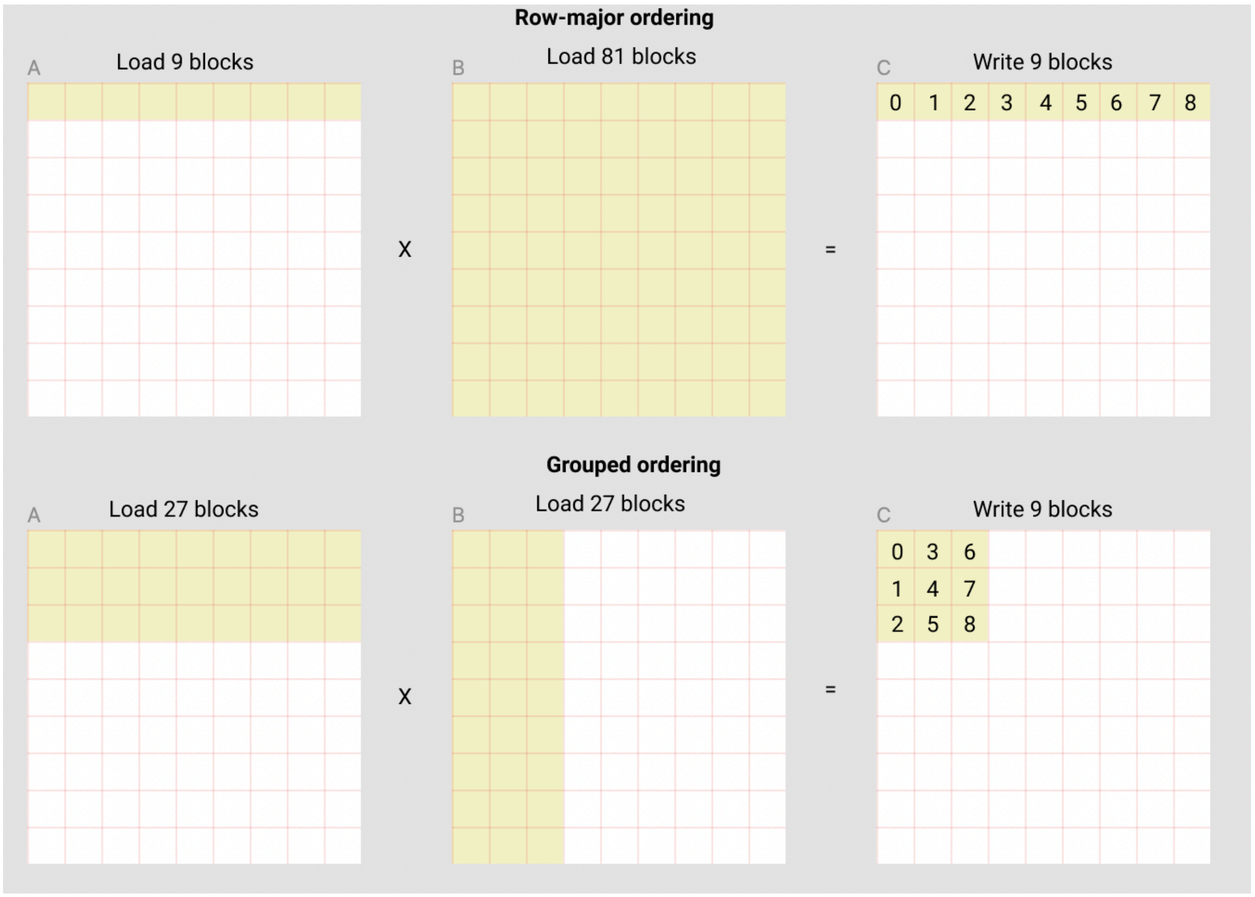


Figure 1: Memory Access for Triton Matrix Multiplication (top: naive approach, bottom: optimized approach)

How does this implement?
We split the input matrices into tiles of size $BLOCK\_SIZE\_M$, $BLOCK\_SIZE\_N$ and $BLOCK\_SIZE\_K$. Tiles along the M dimension can be grouped together using $GROUP\_SIZE\_M$. For example, if $GROUP\_SIZE\_M = 3$, each program instance processes 3 tiles along M at the same time.

The algorithm then loops over the tiles in K dimensions:

1. Load the corresponding tile of A and tile of B for each of the tiles in the program instance group into on-chip memory.
2. Multiply and accumulate the loaded tiles into a local accumulator for each output tile. These are partial results, summed over the K dimension.
3. Repeat this process for all tiles along K to compute the full results for the assigned output tiles.

After completing all K blocks, write the final output tiles for the entire group to matrix C.

Autotuning: Block sizes, group size, and number of warps are manually tuned, as shown in the Triton MatMul tutorial.

TorchInductor

We use torch.compile to compile pytorch operations into optimized kernels. In the example, we wrap torch.matmul in compiled function with 2 modes:

1. Default mode: compiles quickly with general optimizations, producing good performance with minimal setup.
2. Max autotune mode: profiles multiple implementations and selects the fastest kernel for the target hardware.

Link to our code: torch.compile.

KernelLLM

We experimented with multiple approaches, but the one that worked best for us was using a prompt with the generate_triton method.

```None
TORCH_ROCM_AOTRITON_ENABLE_EXPERIMENTAL=1
TORCH_CUDA_ALLOC_CONF=expandable_segments:True
```

To set up the environment:

1. Clone the kernelLLM repository.
2. Install Pytorch for ROCm:

```None
pip3 install torch torchvision --index-url
https://download.pytorch.org/whl/rocm6.4
```

3. Installed kernel LLM packages.
4. Create and run the prompt:

Challenges encountered:

- **External Triton Kernels**
  The code relies on extern_kernels, which makes it dependent on Triton's vendor functions.
- **Grid Specification Issue**
  The original code used:

```Python
from torch._inductor.runtime.triton_heuristics import grid
```

```python
triton_poi_fused__to_copy_0[grid(65536)](arg0_1, buf0, 65536, XBLOCK=512,
num_warps=4, num_stages=1)
```

This did not work. To fix it, we had to explicitly define the grid size for the kernel execution:

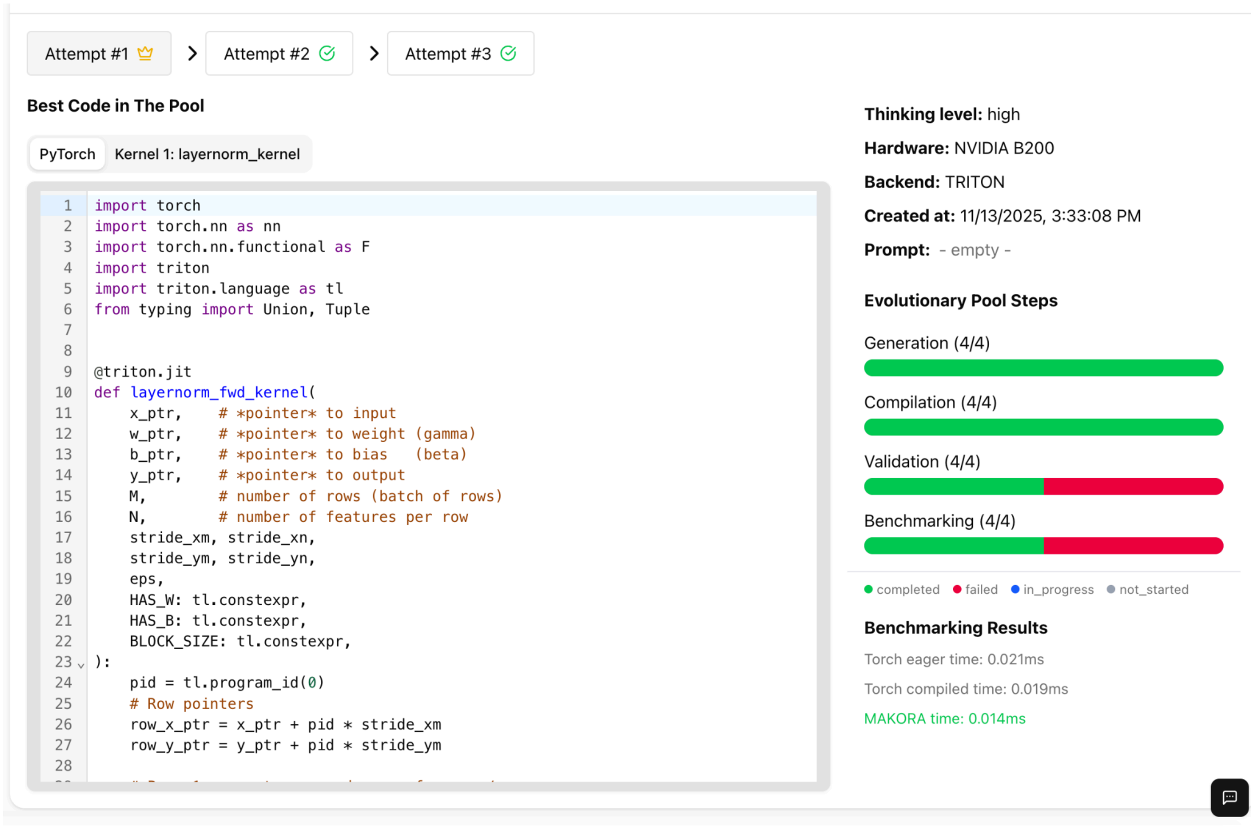
```python
grid0 = lambda META: (triton.cdiv(num_elements0 + META['XBLOCK0'] - 1,
META['XBLOCK0']), )
triton_poi_fused__to_copy_0[grid0](arg0_1, buf0, num_elements0)
```

- **Tensor Data Type Mismatch:**
  Buffers were initially torch.float32, which caused compilation/runtime errors. Changing them to torch.bfloat16 fixed the problem.
- **Shape Handling and autotuning**
  The current implementation has hard-coded shape handling. Any engineer wanting to generalize this code will need to manually adjust shapes and add autotuning.
- **Redundant Code**
  There are pieces of code in the repo that are unset or have no effect. These could be removed to simplify the implementation.

We decided to generalize this kernel and implement autotuning to find the optimal configuration for performance, as the original kernel was implemented without autotuning support.

Makora Generate (formerly Mako)

We created an account in mako.dev and selected Triton as the backend, and chose the target GPU, and set the kernel generation to high thinking level. Using the same prompt as in KernelLLM, the platform generated the kernel code. Mako.dev then automatically compiled, validated, and benchmarked the kernel against PyTorch eager mode to ensure correctness and measure performance.



```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import triton
import triton.language as tl
from typing import Union, Tuple


@triton.jit
def layernorm_fwd_kernel(
    x_ptr,    # *pointer* to input
    w_ptr,    # *pointer* to weight (gamma)
    b_ptr,    # *pointer* to bias   (beta)
    y_ptr,    # *pointer* to output
    M,        # number of rows (batch of rows)
    N,        # number of features per row
    stride_xm, stride_xn,
    stride_ym, stride_yn,
    eps,
    HAS_W: tl.constexpr,
    HAS_B: tl.constexpr,
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(0)
    # Row pointers
    row_x_ptr = x_ptr + pid * stride_xm
    row_y_ptr = y_ptr + pid * stride_ym

```

Thinking level: high
Hardware: NVIDIA B200
Backend: TRITON
Created at: 11/13/2025, 3:33:08 PM
Prompt: - empty -

Evolutionary Pool Steps

Generation (4/4)
Compilation (4/4)
Validation (4/4)
Benchmarking (4/4)

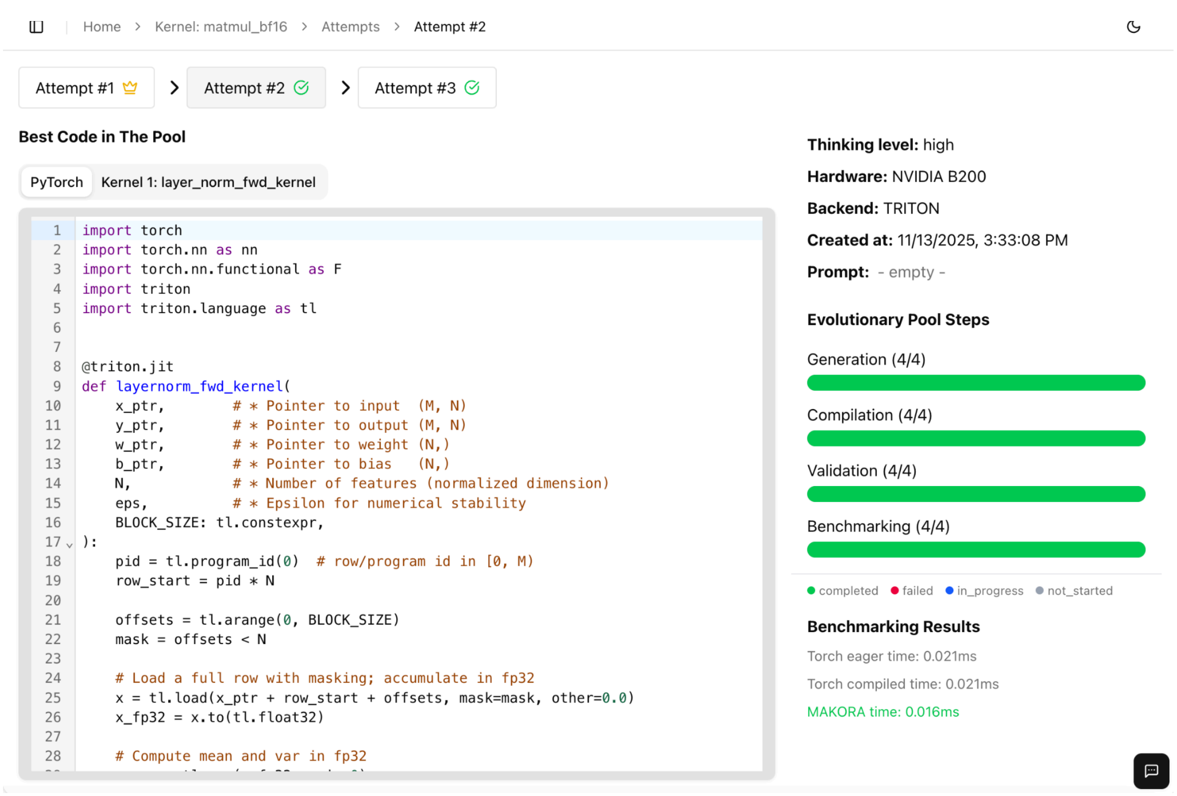
● completed ● failed ● in_progress ● not_started

Benchmarking Results

Torch eager time: 0.021ms
Torch compiled time: 0.019ms
MAKORA time: 0.014ms

Figure 2: 1st attempt matmul_bf16 on mako.

As you can see from the figure, at the first attempt it fails 2/4 for validation and benchmarking. At the second and third attempt it passes, but the second one achieves better performance.



Home > Kernel: matmul_bf16 > Attempts > Attempt #2

Attempt #1 > Attempt #2 > Attempt #3

Best Code in The Pool

PyTorch | Kernel 1: layer_norm_fwd_kernel

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import triton
import triton.language as tl


@triton.jit
def layernorm_fwd_kernel(
    x_ptr,        # * Pointer to input  (M, N)
    y_ptr,        # * Pointer to output (M, N)
    w_ptr,        # * Pointer to weight (N,)
    b_ptr,        # * Pointer to bias   (N,)
    N,            # * Number of features (normalized dimension)
    eps,          # * Epsilon for numerical stability
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(0)  # row/program id in [0, M)
    row_start = pid * N

    offsets = tl.arange(0, BLOCK_SIZE)
    mask = offsets < N

    # Load a full row with masking; accumulate in fp32
    x = tl.load(x_ptr + row_start + offsets, mask=mask, other=0.0)
    x_fp32 = x.to(tl.float32)

    # Compute mean and var in fp32
```

Thinking level: high
Hardware: NVIDIA B200
Backend: TRITON
Created at: 11/13/2025, 3:33:08 PM
Prompt: - empty -

Evolutionary Pool Steps

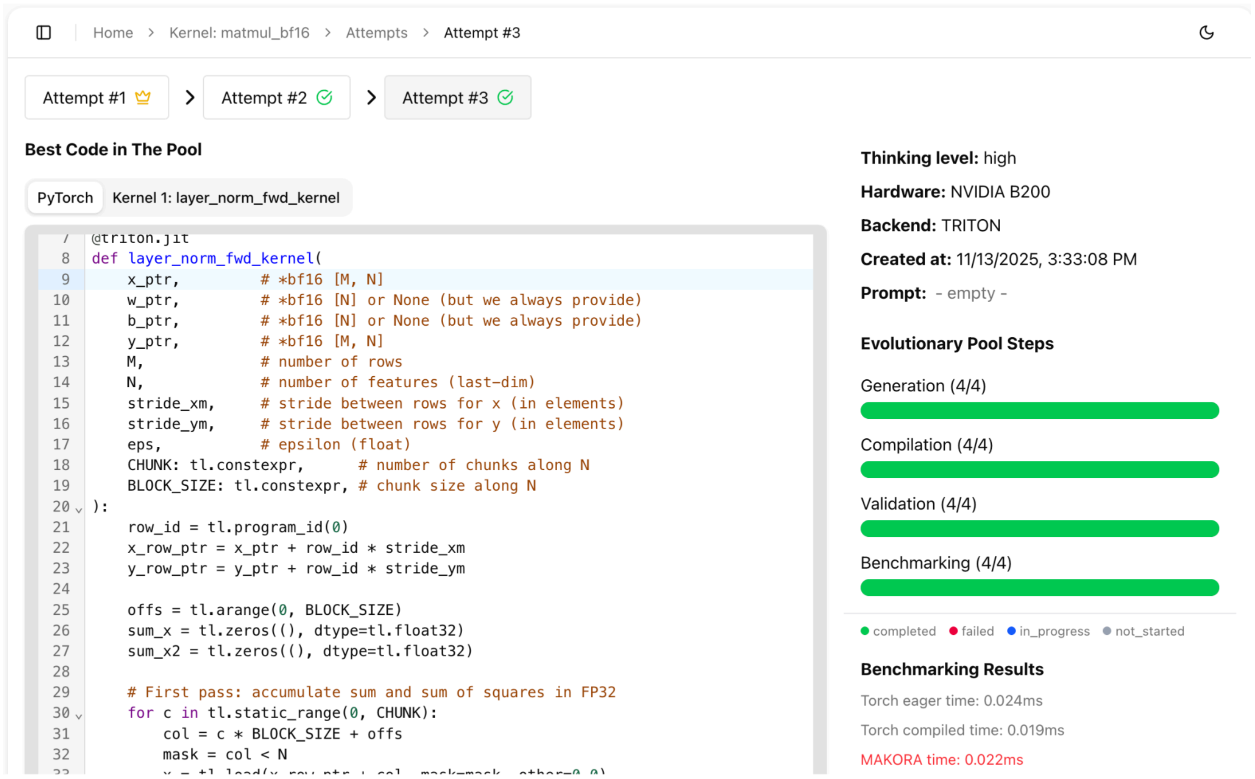
Generation (4/4)
Compilation (4/4)
Validation (4/4)
Benchmarking (4/4)

● completed ● failed ● in_progress ● not_started

Benchmarking Results

Torch eager time: 0.021ms
Torch compiled time: 0.021ms
MAKORA time: 0.016ms

Figure 3: 2nd attempt, matmul_bf16 on mako.



Home > Kernel: matmul_bf16 > Attempts > Attempt #3

Attempt #1 > Attempt #2 > Attempt #3

Best Code in The Pool

PyTorch | Kernel 1: layer_norm_fwd_kernel

```
@triton.jit
def layer_norm_fwd_kernel(
    x_ptr,          # *bf16 [M, N]
    w_ptr,          # *bf16 [N] or None (but we always provide)
    b_ptr,          # *bf16 [N] or None (but we always provide)
    y_ptr,          # *bf16 [M, N]
    M,              # number of rows
    N,              # number of features (last-dim)
    stride_xm,      # stride between rows for x (in elements)
    stride_ym,      # stride between rows for y (in elements)
    eps,            # epsilon (float)
    CHUNK: tl.constexpr,      # number of chunks along N
    BLOCK_SIZE: tl.constexpr, # chunk size along N
):
    row_id = tl.program_id(0)
    x_row_ptr = x_ptr + row_id * stride_xm
    y_row_ptr = y_ptr + row_id * stride_ym

    offs = tl.arange(0, BLOCK_SIZE)
    sum_x = tl.zeros((), dtype=tl.float32)
    sum_x2 = tl.zeros((), dtype=tl.float32)

    # First pass: accumulate sum and sum of squares in FP32
    for c in tl.static_range(0, CHUNK):
        col = c * BLOCK_SIZE + offs
        mask = col < N
```

Thinking level: high
Hardware: NVIDIA B200
Backend: TRITON
Created at: 11/13/2025, 3:33:08 PM
Prompt: - empty -

Evolutionary Pool Steps

Generation (4/4)
Compilation (4/4)
Validation (4/4)
Benchmarking (4/4)

● completed ● failed ● in_progress ● not_started

Benchmarking Results

Torch eager time: 0.024ms
Torch compiled time: 0.019ms
MAKORA time: 0.022ms

Figure 4: 3rd attempt, matmul_bf16 on mako.

Makora Generate Experience

Using Mako.dev was very easy. Since we already had a working prompt from KernelLLM, we simply reused it — the platform automatically checked the prompt, generated a Triton kernel, compiled it, validated correctness, and benchmarked it against PyTorch eager and compiled execution. This made the workflow smooth and fast without needing to write or debug the kernel ourselves.

The only downside we found is that we couldn't run the generated kernels directly on the MI300X system. However, the generated kernel did run successfully on other GPUs ensuring portability.

Helion

Helion is a Python DSL that lets you write kernels in a tile-oriented way ("PyTorch with tiles"), and compiles the code inside the outer *hl.tile(...)* loop into a single Triton kernel. The key idea is: you express what should be tiled, while Helion's autotuner decides how (tile sizes, iteration order, memory-layout tricks, etc.) for a given GPU and shape.

In our matmul Helion kernel [x] , the outer hl.tile([m, n]) defines the output tile grid, and the inner hl.tile(k) becomes the K-reduction loop using torch.addmm. We accumulate in FP32 and cast back to the promoted output dtype, matching the standard matmul stability pattern.