Samvrit Srinath, Srivishnu Ramamurthi, Zhiyuan Guo, Zijian He

Channel: Megakernels

Megakernels Blog Post

Intro [Srivishnu Ramamurthi, Samvrit (helped)]

In the world of Large Language Models (LLMs), performance is often dictated by how efficiently we can string together complex operations—such as chaining Attention layers into MLPs. One powerful technique to optimize this pipeline is the **GPU Megakernel**: the fusion of multiple distinct GPU kernels into a single execution unit.

By fusing kernels, we unlock performance gains from several angles:

- **Latency Reduction:** It eliminates the GPU overhead of kernel launch (startup) and teardown (spin down) for individual operations.
- **Improved Pipelining:** It allows for tighter memory / computation pipelining between logical steps.
- **Memory Efficiency:** It enables potential memory sharing between kernels, reducing expensive global memory round-trips.

However, building a Megakernel is notoriously difficult. It requires engineers to manually fuse existing kernels at a fine-grained level; usually, the more performance you want, the higher the engineering cost. This complexity has driven the demand for compilers and frameworks—such as **Mirage** and **Thunderkittens**—that aim to automate or simplify the creation of Megakernels.

While these tools are promising, they are not without flaws. In this post, we are going to focus specifically on **Thunderkittens**. We will deep dive into its current architectural drawbacks and introduce a novel approach designed to alleviate them. We then validate this method through basic profiling, overhead analysis, and execution-time comparisons. Finally, we use these results to guide the future direction of this research and outline the next steps toward an external blog post, as the implementation is still in progress.

Deep Dive into ThunderKittens

Writing efficient GPU kernels is notoriously difficult because of the legacy of the hardware itself. GPUs were originally architected for general-purpose tasks—rendering graphics and running scientific simulations. Deep Learning is a relatively "novel" use case in the lifespan of GPU architecture, yet standard CUDA still caters to that broad, general-purpose history.

ThunderKittens bridges this gap by strictly specializing for Machine Learning workloads. It acts as an embedded framework that manages the complexities of the GPU memory hierarchy (specifically tiling) so the developer doesn't have to.

Think of it as an abstraction layer similar to modern web frameworks. Just as React simplifies raw JavaScript to streamline web development, ThunderKittens wraps the complexity of raw CUDA. It allows engineers to build faster, simpler kernels without sacrificing the fine-grained control needed for high performance.

A standard ThunderKittens kernel follows the **LCSF** (Load, Compute, Store, Finish) template, implemented using a **Producer-Consumer** paradigm. In this model, the GPU warps are divided into two specialized groups:

- **Producer Warps (Load/Store):** Responsible for memory I/O. They handle the "Load" step (moving data from HBM to shared memory) and the "Store" step (writing results back to HBM).
- **Consumer Warps (Compute/Finish):** Responsible for the heavy lifting. They handle the "Compute" step on tiles in fast memory and the "Finish" synchronization steps.

The power of this approach lies in **latency hiding** via pipelined buffers. By specializing SM warps, the Producers can issue asynchronous Tensor Memory Accelerator (TMA) instructions to fetch the next batch of data *while* the Consumers are simultaneously processing the current batch. This effective overlap of memory access and computation allows the kernel to mask the latency of slow global memory operations, keeping the compute units fully saturated.

ThunderKittens provides two distinct templates for implementing Megakernels: the **Interpreter Template** and the **VM Template**.

1. The Interpreter Template

The **Interpreter Template** acts as the glue that ties multiple kernels together under a single persistent umbrella.

While standard ThunderKittens kernels use the **LCSF** (Load-Compute-Store-Finish) template to handle **intra-op pipelining** (overlapping memory and compute *within* a single operation), the Interpreter template enables **inter-op pipelining**—overlapping the execution of entirely *different* kernels.

In this model, developers manually fuse operations (e.g., combining partial attention with reduction stages) by orchestrating synchronization explicitly:

- **Data Exchange:** Intermediate results are written to global scratch buffers.
- **Signaling:** Task completion is indicated via semaphores.
- **Synchronization:** Dependencies are managed by polling those semaphores.

A prime example of this architecture is **ThunderMLA** (see the Hazy Research blog: ThunderMLA). The team used the Interpreter template to separate the partial and reduction stages of the attention mechanism, allowing them to time these stages together within a single grid.

The Interpreter is lightweight but has a hard constraint: inter-op pipelining is restricted to a depth of 2. This means you cannot begin loading memory for a third kernel while the first is still computing. For deeper pipelines and more complex megakernels, we need a more robust abstraction.

2. The VM Template

The **VM Template** transforms the GPU into a virtual machine capable of complex, dynamic execution. It introduces significant architectural changes to support deep pipelining.

The VM template replaces static buffers with a **page-based memory system**. This allows for dynamic memory management similar to an OS:

- **Allocation:** Memory pages are allocated to specific operations on the fly.
- **Reuse:** Physical memory pages are reused across different operations to minimize footprint.
- **Synchronization:** Synchronization primitives are tied directly to memory pages, simplifying dependency tracking.

The VM template also modifies the standard LCSF model. It removes the "Finish" role (now handled by the Compute step) and introduces two control roles:

1. **Controller (New):** The "brain" of the operation. It manages the instruction stream, allocates memory pages, and creates semaphores.
2. **Launcher (New):** Coordinates timing and dependencies, ensuring all prerequisites are met before computation starts.

The VM template introduces overhead due to the complexity of the controller and page management logic, this abstraction enables **deep inter-op pipelines** that exceed the Interpreter's depth limit, allowing for true multi-operation Megakernels that overlap many stages of execution simultaneously. The ThunderKittens team demonstrated the power of this approach by implementing Llama 1 and Llama 8B using the VM template (see Hazy Research blog: Look Ma, No Bubbles).

Both of these megakernel templates rely on a shared control plane: **The Python Coordinator**.

To manage these persistent kernels, an external Python script acts as the orchestrator. This script coordinates execution across the GPU's Streaming Multiprocessors (SMs) by building a **static schedule of operations** for each SM to follow. Whether utilizing the simpler Interpreter or the complex VM, this Python layer ensures that the persistent grid executes the correct kernel logic in the correct order.

The Downsides of ThunderKittens

Several architectural constraints limit ThunderKittens absolute efficiency in complex scenarios. These constraints can be classified into two areas.

1. Memory Related

- **Fixed Pipeline Depth:** Users must predefine pipeline depth (e.g., depth of 2).
    - **Bandwidth Inefficiency:** If a primary kernel has a long computation time, a third kernel cannot begin loading, which wastes available memory bandwidth.
    - **Resource Trade-offs:** Increasing depth is not a universal fix; it introduces overhead for synchronization primitives and consumes valuable scratch space—more than **10kB per depth level** in production level megakernels (link)—reducing the total shared memory available.
- **Lack of Workload Dynamism:** TK's static dependency mapping makes it inherently incompatible with dynamic kernels. Complex control flows, such as those required by **Mixture of Experts (MoE)** where the operation sequence changes based on runtime outputs, are currently unsupported.
- **Manual Synchronization Overheads:** Synchronization is a manual, low-level process. Unlike ideal "one-button" megakernel compilers, TK requires careful manual implementation of primitives to ensure memory dependencies are handled without sacrificing performance.

2. Scheduler Related:

- **Static Memory Dependencies:** Memory dependencies in TK are statically resolved by the compiler. This creates suboptimal execution when handling arbitrary kernel sequences. For instance, in dual matrix multiplication operations, a block calculated by SM 1 must currently be stored to global memory (HBM) to be re-accessed by a subsequent operation. A framework that does not co-optimize the scheduler with the kernel misses opportunities to prevent these costly global memory round-trips.
- **The Python Scheduler Bottleneck:** TK relies on an external Python script to coordinate execution across SMs. While this overhead is manageable for static inputs (like decoding-only kernels) where schedules are cached, it becomes a critical bottleneck for dynamic workloads like the prefill phase. Because token counts vary significantly per request, the schedule must be regenerated on the fly; this generation cost introduces significant latency.

Currently, our solution focuses on solving the **memory related downsides** of ThunderKittens - we hope to then extend our solution to co-optimize with the scheduler to solve the **scheduler related downsides** at a later date.

Our Proposed Solution: Dynamic Runtime Memory Management

To solve memory related issues, we propose transitioning from static, compile-time management to a **runtime-managed shared memory architecture**.

1. Decoupled Control Flows

At compile time, we decompose each megakernel task into two loosely coupled instruction flows:

- **Memory Flow**: Handles data movement, allocation, and prefetching
- **Computation Flow**: Focuses purely on arithmetic execution

These flows communicate asynchronously via **signal FIFOs** to exchange dependency state, enabling several key optimizations:

- **Dynamic Memory Hierarchy Selection**: The memory flow can switch between intra-warp shared memory and inter-warp global memory at runtime without modifying compute kernels
- **Adaptive Synchronization**: Synchronization methods (e.g., barriers vs. fine-grained semaphores) can be selected based on workload characteristics without recompilation
- **Independent Optimization**: Compute and memory subsystems can be tuned separately - for example, changing a prefetch strategy doesn't require re-validating arithmetic correctness

This decoupling mirrors CPU out-of-order execution principles: as long as dependencies are satisfied via signals, the "how" of data delivery remains transparent to computation.

2. Runtime Dependency Resolution

Our framework manages shared memory at execution time:

- **Pre-execution Allocation:** Memory instructions must allocate required dependencies before they execute.
- **Feedback Loops:** Once data is loaded, a message is sent to the **Memory-to-Compute (M2C)** queue. After compute finishes, it alerts the **Compute-to-Memory (C2M)** queue to free the space.
- **Dynamic Blocking:** Both the memory and compute sides can optionally block on specific messages, allowing the system to wait for dependencies only when strictly necessary.

3. On-GPU Dynamic Memory Allocator

A dedicated allocator runs directly on the memory side to manage shared memory pools dynamically. This enables runtime re-scheduling and better resource utilization for dynamic workloads (such as MoE) as well as shared memory sharing across operations as memory can be freed and reused based on actual execution paths rather than static compile-time assumptions.

Profiling and Instrumentation [Samvrit Srinath]

Llama 1B

We situate this Megakernel analysis with a timing experiment with TinyLlama 1B, identifying where these bottlenecks under the Thunderkittens framework are prevalent.

A single Llama Transformer block is decomposed into a sequence of **10 Micro-Operations (Opcodes)** sequentially by the Virtual Machine.

The Data Flow Dependencies

The execution follows a strict dependency graph. Crucially, intermediate results (activations) are often written to Global Memory (HBM) between these steps, creating the "Inter-Op" boundaries we profiled.

1. **Attention Norm (RMS):** Reads Input → Writes Normed Input.
2. **QKV + RoPE:** Reads Normed Input → Computes Query/Key/Value → Applies Rotary Embedding → Writes to KV Cache & HBM.
3. **Attention Decode:** Reads Q/K/V → Computes Attention Scores → Writes Partial Results (if split across SMs).
4. **Attention Reduction:** Reads Attention Partials → Aggregates → Writes Final Attention Output.
5. **Output Projection (Residual):** Reads Attention Output + Original Input → Computes Projection → Adds Residual → Writes Result.
6. **MLP Norm (RMS):** Reads Result → Writes Normed Input.
7. **Gate + Up Proj:** Reads Normed Input → Computes Gate/Up → Applies SiLU.
8. **Down Proj (Residual):** Reads MLP Output + Residual Input → Writes Final Layer Output.

ThunderKittens Implementation Concerns

The system is bifurcated into a **Host Driver** (Python) that compiles the schedule and a **Device Runtime** (C++) that executes it.

Host Driver

Located in demos/kvm-runner, this library acts as the "Operating System" for the GPU.

- **LlamaKVM Class:** Responsible for mapping PyTorch tensors to the raw GPU pointers required by the kernel. It allocates the critical timings tensor used for our profiling.
- **Instruction Generation:** It statically compiles the sequence of 10 Opcodes into a binary command stream. This allows the GPU to execute the entire 32-layer model without CPU intervention.

Device Runtime

Located in tests/vm/llama_official, the kernel implements each component of the Llama Inference Procedure (listed below). Each Opcode triggers a specific C++ template specialized for that mathematical operation.

Kernel 1: RMS QKV + RoPE

- **Pipeline:** Uses rms_matvec_pipeline. This is a specialized pipeline that first loads the RMS weight vector, then streams the matrix tiles.
- **Consumer Loop:** The warp waits for the input vector to be normed, then iterates through weight tiles to perform the $X \times W_{qkv}$ multiplication. Finally, it applies the rotary embedding math in registers before writing to the KV cache.

Kernel 2: Partial Attention

- **Pipeline:** Uses a custom attention_pipeline to stream KV cache blocks.
- **Consumer Loop:** This is the most complex loop. It loads the Query vector once, then iterates over the KV cache pages. It computes Score = Q @ K.T, applies Softmax, and accumulates Output += Score @ V.
- **Constraint:** It must wait for the QKV kernel (Op 1) to finish writing the Query vector to HBM before it can load it.

Kernel 3: Attention Reduction

- **Pipeline:** Simple load-store loop.
- **Consumer Loop:** It does not start until a global barrier *L_partial_all_arrived* is lifted. It reads partial outputs from multiple SMs, sums them, and writes the final result.

Kernels 4 & 6: Projections

- **Pipeline:** matvec_pipeline (Standard 3-stage pipe).
- **Consumer Loop:** Performs standard Matrix-Vector multiplication ($A \times W$) but adds a "Residual Load" step: it fetches the original input vector from HBM to add to the result ($Y = Ax + x_{resid}$).

Profiling Experiments

To quantify the overhead of this architecture, we designed an experiment to isolate **Useful Work** (Active Compute) from **Structural Latency** (Stalls).

Standard profiling conflates "Kernel Running Time" with "Compute Time." To resolve this, we instrumented the C++ source to record a *real compute start* (first usage of the relevant information).

The Logic:

The ThunderKittens VM uses barriers (semaphores) to synchronize data movement. The "Consumer Warp" (the arithmetic unit) sits in a loop:

1. wait(data_ready) ← **STALL STATE**
2. compute() ← **ACTIVE STATE**

We injected a timestamp record *immediately* after the wait returns. This timestamp marks the exact nanosecond when data arrived and useful math began.

```
// Inside matvec_pipeline.cuh :: consumer_loop
void consumer_loop(...) {
    // 1. Wait for Global Memory (HBM) weights to arrive in Shared Mem
    wait(gmem_wait_token);

    // <--- INSTRUMENTATION POINT --->
    if (lane_id == 0) s.record(90); // TEVENT_REAL_COMPUTE_START
    // <--- END INSTRUMENTATION --->

    // 2. Begin Math (WGMMA)
    compute_gemm(...);
}
```

Results

Hardware Context

All experiments were conducted on a **GCP A3 instance with NVIDIA H100 SXM5 GPUs**. GCP's A3 machine types feature H100 SXM GPUs (nvidia-h100-80gb), which provide the following memory bandwidth specifications:

H100 SXM5 (GCP A3 configuration):

- **HBM3 Bandwidth**: 3 TB/s (over 3 TB/sec)
- **L2 Cache**: 50 MB capacity
- **L2 Cache Bandwidth**: ~5-6 TB/s
- **Shared Memory per SM**: Up to 228 KB maximum per SM
- **GPU-to-GPU Bandwidth**: Up to 900 GB/s via NVLink 4.0

To quantify the performance characteristics of the Llama 1B pipeline, we decomposed the execution lifecycle of each kernel into distinct temporal phases. The variables presented in **Table 1** are defined as follows:

- Total Time (T_total):
  The end-to-end wall-clock duration of a single instruction execution, measured from the moment the Controller Warp issues the instruction T_startto the moment the Storer Warp signals completion (T_end). This encompasses all memory fetching, synchronization, computation, and write-back phases.
- **Delay til Real Compute** (T_delay):
  This metric isolates the "Initial Stall" or "Warmup Latency." It is calculated as the time elapsed between the instruction start and our instrumented time stamp.
  - *Significance:* This represents the duration during which the Tensor Cores (ALU) are strictly idle, blocked by synchronization barriers (e.g., L_partial_all_arrived) or waiting for the first tile of weights to be transferred from Global Memory to Shared Memory.
- Compute Stalls (T_stalls):
  The cumulative time spent waiting for data dependencies after the initial computation has begun. This is measured by instrumenting the Consumer Loop to record timestamps around internal wait() instructions.
  - *Significance:* A low value indicates effective **Intra-Operator Pipelining**—the Loader Warp is successfully fetching Tile N+1 while the Consumer processes Tile N. A high value would indicate bandwidth starvation or an insufficient pipeline depth.
- Work Time (T_work):
  The duration of "Useful Execution," derived as:
  T_{work} = T_{total} - T_{delay} - T_{stalls}
  This captures the time the hardware is actively engaged in Matrix Multiplication (w_gamma), Vector Math (RoPE/SiLU), or Epilogue Write-back. It represents the theoretical lower bound of execution time assuming infinite memory bandwidth and zero synchronization overhead.
- Start Delay %:
  The proportion of total execution time lost to the initial setup phase:
  $$\text{Start Delay } \% = \frac{T_{delay}}{T_{total}} \times 100$$
  - *Interpretation:* A high percentage (e.g., >60% for **O Proj**) identifies a kernel as "Latency Bound" rather than "Throughput Bound," highlighting the inefficiency of the static scheduling model for that specific operator.

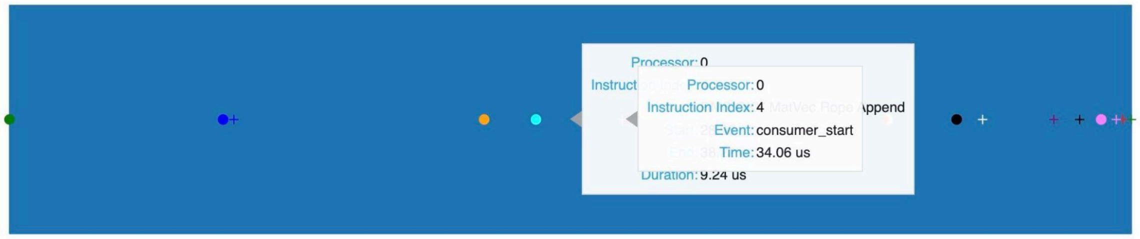
Real Delay Based on Kernel					
Kernel Type	Total Time (μS)	**Delay til Real Compute (μS)**	Compute Stalls (μS)	Work Time(μS)	Start Delay %
RMS QKV (Op 1)	8.96	2.39	0.65	5.92	26.6%
Partial Attn (Op 2)	8.82	1.20	0.66	6.97	13.6%
Attn Reduction (Op 3)	10.71	**6.50**	0.18	4.04	**60.7%**
O Proj Residual (Op 4)	11.91	**7.46**	0.06	4.39	**62.6%**
RMS SiLU (Op 5)	19.42	2.05	3.00	14.37	10.5%
Down Proj (Op 6)	17.21	2.62	1.76	12.83	15.2%
LM Head (Op 7)	97.19	3.99	3.31	89.89	4.1%

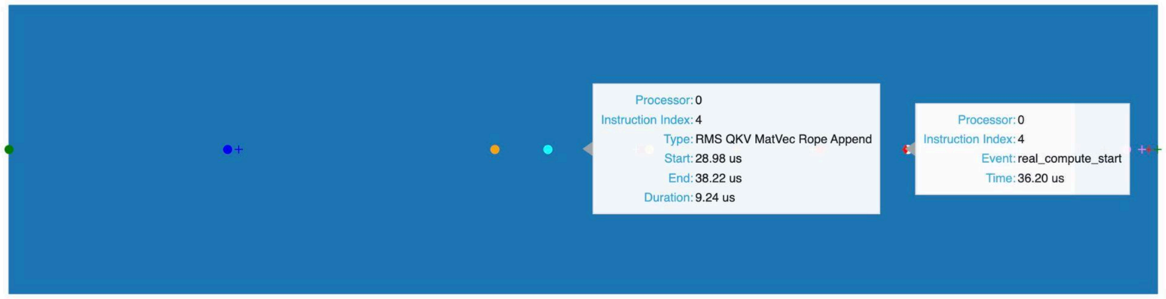
The "Delay til Real Compute" represents the **Theoretical Upper Bound** of performance recovery. In a proposed **decoupled Compute Memory** System (i.e. separating Compute and Memory Warps), these delays represent vacancies where potential work can be scheduled.

The “Round-Trip” Penalty in Op 4 (O Proj) reflects a measured delay of 7.46 microseconds, representing roughly 62.6% overhead. This happens because O Proj must wait for Attn Reduction to finish writing its output to HBM and then must read that same data back. The delay is therefore the physical cost of a full HBM round-trip. The proposed fix is to use Data Persistence. If Op 3 leaves its output in Shared Memory Page 12, and Op 4 reads directly from that page instead of performing an HBM round-trip, the 7.46-microsecond delay effectively falls to about zero.

The “Barrier Penalty” in Op 3 (Attn Reduction) shows a delay of 6.50 microseconds, or about 60.7% overhead. This occurs because the operation must wait for the slowest “Partial Attention” tile to finish before the reduction can continue.

To better visualize this, below is a demonstration of the RMS QKV Rope Append phase on a given processor. While the Consumer Warp (red circle) begins at close to 34 μS, the actual usage of data and when the TMA receives instructions is closer to 36.2 μS.The ~2.2 μs gap between warp scheduling (34 μs) and actual data consumption (36.2 μs) suggests kernel orchestration overhead - likely from synchronization primitives, launch latency, or communication barriers. This represents low-hanging optimization fruit within the current kernel implementation, independent of the broader cross-layer scheduling opportunities.


Processor: 0
Instruction Index: 4
Event: consumer_start
Time: 34.06 us
Duration: 9.24 us


Processor: 0
Instruction Index: 4
Type: RMS QKV MatVec Rope Append
Start: 28.98 us
End: 38.22 us
Duration: 9.24 us

Processor: 0
Instruction Index: 4
Event: real_compute_start
Time: 36.20 us

When calculating the total headroom across the critical path (Ops 1 through 6), the full execution time is approximately 77 microseconds. The sum of all identifiable delay components is 22.22 microseconds (from 2.39 + 1.20 + 6.50 + 7.46 + 2.05 + 2.62). This means that about **28.8%** of a layer's latency could theoretically be eliminated by removing startup stalls and unnecessary waits. Extending this to the entire Llama 1B model with 16 layers, the per-token latency improvement becomes 355.5 microseconds. On hardware with 132 SMs, eliminating this wasted time recovers roughly **47,000** core-microseconds of compute capacity for each generated token.

HBM Limitations[Samvrit Srinath]

To contextualize our measured delays, we can estimate whether the observed stalls represent fundamental hardware bandwidth limits or recoverable inefficiencies:

Op 4 (O Proj) - 7.46 μs Round-Trip Penalty:

- This operation reads attention output (~4096 floats × 4 bytes = 16 KB for a single token)
- At 3 TB/s HBM3 bandwidth, a 16 KB transfer theoretically requires **~5.3 nanoseconds**
- The measured 7.46 μs penalty is **~1,400x higher** than the theoretical minimum
- **Conclusion**: This is NOT bandwidth-limited. The delay stems from kernel launch overhead, synchronization barriers, and TMA instruction dispatch latency—all of which could be eliminated through shared memory persistence.

Op 3 (Attn Reduction) - 6.50 μs Barrier Penalty:

- This operation waits for partial attention results from multiple SMs to synchronize
- The barrier delay is not a memory transfer bottleneck—it's **coordination latency** waiting for the slowest SM's compute to complete
- **Conclusion**: This is compute-skew limited, not bandwidth-limited. Better load balancing across SMs or finer-grained synchronization (avoiding coarse "all-or-nothing" barriers) could reduce this penalty.

Op 1 (RMS QKV) - 2.39 μs Delay:

- The ~2.2 μs orchestration overhead (warp scheduling to data consumption gap) similarly shows no bandwidth constraint
- **Conclusion**: This represents kernel orchestration and synchronization overhead that could be amortized through persistent kernel techniques.

The H100 SXM5's HBM3 memory delivers over 3 TB/sec of bandwidth, which is **significantly underutilized** in our measured stalls. The 22.22 μs of total delay represents coordination and synchronization overhead, not memory bandwidth saturation.

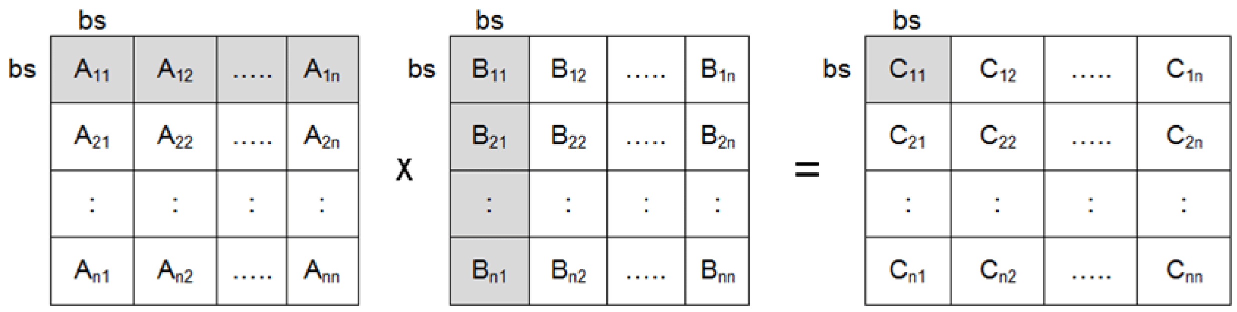
Our analysis shows that HBM3 bandwidth could theoretically service these transfers 100-1,400x faster than observed. This strongly supports our hypothesis that **runtime-managed shared memory** could recover most of this latency by:

1. **Eliminating unnecessary HBM round-trips** (Op 4): Keep data in the 228 KB per-SM shared memory instead of flushing to HBM
2. **Enabling finer-grained synchronization** (Op 3): Replace coarse barriers with dynamic dependency tracking
3. **Reducing kernel orchestration overhead** (Ops 1, 5, 6): Leverage H100's 3x compute throughput improvements and asynchronous execution capabilities to overlap scheduling with computation

Explanation [Srivishnu Ramamurthi]

This "headroom" represents an optimistic view of what could be optimized away. While some barriers are unavoidable—you physically cannot compute an operation until its dependencies exist—ThunderKittens often incurs a "synchronization tax" where barriers persist longer than necessary due to coarse-grained logic and unnecessary global memory trips.

We see this clearly in the transition from **Attention Reduction** to **Output Projection** (e.g., in the batched implementation of Llama). The output projection is essentially a matrix multiplication (C = A X B), where A is the output of the attention reduction and B is the projection weights, as shown in the image below.


bs
bs
A_{11} A_{12} ..... A_{1n}
A_{21} A_{22} ..... A_{2n}
A_{n1} A_{n2} ..... A_{nn}
X
B_{11} B_{12} ..... B_{1n}
B_{21} B_{22} ..... B_{2n}
B_{n1} B_{n2} ..... B_{nn}
=
C_{11} C_{12} ..... C_{1n}
C_{21} C_{22} ..... C_{2n}
C_{n1} C_{n2} ..... C_{nn}

1. The Coarse-Grained Barrier Problem

In the standard ThunderKittens model, synchronization is "all-or-nothing."

- **The Scenario:** Let's say an SM is tasked with calculating block C11 of the output. Mathematically, this relies on the dot product of Row A1 and Column B1.
- **The Bottleneck:** Currently, this calculation cannot start until the previous attention decode kernel has finished calculating **the entire Row A1**
- **The Missed Opportunity:** Theoretically, computation should be fluid. As soon as any sub-block (e.g., A12) is ready, the SM could immediately begin calculating the partial product (A12 X B21). ThunderKittens cannot support this fine-grained interleaving due to its lack of dynamic shared memory management; it effectively forces the consumer to wait for the full "batch" of producer data before lifting a finger.

2. The Global Memory Round-Trip

ThunderKittens also suffers from a lack of "locality awareness" between operations.

- **The Scenario:** Consider the case where the *same* SM that is scheduled to calculate C11 (Output Projection) was also responsible for calculating A11 (Attention Reduction) in the previous step.
- **The Bottleneck:** Ideally, the SM would keep A11 in fast shared memory, distributed shared memory, or registers and immediately use it to compute C11. Instead, ThunderKittens enforces a rigid static dependency structure: it flushes A11 out to slow Global Memory (HBM), triggers a synchronization barrier, and then forces the SM to **reload** that exact same data back from HBM.
- **The Result:** This unnecessary round-trip saturates memory bandwidth and adds latency that a smarter, dynamic scheduler would completely eliminate.

Our theory is that a significant portion of the theoretical performance gap comes from these structural inefficiencies. Here is where the static TK model fails, and where our dynamic approach can win.

Conclusion and Next Steps [Srivishnu Ramamurthi]

Through our work so far, we have explored several methodologies for using ThunderKittens to construct megakernels, developed a deeper understanding of ThunderKittens' internal model, identified its key performance limitations, and conducted timing analysis on a complex kernel (Llama-1B). From this, two core insights emerged:

1. **ThunderKittens cannot currently reach peak performance due to memory bottlenecks.**
2. **There is substantial room to optimize memory movement—particularly interop—within TK kernels.**
   Our profiling and instrumentation work highlights this, and our deeper analysis of an even more complex kernel (Llama-8B with batching), presented in the appendix, reinforces these opportunities in matrix–matrix scenarios.

These findings point to a clear direction for our runtime-managed shared-memory megakernel architecture: we should focus on optimizing the runtime memory management system to enable more efficient interop memory sharing, which our global queue can naturally be extended to support. Strengthening this layer should directly address the bottlenecks we have observed.

As we prepare for the official blog post, our next steps are to establish controlled baseline experiments that run on both ThunderKittens and our alternative methodology. To do this we will be replicating the kernel MLP_Micro, an up-projection followed by a down-projection (a matrix–vector multiply), before extending the demonstration to a kernel we will create, Batched_MLP_Micro, which performs a batch of up/down projections (a matrix–matrix multiply).

We selected these two kernels because our disassembly of complex kernels shows that most megakernel structures simplify down to matrix–vector and matrix–matrix multiplications. Thus, **MLP_Micro and Batched_MLP_Micro form strong, representative baselines** for comparing ThunderKittens with our proposed improvements.

Appendix

Analysis of Batch Llama

```
kvm<llama_config,
   llama_8b_globals,
   attn_norm_op,           // Operation 1: Attention normalization (rms_norm)
   qkv_rope_append_op,     // Operation 2: QKV + RoPE [matmul + rope]
   attention_decode_op,    // Operation 3: Decode attention
   o_proj_op,              // Operation 4: Output projection (matmul_adds) [matmul +
residual add]
   mlp_norm_op,            // Operation 5: MLP normalization (rms_norm)
   gate_silu_op,          // Operation 6: Gate + SiLU [matmul + SiLU]
   up_matmul_op,          // Operation 7: Up MatMul [matmul]
   downproj_op,           // Operation 8: Down projection (matmul_adds) [matmul +
residual add]
   lm_head_norm_op,       // Operation 9: LM head normalization (rms_norm)
   lm_head_op             // Operation 10: LM head MatMul [matmul]
>
```

We're looking at LLama 8B.
Architecture - decoder only, 32 layers, hidden dim is 4096. Each attention layer does all of the operations stated above. Within the attention decode, we also split between attention heads - each with a dimension of 128 - this means we have 4096 / 128 = 2^5 = 32 heads.
This implementation handles decode only - no prefill. In non batch llama, prefill was handled with pytorch, and thunderkittens was only used for decode as well.
This means the input matrix dimension is # of inputs x hidden dimension, not # of inputs x sequence length x hidden dimension (decode does not require previous tokens - kv cache supplies all required previous tokens).

RMS_Norm

RMS_Norm operation takes in a matrix of (num inputs) x (hidden dimension).
The number of inputs is then split across the thread blocks (or SMs, as this is a persistent kernel), with each thread calculating on average num inputs / num SMs.
So if we had 256 inputs, and 128 SMs, each thread block would calculate 2 inputs.
RMS_Norm is done over the activations of an input - so the kernel essentially loads the whole input, calculates the RMS norm across the warps, and then stores it back to global memory.
RMS_Norm requires a distinct scaling vector per each operation. This is a vector of dimension 2048 that is TMA loaded in the loader loop.
When RMS norm instructions are generated, each SM will get an instruction that states how many inputs to process - the scaling vector is reused across these inputs.
If for some reason the instruction generation splits up the RMS norm into multiple instructions, you would not get memory reuse here.

KQV Rope Append

Here we use grouped query attention, so we have 32 attention heads (Qs), and 8 KV heads.
This operation calculates the KQV values for each input, and then it calculates RoPE for K and Q, and finally appends K and V to the KV cache.

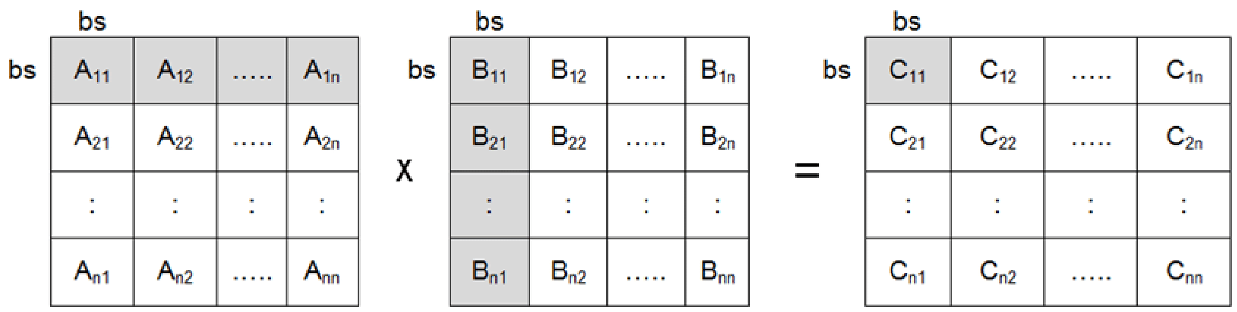
The main matrix multiplication is
# of inputs * hidden dimension @ hidden dimension * kqv

Where the dimension of kqv is given by (num of q heads + num of k heads + num of v heads) * head dimension.
The number of Q, K, and V heads are independent because we are using grouped query attention. In this schema, k and v heads are reused per Q.

So with an example size of 1024 batch size, 4096 hidden dimension, 32 q heads, and 8 k and v heads, and 128 head dim, you have dimensions

1024 x 4096 @ 4096 x 64,000 ((32 + 16) * 128)

Now - for the meat of it - matmul. We do a tiled (/ blocked matmul), and blocks are assigned to SMs. This is done in the instruction scheduling (look at qkv_rope_append.py). The way this is assigned is there is a BATCH_BLOCK_SIZE, let's take the example from qkv_rope_append and say it is 256. This means each block of work an SM has to execute is responsible for 256x256 elements of the output.
So let's say we're calculating C11, as in this image:



Here, A corresponds to the batched hidden vector matrix, and B corresponds to the QKV matrix. C is calculated by iteratively adding A11 @ B11 + A12 @ B21 + A13 @ B31 + … = C11. This equation is what each SM is responsible for calculating.
Parallelism here happens by using TMA to load A11, B11, A12, B21, A13, B31 when A11 @ B11 is being calculated (pipeline depth of 3).

Potential interop memory reuse:

- Notice how an input is fully loaded across its hidden dimension through the course of the matrix multiply - more optimal memory usage could be done by making the SM that would calculate an input in RMS_Norm calculate a batch that requires that entire input and keeping that input in shared memory instead of having to fetch it from memory again.

After this, we do the RoPe embedding - this is done by every SM loading the same row from the RoPE matrix (via TMA), and doing more calculations.

Finally, you append K and V to the global memory for the KV cache, and store Q in global memory as well.

Attention

Based on attention_decode.py - let's say we have a batch size of 128, batch block size of 4, head dim of 128, 32 Qs and 8 K and Vs.

What happens in this step is each instruction is associated with 4 Qs (out of 32, because the ratio of Q to K / V is 4 since this is grouped query attention), and 1 K/V index for 4 inputs (4 rows). The SM will then execute the Qs one by one, doing Q * K + V for every K value in the KV cache including its own. These instructions are then dispersed across the SMs. Intraoperative, there's a 6 stage pipeline for loading old KV blocks. One thing to keep in mind is that the amount of KV blocks change - so if you have a small KV cache - this can be aggressively blocking memory reuse as you would be preallocating 6 pages for the kv cache. This might just be an engineering problem, but I think our idea can be better here.

Potential interop memory reuse:

- Within two instructions of attention itself, with scheduler / megakernel co-optimization you can have great memory reuse (with modifications to the compute kernel as well). Lets say in an SM, we schedule it to take inputs 0-3, and do query 0-3 with K/V 0. It will then load all the previous K/V values associated with K/V 0, and let's say it loads from blocks 0-127. Once we are done with this instruction, we will have K/V cache pages 121-127 (cause 6 stage pipeline) stored in memory. If we then make the next instruction be inputs 4-7, but again with query 0-3 with K/V 0, we can reuse these pages - with online softmax, order of processing does not matter!
- Technically, we can have interop reuse here as well if we make it so that the SM that calculates block C11 (from previous operation) calculates inputs 0-3 with query 0-3 w/ K/V 0 - as we would be able to reuse the query from memory (K/V 0 will probably be a column much further down).

The output of this entire operation is a batch size * hidden dim matrix again. In this case, each instruction contributes a 4 x 512 (128 head dim * 4 Qs) block.

Rest of operations

They're all just matmul or rms_norm - and they show similar behaviours as the operations shown before - interop memory sharing is definitely possible.