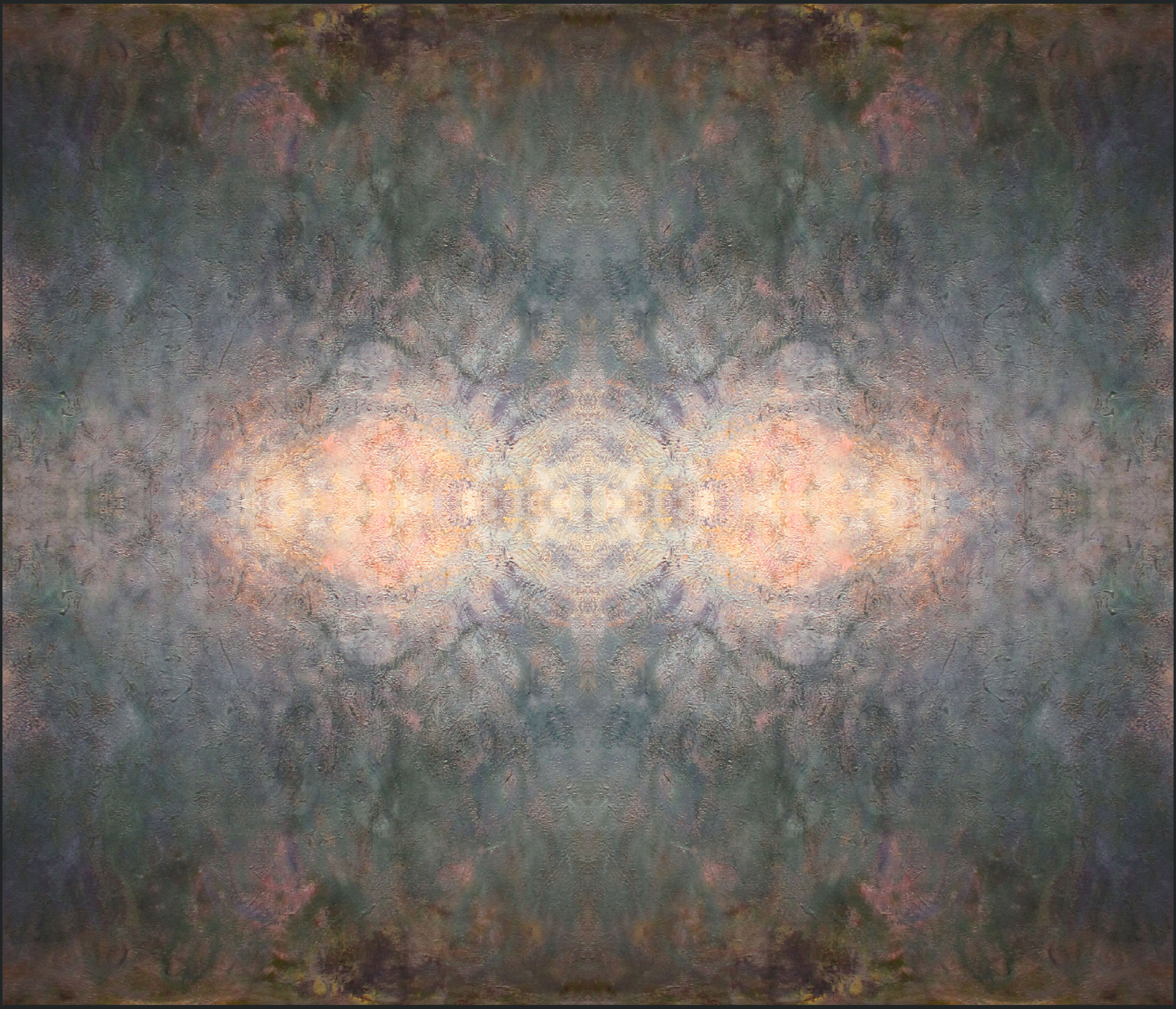
ALEPH ALPHA

ALPHA-MoE

A megakernel for faster tensor parallel inference



Authors: Szymon Ożóg, Eric Schreiber, Lukas Blübaum @ Aleph Alpha GmbH, December 2025

Alpha-MoE
A megakernel for faster tensor parallel inference

Szymon Ożóg, Eric Schreiber, Lukas Blübaum © Aleph Alph GmbH

December 2025

Most new models today use a Mixture of Experts (MoE) architecture. While MoE offers clear advantages over dense architectures, its sparse communication pattern introduces unique challenges for performance optimization.

We're excited to announce the release of **Alpha-MoE**, a kernel library built around a fused MoE megakernel for FP8 W8A8 (8-bit weights, 8-bit activations). Alpha-MoE delivers up to **200% speed improvements** compared to current Triton kernels in open-source LLM serving frameworks such as vLLM and SGLang.

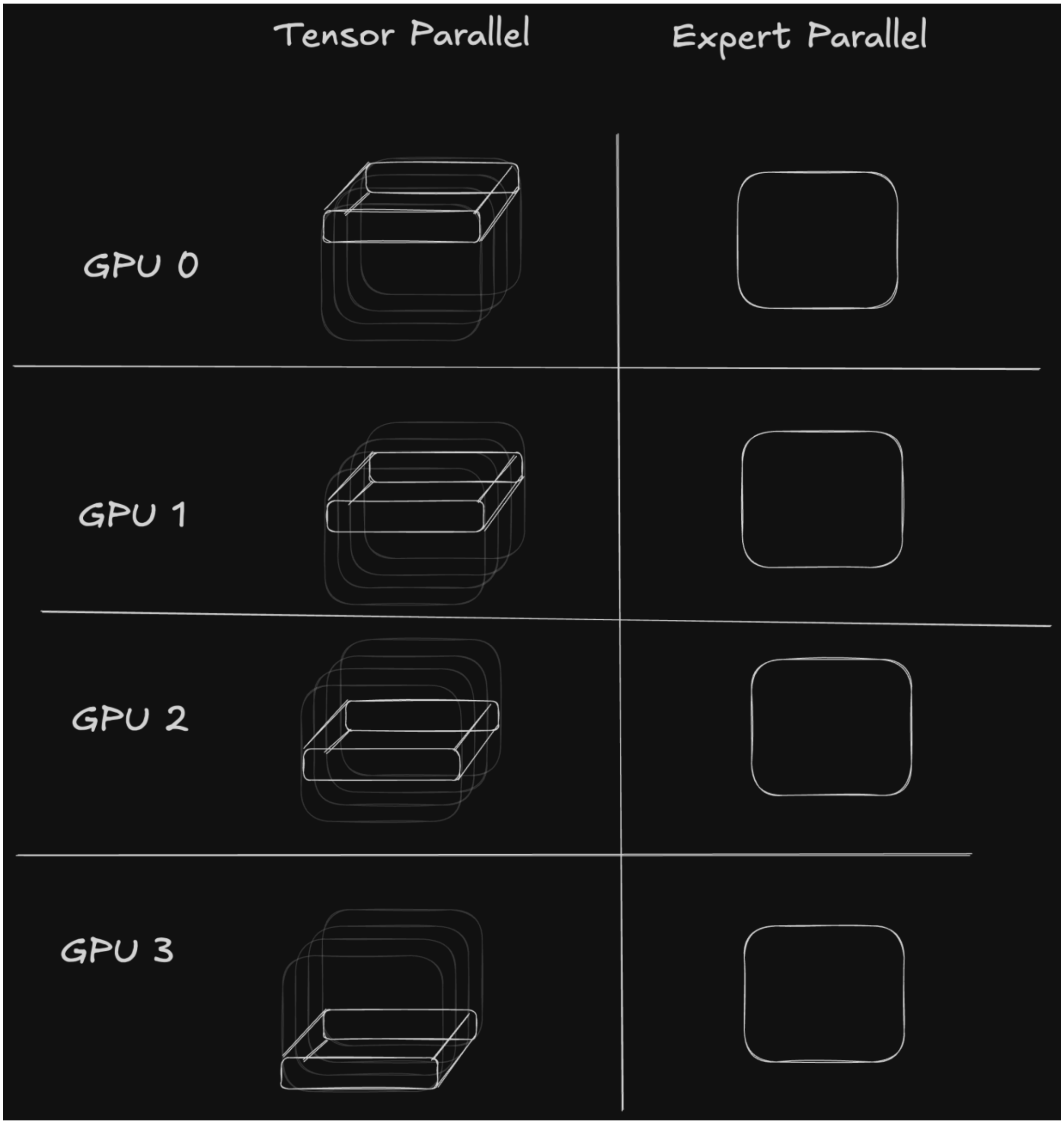
At its core, Alpha-MoE introduces a fused W8A8 MoE megakernel for Hopper that combines **Up-Proj + Gate GEMM → SwiGLU → activation quantization → Down-Proj** into a single persistent kernel.

Motivation: Closing the Gap Between Expert Parallelism and Tensor Parallelism

When serving MoE models, providers typically choose between **Expert Parallelism (EP)** and **Tensor Parallelism (TP)**. Both approaches have clear benefits and trade-offs.

In EP, the MoE matrices are sharded across the expert dimension. This provides very good GPU utilization but requires a very large batch size to maintain good expert utilization. While effective for large-scale deployments or throughput-oriented scenarios, EP can be challenging for latency-sensitive or small to medium-scale deployments.

In TP, the up-projection matrix is sharded across the row dimension, and the down-projection matrix is sharded across the column dimension. This ensures uniform GPU utilization but introduces issues with low dimensionality. At large TP sizes, matrix shards become very small, shifting operations from compute-bound to memory-bound as TP size increases.



Kernel Fusion and Design

Due to low dimensionality in high TP scenarios, the kernel is highly prone to fusion to reduce global memory access. This work proposes a fused megakernel that handles most operations performed by a MoE layer.

Kernel Operations

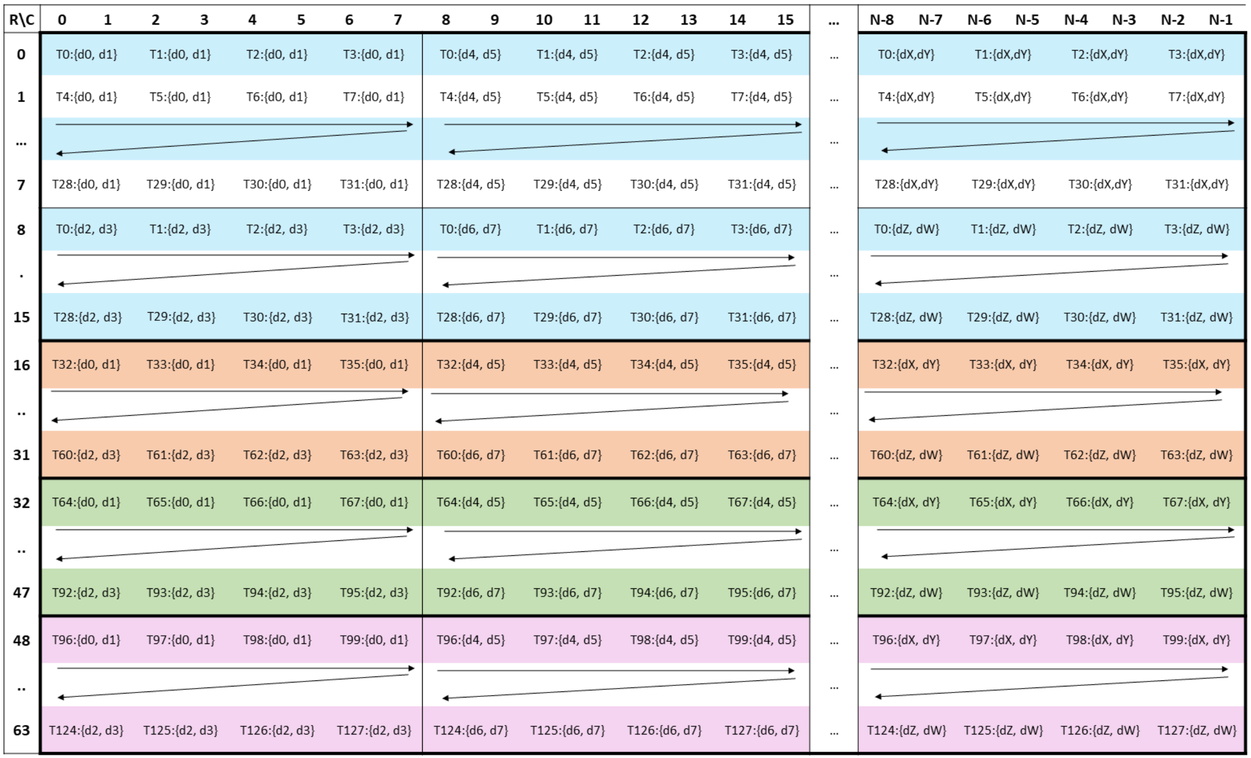
A MoE layer for W8A8 quantization involves several steps:

1. Routing tokens to the correct experts
2. Performing the Up Projection + Gate GEMM
3. Applying the SwiGLU activation function
4. Quantizing activations for the next GEMM
5. Performing the Down Projection GEMM
6. Combining results locally on the device
7. Performing an AllReduce for a global combine across GPUs

Our approach focuses on **combining steps 2 through 6 into one fused megakernel**. The kernel is optimized for the Hopper architecture and, in addition to traditional Hopper GEMM optimizations — such as Producer/Consumer pipelines, multi-stage loading, WGMMA, and async stores — it introduces two additional techniques to enable efficient fusion of all kernels.

SwiGLU Weight Interleaving

The first challenge in fusing the SwiGLU activation function without adding memory overhead is ensuring that the results of both the Up Projection and Gate GEMM land on the same thread. To achieve this, we need to examine the output registers of WGMMA operations in CUDA:

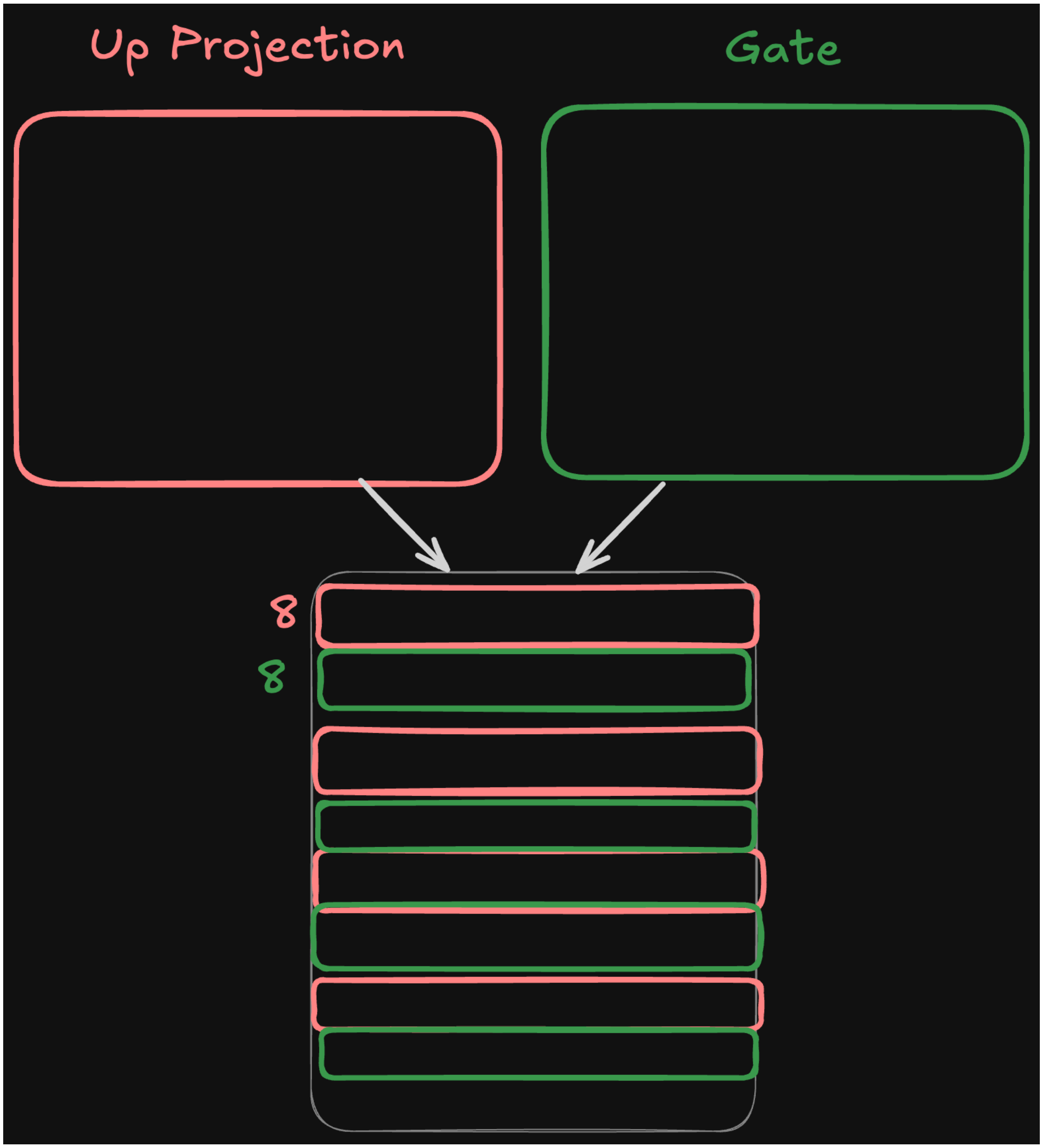


R\C	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…	N-8	N-7	N-6	N-5	N-4	N-3	N-2	N-1
0	T0:{d0, d1}		T1:{d0, d1}		T2:{d0, d1}		T3:{d0, d1}		T0:{d4, d5}		T1:{d4, d5}		T2:{d4, d5}		T3:{d4, d5}		…	T0:{dX,dY}		T1:{dX,dY}		T2:{dX,dY}		T3:{dX,dY}	
1	T4:{d0, d1}		T5:{d0, d1}		T6:{d0, d1}		T7:{d0, d1}		T4:{d4, d5}		T5:{d4, d5}		T6:{d4, d5}		T7:{d4, d5}		…	T4:{dX,dY}		T5:{dX,dY}		T6:{dX,dY}		T7:{dX,dY}	
…																	…								
7	T28:{d0, d1}		T29:{d0, d1}		T30:{d0, d1}		T31:{d0, d1}		T28:{d4, d5}		T29:{d4, d5}		T30:{d4, d5}		T31:{d4, d5}		…	T28:{dX,dY}		T29:{dX,dY}		T30:{dX,dY}		T31:{dX,dY}	
8	T0:{d2, d3}		T1:{d2, d3}		T2:{d2, d3}		T3:{d2, d3}		T0:{d6, d7}		T1:{d6, d7}		T2:{d6, d7}		T3:{d6, d7}		…	T0:{dZ, dW}		T1:{dZ, dW}		T2:{dZ, dW}		T3:{dZ, dW}	
.																	…								
15	T28:{d2, d3}		T29:{d2, d3}		T30:{d2, d3}		T31:{d2, d3}		T28:{d6, d7}		T29:{d6, d7}		T30:{d6, d7}		T31:{d6, d7}		…	T28:{dZ, dW}		T29:{dZ, dW}		T30:{dZ, dW}		T31:{dZ, dW}	
16	T32:{d0, d1}		T33:{d0, d1}		T34:{d0, d1}		T35:{d0, d1}		T32:{d4, d5}		T33:{d4, d5}		T34:{d4, d5}		T35:{d4, d5}		…	T32:{dX, dY}		T33:{dX, dY}		T34:{dX, dY}		T35:{dX, dY}	
..																	…								
31	T60:{d2, d3}		T61:{d2, d3}		T62:{d2, d3}		T63:{d2, d3}		T60:{d6, d7}		T61:{d6, d7}		T62:{d6, d7}		T63:{d6, d7}		…	T60:{dZ, dW}		T61:{dZ, dW}		T62:{dZ, dW}		T63:{dZ, dW}	
32	T64:{d0, d1}		T65:{d0, d1}		T66:{d0, d1}		T67:{d0, d1}		T64:{d4, d5}		T65:{d4, d5}		T66:{d4, d5}		T67:{d4, d5}		…	T64:{dX, dY}		T65:{dX, dY}		T66:{dX, dY}		T67:{dX, dY}	
..																	…								
47	T92:{d2, d3}		T93:{d2, d3}		T94:{d2, d3}		T95:{d2, d3}		T92:{d6, d7}		T93:{d6, d7}		T94:{d6, d7}		T95:{d6, d7}		…	T92:{dZ, dW}		T93:{dZ, dW}		T94:{dZ, dW}		T95:{dZ, dW}	
48	T96:{d0, d1}		T97:{d0, d1}		T98:{d0, d1}		T99:{d0, d1}		T96:{d4, d5}		T97:{d4, d5}		T98:{d4, d5}		T99:{d4, d5}		…	T96:{dX, dY}		T97:{dX, dY}		T98:{dX, dY}		T99:{dX, dY}	
..																	…								
63	T124:{d2, d3}		T125:{d2, d3}		T126:{d2, d3}		T127:{d2, d3}		T124:{d6, d7}		T125:{d6, d7}		T126:{d6, d7}		T127:{d6, d7}		…	T124:{dZ, dW}		T125:{dZ, dW}		T126:{dZ, dW}		T127:{dZ, dW}	

(tid % 128) : fragments

Image reference: 1. Introduction — PTX ISA 9.0 documentation

We observe that each thread contains results from two tiles separated by eight rows. By interleaving the Up Projection and Gate weight matrices in chunks of eight, we ensure that after the first GEMM stage, each thread holds the outputs of both the Up Projection and the Gate. This arrangement allows the activation function to be applied without additional memory overhead.

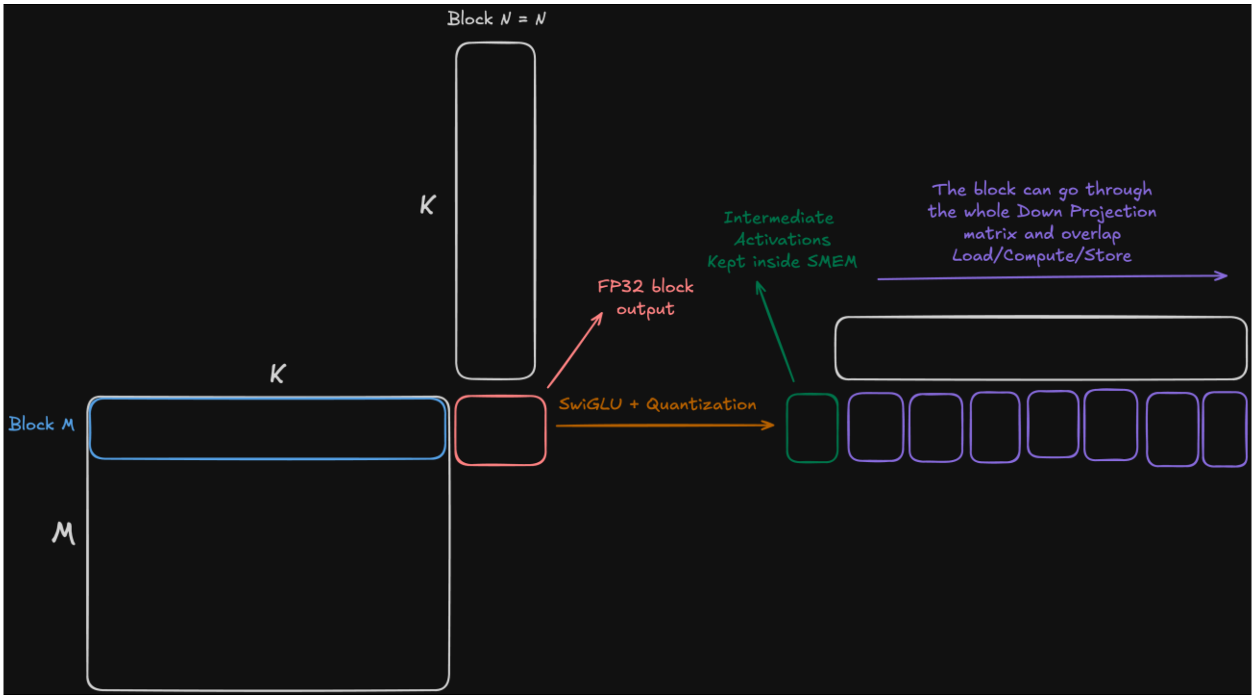


Persistent Down Projection

High TP sizes create challenges for the down projection because heavy column sharding makes the K dimension very small. This severely impacts memory bandwidth since compute and memory overlap is poor, and we cannot achieve enough bytes in flight to saturate available bandwidth.

However, with aggressive sharding (we've observed speed improvements with a moe_intermediate_size shard of 256), we can increase the WGMMA block size enough to store the full result of the Up Projection + Gate GEMM in shared memory. This enables the Down Projection GEMM to run inside a single block, delivering two key benefits:

- Activations remain persistently stored in shared memory for the entire Down Projection, eliminating the need to reload them from global memory.
- We overlap computation of the previous row's multiplication with fetching the next row from global memory, improving efficiency.


Block N = N
K
K
Block M
M
FP32 block output
SwiGLU + Quantization
Intermediate Activations Kept inside SMEM
The block can go through the whole Down Projection matrix and overlap Load/Compute/Store

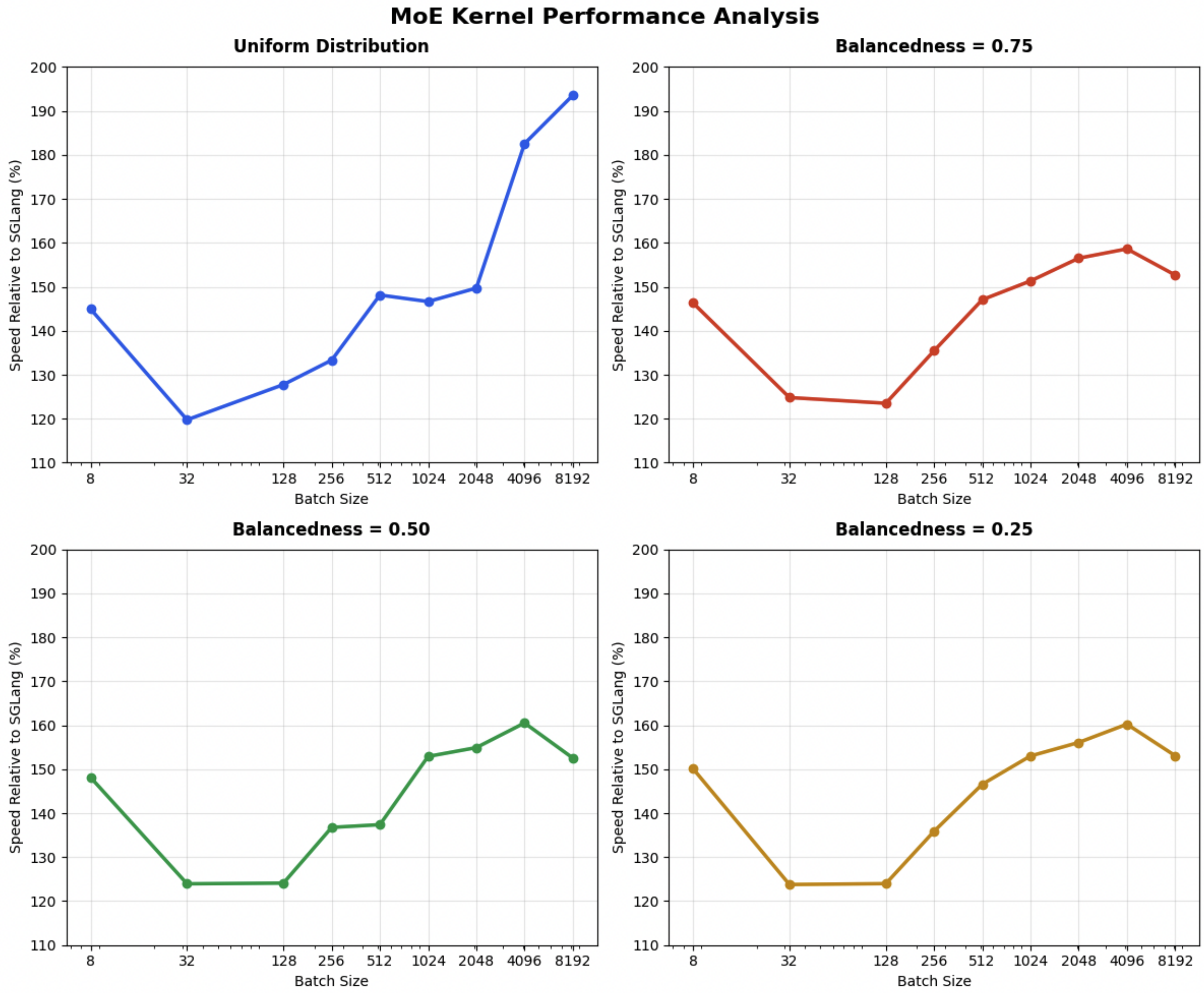
These kernel-level improvements translate directly into faster real-world throughput and lower latency when serving large MoE models in a TP setting.

Results

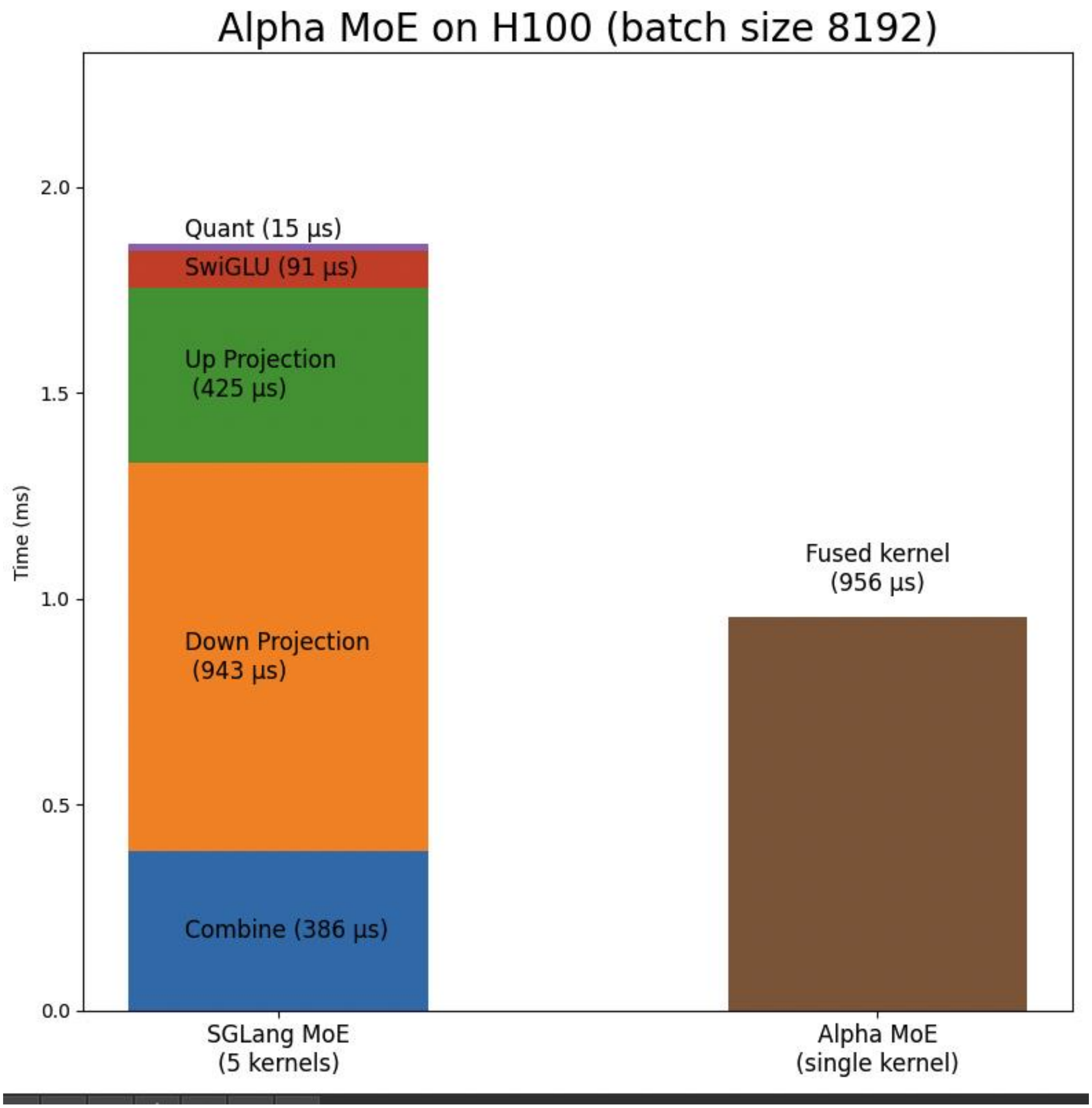
To evaluate performance, we benchmarked Alpha-MoE against state-of-the-art MoE layer implementations used in **vLLM** and **SGLang** for TP deployments, as well as end-to-end serving performance on **Qwen3-Next-80B** and **DeepSeek-R1**.

Standalone Kernel

For the standalone kernel configured for DeepSeek running in **TP16** (256 experts + 1 shared, moe_intermediate_size = 128 per shard, and a hidden size of 7168), Alpha-MoE consistently outperforms the current solution across all batch sizes and expert balance configurations.

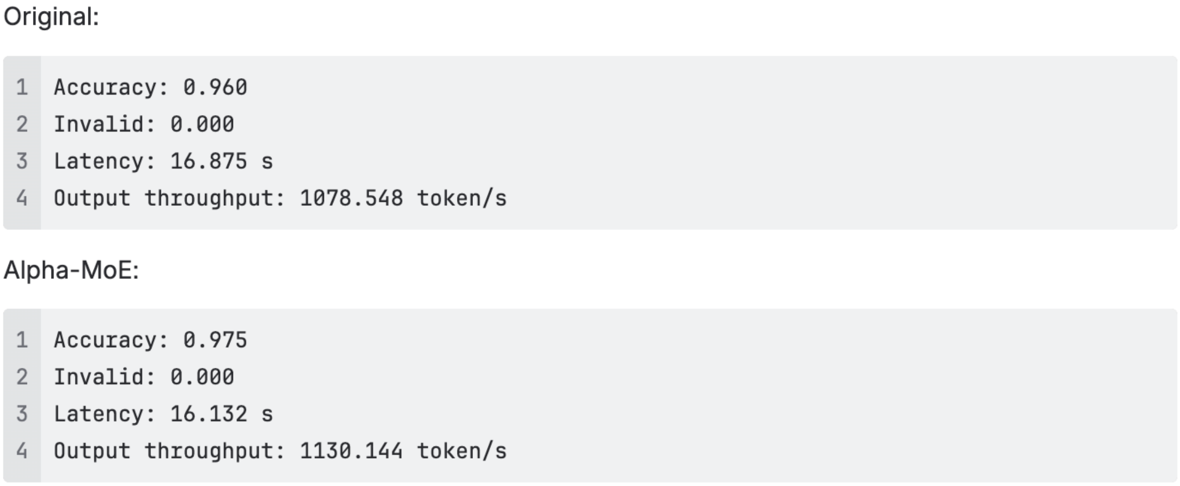


We analyzed each component of the MoE layer in SGLang and found that the largest performance gains come from accelerating the down projection matrix using the previously described **Persistent Down Projection** technique.


Alpha MoE on H100 (batch size 8192)
Time (ms)
2.0
1.5
1.0
0.5
0.0
Quant (15 µs)
SwiGLU (91 µs)
Up Projection
(425 µs)
Down Projection
(943 µs)
Combine (386 µs)
Fused kernel
(956 µs)
SGLang MoE
(5 kernels)
Alpha MoE
(single kernel)

Model Comparisons

Before running model-level benchmarks, we first validated accuracy using **DeepSeek-R1** on **gsm8k** to ensure there was no degradation in model quality.



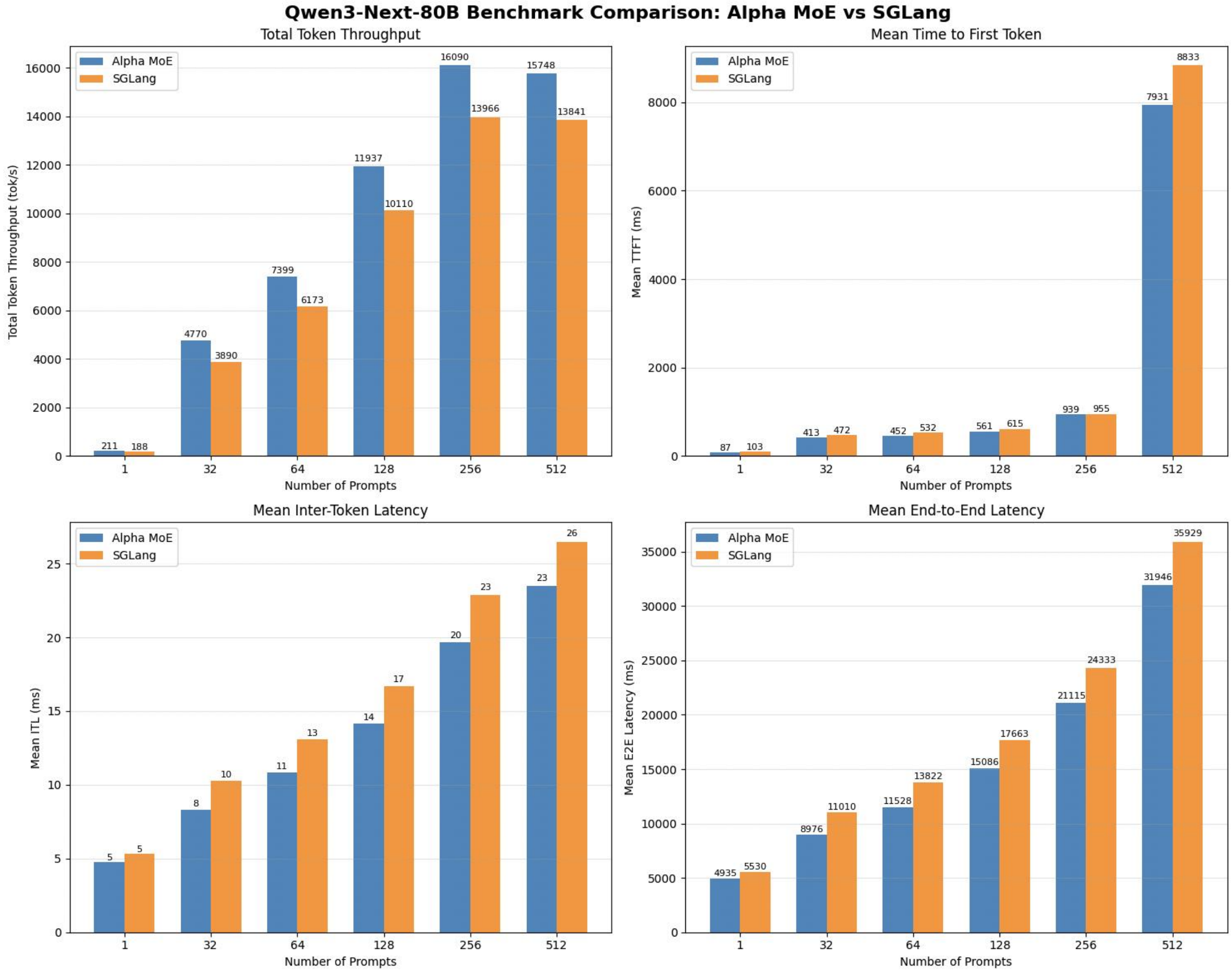
Original:

```
Accuracy: 0.960
Invalid: 0.000
Latency: 16.875 s
Output throughput: 1078.548 token/s
```

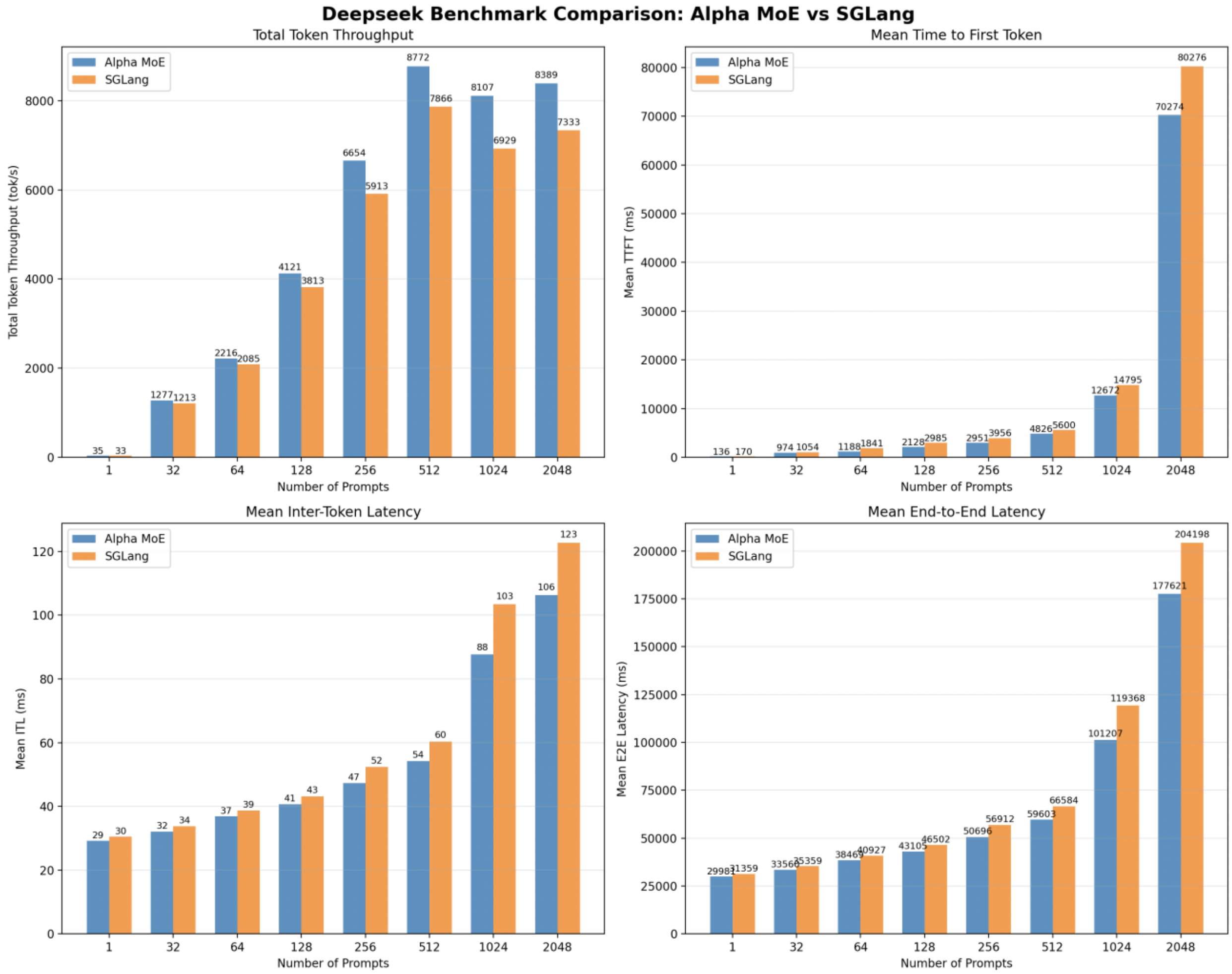
Alpha-MoE:

```
Accuracy: 0.975
Invalid: 0.000
Latency: 16.132 s
Output throughput: 1130.144 token/s
```

Qwen3-Next-80B



DeepSeek-R1



```
Average Improvements:
  Throughput: +10.0%
  TTFT: +19.7%
  ITL: +8.6%
  E2E: +9.1%
```

Usage

We're happy to release the kernel alongside scripts for finding optimal configurations as a standalone Python package available on our GitHub - Aleph-Alpha/Alpha-MoE.