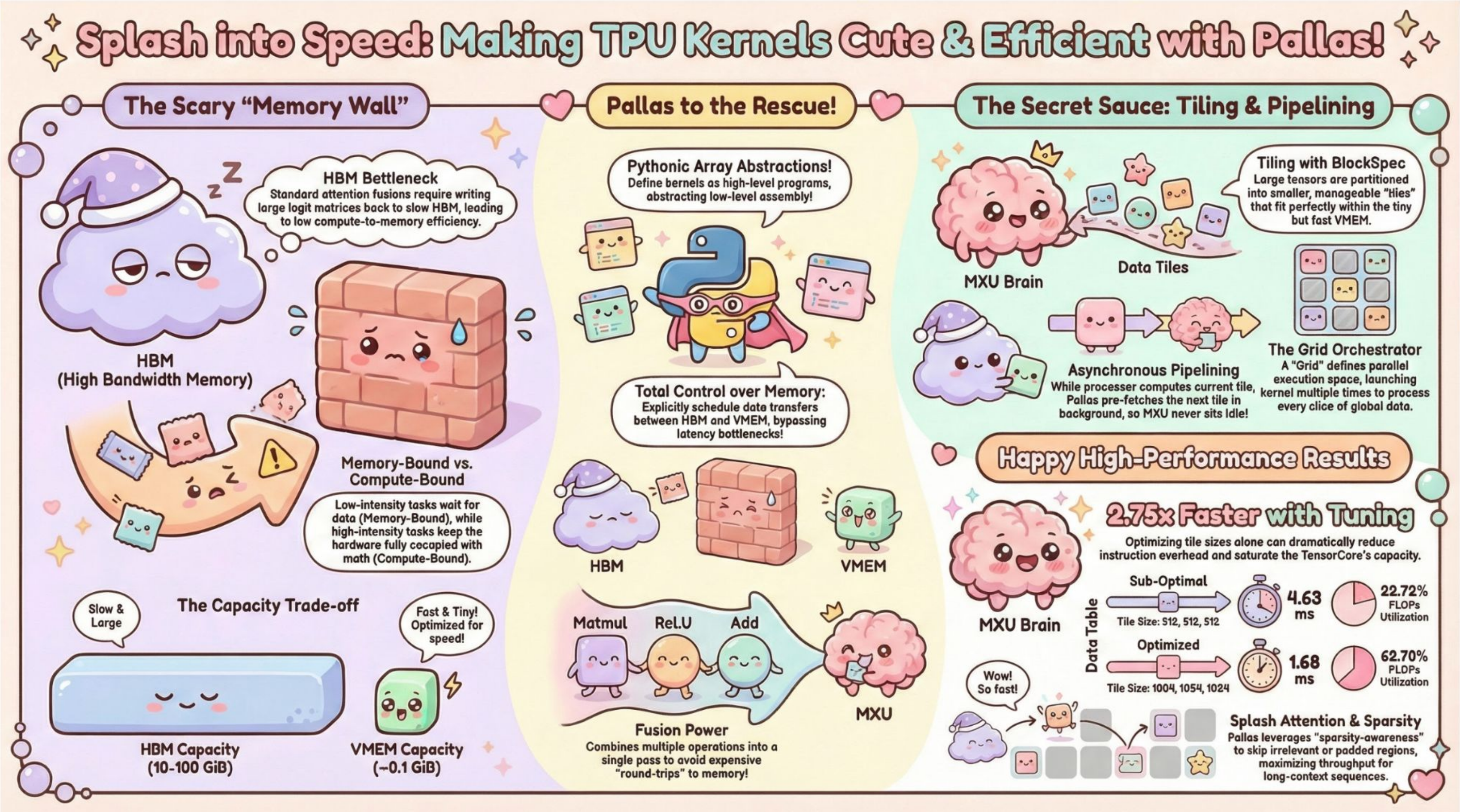
Pallas Kernels
Splash Attention

Srinath Mandalapu
CoreML Frameworks - Google

Agenda

- **Motivation for Custom Kernels**
  - HBM Bottleneck & Memory Wall
- **Pallas Fundamentals**
  - Memory Hierarchy (HBM, VMEM, SMEM)
  - BlockSpec Partitioning, Asynchronous Pipelining
- **Flash Attention Architecture**
  - Tiled execution over Q, K, and V, Online Softmax
  - Fused Key Ops & Delayed Normalization
- **Splash Attention & Sparsity**
  - Sparse Execution Map (MaskInfo)
  - Joint Masking (Causal, Local, & Segment IDs)
  - Performance Tuning (Tile Sizes)



Splash into Speed: Making TPU Kernels Cute & Efficient with Pallas!
The Scary "Memory Wall"
HBM Bettleneck
Standard attention fusions require writing large logit matrices back to slow HBM, leading to low compute-to-memory efficiency.
HBM (High Bandwidth Memory)
Memory-Bound vs. Compute-Bound
Low-intensity tasks wait for data (Memory-Bound), while high-intensity tasks keep the hardware fully cecapied with math (Compute-Bound).
The Capacity Trade-off
Slow & Large
Fast & Tiny! Optimized for speed!
HBM Capacity (10-100 GiB)
VMEM Capacity (~0.1 GiB)
Pallas to the Rescue!
Pythonic Array Abstractions!
Define bernels as high-level programs, abstracting low-level assembly!
Total Control over Memory:
Explicitly schedule data transfers between HBM and VMEM, bypassing latency bottlenecks!
HBM
VMEM
Matmul
ReLU
Add
MXU
Fusion Power
Combines multiple operations into a single pass to avoid expensive "round-trips" to memory!
The Secret Sauce: Tiling & Pipelining
MXU Brain
Data Tiles
Tiling with BlockSpec
Large tensors are partitioned into smaller, manageable "Hies" that fit perfectly within the tiny but fast VMEM.
Asynchronous Pipelining
While processer computes current tile, Pallas pre-fetches the next tile in background, so MXU never sits idle!
The Grid Orchestrator
A "Grid" defines parallel execution space, launching kernel multiple times to process every clice of global data.
Happy High-Performance Results
2.75x Faster with Tuning
Optimizing tile sizes alone can dramatically reduce instruction everhead and saturate the TensorCore's capacity.
MXU Brain
Data Table
Sub-Optimal
Tile Size: 512, 512, 512
4.63 ms
22.72% FLOPs Utilization
Optimized
Tile Size: 1004, 1054, 1024
1.68 ms
62.70% PLOPs Utilization
Wow! So fast!
Splash Attention & Sparsity
Pallas leverages "sparsity-awareness" to skip irrelevant or padded regions, maximizing throughput for long-context sequences.

Motivation

Moving Beyond XLA Fusions

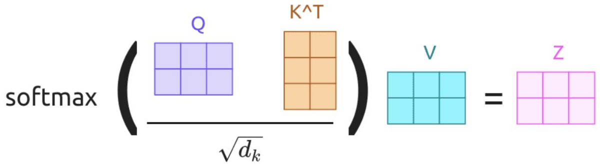
The HBM Bottleneck: While XLA successfully fuses basic ops, standard Attention fusions often require the large logit matrix to be written back to High Bandwidth Memory (HBM) for the Softmax reduction, leading to low compute-to-memory efficiency.

Vector Unit Limitations: Standard JIT-compiled fusions can leave the Matrix Execution Unit (MXU) idle while the Vector Processing Unit (VPU) handles complex exponential and sum operations, resulting in a low percentage of peak FLOPs.

VMEM Residency & Loop Control: Pallas provides direct control over the iteration space and data movement between HBM and VMEM. This enables "Online Softmax" algorithms that keep data local, bypassing the HBM "memory wall" that limits standard XLA fusions.

Temporal Control via LLO: You manage the temporal flow (scheduling *when* compute occurs), while the backend handles the spatial layout (T(8,128) tiling). By bypassing HLO and lowering from Palls Custom Call → MLIR → LLO, Pallas prevents the compiler from reordering your manual optimizations or "undoing" your kernel orchestration.

Recap: Simple Attention Mechanis

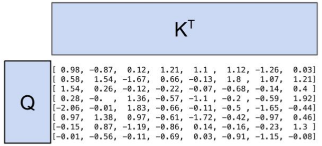


- **Core Mechanism:** Attention weighs input parts, enabling models to focus on relevant information by computing a weighted sum.
  - **O = softmax(QKᵀ / √d□) V**
    - Q = Query, K = Key, V = Value,
    - d□ = Key dimension, O = Output
- **Causal Masking:** Causal masks are applied to QKᵀ in decoders to prevent attending to future tokens, before the softmax function.
  - If **mask[i, j] = 0**, then **(QKᵀ)[i, j] = -∞**
  - Where i is the query position and j is the key pos.
- **Regularization:** Dropout (p) for regularization is applied to the softmax output before multiplying by V.
  - **O = dropout(softmax(QKᵀ / √d□), p) V**

causal attention mask

```
[1, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 1, 1, 1]
```

Simple Example:
8 tokens



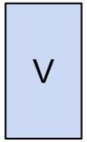

```
[ 0.98, -0.87,  0.12,  1.21,  1.1 ,  1.12, -1.26,  0.03]
[ 0.58,  1.54, -1.67,  0.66, -0.13,  1.8 ,  1.07,  1.21]
[ 1.54,  0.26, -0.12, -0.22, -0.07, -0.68, -0.14,  0.4 ]
[ 0.28, -0.  ,  1.36, -0.57, -1.1 , -0.2 , -0.59,  1.92]
[-2.06, -0.01,  1.83, -0.66, -0.11, -0.5 , -1.65, -0.44]
[ 0.97,  1.38,  0.97, -0.61, -1.72, -0.42, -0.97,  0.46]
[-0.15,  0.87, -1.19, -0.86,  0.14, -0.16, -0.23,  1.3 ]
[-0.01, -0.56, -0.11, -0.69,  0.03, -0.91, -1.15, -0.08]
```

jnp.where(mask, 0, -jnp.inf)

```
[  0., -inf, -inf, -inf, -inf, -inf, -inf, -inf]
[  0.,   0., -inf, -inf, -inf, -inf, -inf, -inf]
[  0.,   0.,   0., -inf, -inf, -inf, -inf, -inf]
[  0.,   0.,   0.,   0., -inf, -inf, -inf, -inf]
[  0.,   0.,   0.,   0.,   0., -inf, -inf, -inf]
[  0.,   0.,   0.,   0.,   0.,   0., -inf, -inf]
[  0.,   0.,   0.,   0.,   0.,   0.,   0., -inf]
[  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.]
```

Apply softmax

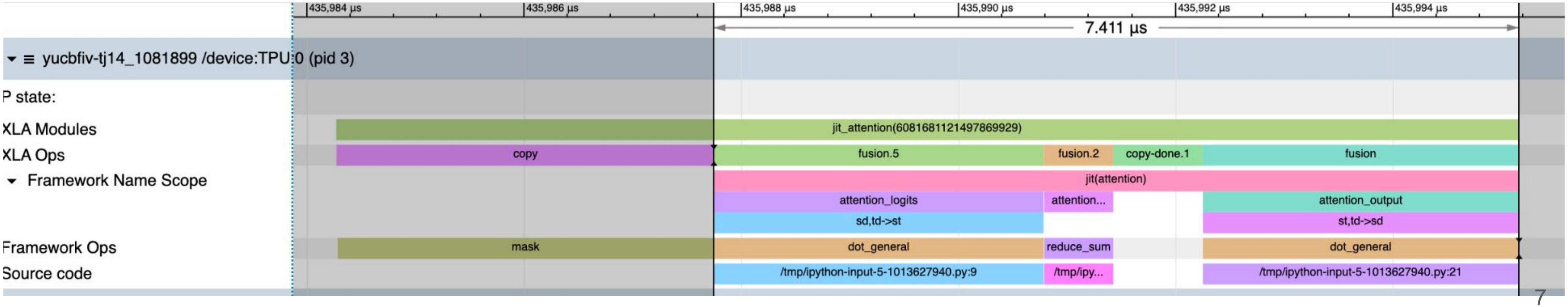
```
[1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]
[0.28, 0.72, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]
[0.68, 0.19, 0.13, 0.  , 0.  , 0.  , 0.  , 0.  ]
[0.2 , 0.15, 0.57, 0.08, 0.  , 0.  , 0.  , 0.  ]
[0.01, 0.11, 0.71, 0.06, 0.1 , 0.  , 0.  , 0.  ]
[0.25, 0.37, 0.25, 0.05, 0.02, 0.06, 0.  , 0.  ]
[0.13, 0.35, 0.04, 0.06, 0.17, 0.13, 0.12, 0.  ]
[0.18, 0.1 , 0.16, 0.09, 0.18, 0.07, 0.06, 0.16]
```



Basic Attention

```python
import jax
import jax.numpy as jnp

# Simplified Attention
def attention(mask, q, k, v, mask_val=-1e38):

    # Q @ K^T
    with jax.named_scope("attention_logits"):
        logits = jnp.einsum("sd,td->st", q, k)
        logits = jnp.where(mask, logits, mask_val)

    # Softmax
    with jax.named_scope("attention_softmax"):
        m = jnp.max(logits, axis=-1, keepdims=True)
        s = jnp.exp(logits - m)
        l = jnp.sum(s, axis=-1, keepdims=True)
        probs = s / l

    # S @ V
    with jax.named_scope("attention_output"):
        out = jnp.einsum("st,td->sd", probs, v)

    return out
```

```python
# Create inputs
key = jax.random.PRNGKey(0)
kq, kk, kv = jax.random.split(key, 3)

s, d = 1024, 512
q = jax.random.normal(kq, (s, d), dtype=jnp.float32)
k = jax.random.normal(kk, (s, d), dtype=jnp.float32)
v = jax.random.normal(kv, (s, d), dtype=jnp.float32)

# Boolean mask
mask = jnp.tril(jnp.ones((s, s), dtype=jnp.bool_))

# JIT and execute
apply_attn = jax.jit(attention)
output = apply_attn(mask, q, k, v)
```



Summary of XLA Fusions for Attention

Fusion Name	Description & Key Steps	Primary Inputs & Outputs
Logit & Max Fusion (%fused_computation.7)	**Logit Generation & max :** Performs the first matrix contraction (Q×K^T), applies the attention mask, and immediately finds the row-wise maximum to prepare for numerical stability.	In: Q (1024x512), K(1024x512), Mask (1024×1024) Out: Max Logits (1024), Masked Logits (1024×1024)
Softmax Denominator (%fused_computation.3)	**Exp & Reduction:** Subtracts the row maximums from the logits, computes the hardware-native base-2 exponential (e^(x−max)), and reduces the results via summation to create the Softmax denominator.	In: Masked Logits (1024x1024), Max Logits(1024) Out: Sum of Exponentials (1024)
Attention Output (%fused_computation)	**Rematerialize & Project:** Re-calculates e^{(x-max)} and divides by the sum to generate probabilities. These results are fed directly into the MXU for the final PV contraction.	In: Value V(1024x512), Masked Logits (1024x1024), Max Logits (1024), Sum of Exponentials (1024) Out: Context Vector (1024×512)

Simple Attention Fusions Summary

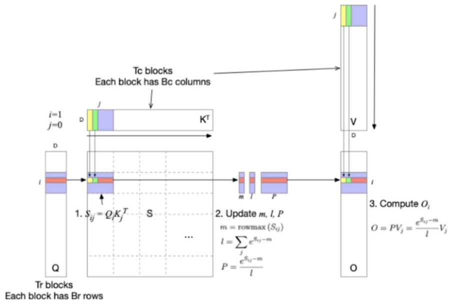
TPU Ironwood

Peak FLOP Rate per TensorCore: **1028.75 TFLOP/s**
Peak HBM Bandwidth per TensorCore: **3433 GiB/s**
Peak VMEM Read Bandwidth per TensorCore: 27180 GiB/s
Peak VMEM Write Bandwidth per TensorCore: 19932 GiB/s

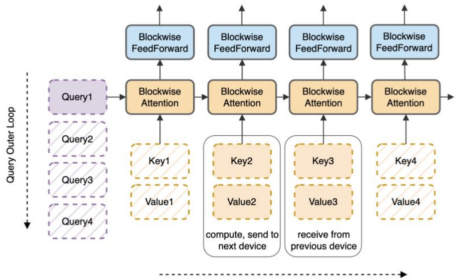
Metric	Fusion 1: Logit & Max	Fusion 2: Softmax Denom	Fusion 3: Attention Output
Primary Task	Q×K^T, Mask, Row-Max Q(1024, 512) K(1024, 512) Flops = 2 * 1024* 512*1024	e^(x−max) and Row-Sum	P×V Projection P(1024, 1024) V(1024, 512)
Key Inputs	Q,K, Mask	Masked Logits, Max Logits	V, Masked Logits, Max, Sum
Key Outputs	Max Logits, Masked Logits	Sum of Exponentials	Context Vector
Avg Execution Time	3.04 us	636.25 ns	2.92 us
FLOPS Utilization	**34.45% (of 1028.75)**	**0.32%**	**35.82%**
FLOP Rate (Core)	**354.36 TFLOP/s**	**3.29 TFLOP/s**	**368.49 TFLOP/s**
HBM Bandwidth Util	**18.74% (of 3433 GiB/s)**	**0.00% (On-Chip only)**	**19.49%**
On-Chip Read Util	**3.55%(of 27180 GiB/)**	**22.61%**	**7.39%**

- **Compute Throughput & MXU Efficiency:** Fusions 1 and 3 leverage the Matrix Execution Unit (MXU) to achieve over **350 TFLOP/s** (~35% of peak), calculating 2mkn operations (2x1024x512x1024) in ~3mus while processing masks and biases "for free".
- **SRAM Residency & HBM Bypass:** XLA achieves **0.00% HBM utilization** in Fusion 2 by keeping the 1024x1024 logit matrix resident in **VMEM**, bypassing the 3,433 GiB/s HBM bottleneck and allowing for a high-speed local reduction.
- **Vector Unit Bottleneck & VMEM Demand:** The shift to the VPU for Softmax reduces FLOPS to **0.32%** but spikes On-Chip Read utilization to **22.61%**, as the VPU consumes data from VMEM at **6.60 TB/s** to perform exponentials and row-sums.

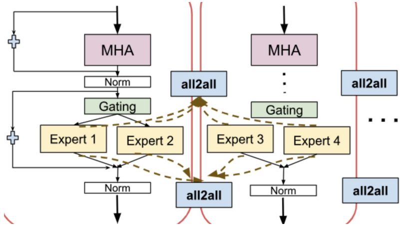
Pallas Kernels in LLM Training



Splash Attention (Fusion & Sparsity): Optimizes memory by fusing multiple computational stages into a single kernel to avoid HBM bottlenecks. It leverages "sparsity-awareness" to skip irrelevant or padded regions, maximizing throughput for long-context sequences.



Ring Attention (Distributed Context): Distributes the attention calculation across a device mesh by "ringing" key-value blocks between devices. This enables processing of massive sequence lengths that exceed the memory capacity of a single TPU node.

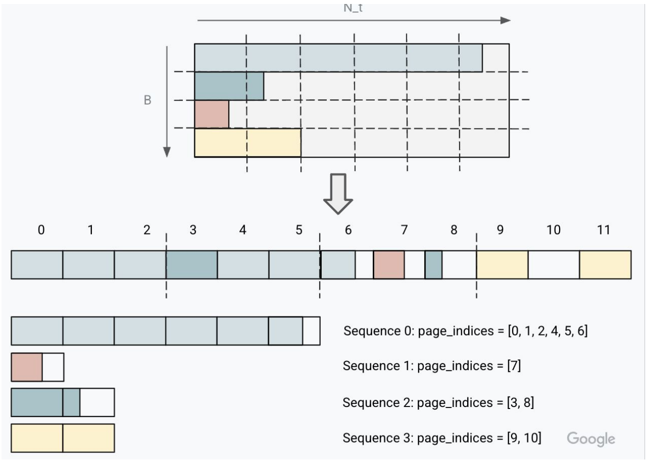


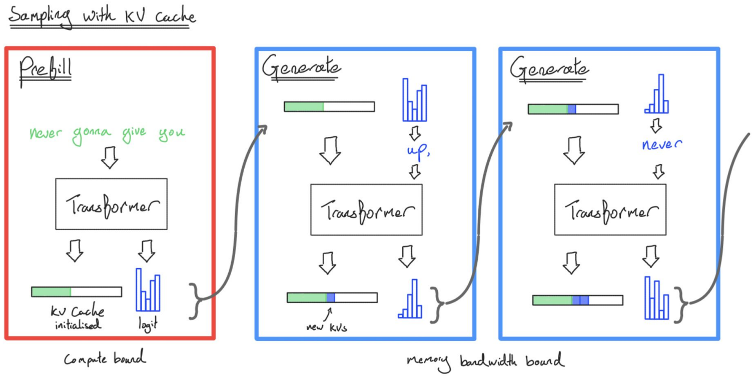
MoE Optimization (Ragged All-to-All & GMM): Replaces standard MLP layers with sparse experts. Specialized **Ragged All-to-All** kernels handle variable-sized token exchanges between devices, while **Grouped Matrix Multiplication (GMM)** executes multiple experts in one pass—eliminating padding and the need for "token dropping."

Pallas Kernels in LLM Inference

Paged Memory Mapping: To eliminate padding waste, **Paged Attention** partitions the KV cache into fixed-size physical pages. This allows non-contiguous storage of logical sequences in a global pool, achieving near-100% memory utilization for "ragged" batches.

Pallas for Page Indirection: Since standard XLA cannot traverse non-contiguous page tables, **Pallas kernels** manually orchestrate the pointer indirection. This enables the TPU to load fragmented pages into at peak speeds, bypassing the "memory wall."





Page Attention optimizes memory by mapping logical sequences to a fragmented physical memory, allowing for efficient storage of Key-Value (KV) cache data

- (Example: Four Sequences)

Pallas Kernels Fundamentals

Why Pallas Kernels?

The Abstraction Ceiling: While **JAX JIT** automates HLO fusions, its generic lowering can obscure the "physical reality" of data movement. For complex operations like Attention, the compiler may default to conservative memory patterns that leave the hardware underutilized.

Breaking the Memory Wall: Pallas allows you to explicitly orchestrate the **memory hierarchy**. By manually scheduling data transfers between **HBM** and **VMEM**, you can ensure that the data required for the next computation is pre-fetched, bypassing the latency bottlenecks of automated fusions.

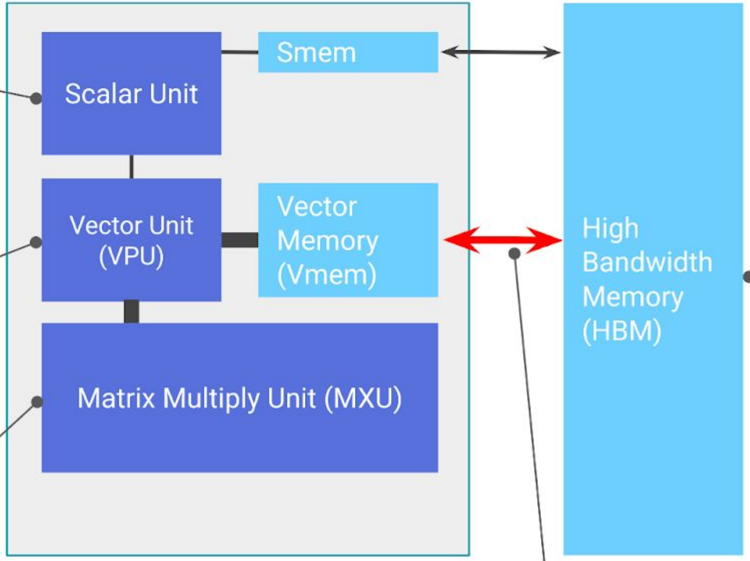
Manual Macro-Tiling & Pipelining: You gain direct control over **loop nesting and block sizes** (macro-tiling). This enables "ping-pong" buffering—where you overlap the loading of the *next* tile with the computation of the *current* tile—ensuring the **MXU** never sits idle waiting for memory.

TPU Tensor Core Layout

The **Scalar Unit** sort of acts like a CPU ‘dispatching’ instructions to the VPU and MXU

The **VPU** performs elementwise operations (e.g. activations), loads data into the MXU

The MXU performs matrix multiplications - and is therefore our driver of chip FLOP/s.

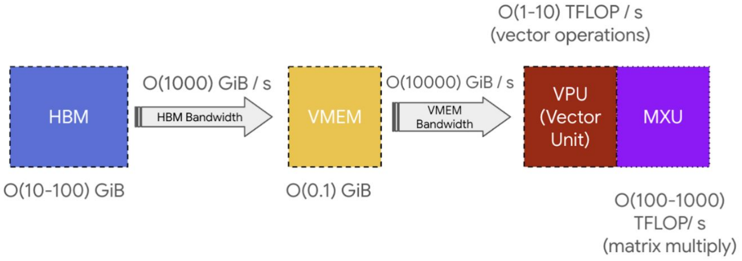


Abstract layout of a TPU TensorCore.

HBM stores the weights, activations, optimiser states, new batch data etc

HBM bandwidth: determines how fast data goes to and from the computational elements

Memory Performance
HBM vs. VMEM



Arithmetic Intensity Defined: This is the ratio of mathematical operations (FLOPs) to the bytes of data moved from memory.

Memory-Bound vs. Compute-Bound: Low-intensity tasks wait for data (memory-bound), while high-intensity tasks keep the hardware fully occupied with math (compute-bound).

The VMEM Speed Advantage: **VMEM** provides significantly higher bandwidth than **HBM**, acting as a high-speed "lane" directly next to the execution units.

Peak Efficiency at Low Intensity: Because data moves faster in VMEM, even data-heavy algorithms with an intensity of **10–20** can reach peak FLOPs.

Capacity Trade-off: VMEM is optimized for speed over size, offering roughly **0.1 GiB** of storage compared to the **10–100 GiB** available in HBM.

TPU Memory Spaces

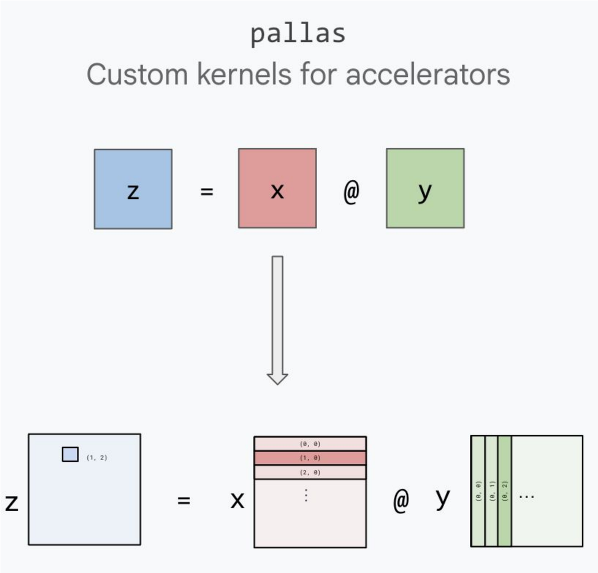
Pallas Enum	TPU Memory Space	Description
pltpu.MemorySpace.**ANY**	HBM (or VMEM)	A default memory space used when the exact location doesn't matter, typically resolves to HBM for large tensors.
pltpu.MemorySpace.**VMEM**	Vector Memory (VMEM)	Fast, on-chip SRAM for the Vector Unit (matrix/vector operations). It's smaller than HBM but much faster.
pltpu.MemorySpace.**SMEM**	Scalar Memory (SMEM)	Fast, on-chip SRAM for the Scalar Unit. Used for storing small, temporary scalar and control flow values.
pltpu.MemorySpace.**SEMAPHORE**	Semaphore Memory	A very small, dedicated SRAM region used for inter-core synchronization (e.g., locking) between computation units.

Pallas Abstractions

Pythonic Array Abstraction: Pallas provides a **Python-based API** that lets you define kernels as high-level array programs. It abstracts away low-level synchronization, allowing you to focus on the logic of the computation rather than hardware-specific assembly.

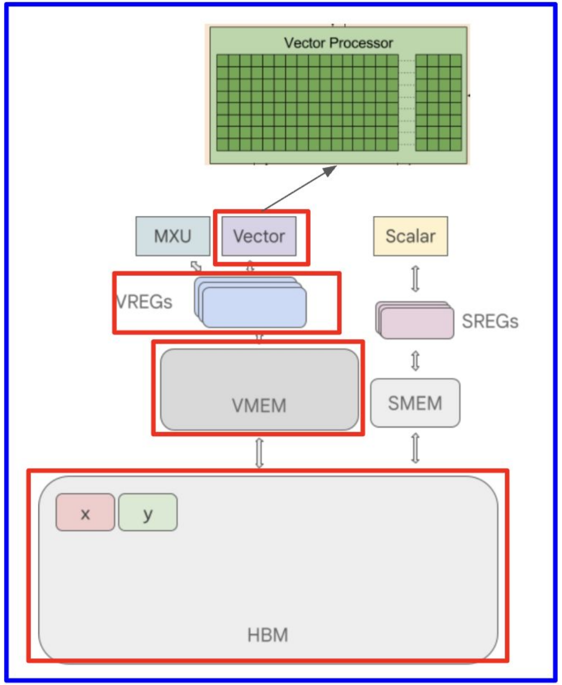
Automated Data Orchestration: The language handles the heavy lifting of **HBM ↔ VMEM transfers** and the **pipelining** of memory movement with execution, removing the need for manual DMA management or explicit memory staging.

Managed Hardware Scaling: Pallas automatically maps your Python logic across the hardware, handling **parallelization over TPU cores** and generating optimized code that balances data flow and compute efficiency.



Simple Matrix Addition Kernel

```python
def add_matrices_kernel(x_vmem_ref, y_vmem_ref, z_vmem_ref):
  # Load x and y from VMEM into registers
  x_regs = x_vmem_ref[:, :]
  y_regs = y_vmem_ref[:, :]
  # Execute a vectorized add
  z_regs = x_regs + y_regs
  # Store the output values in registers back into VMEM
  z_vmem_ref[:, :] = z_regs


def add_matrices(x: jax.Array, y: jax.Array) -> jax.Array:
  # pallas_call will first allocate scratch buffers for `x` and `y` in VMEM.
  # It will then copy `x` and `y` from HBM into VMEM.
  z = pl.pallas_call(
      add_matrices_kernel, out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype)
  )(x, y)
  # pallas_call will also copy the output from VMEM back into HBM.
  return z


x, y = jnp.ones((512, 512)), jnp.ones((512, 512))
add_matrices(x, y)
```



VMEM - Out-of-Memory (OOM)

The VMEM Bottleneck: Simple Pallas (**pallas_call** without **BlockSpec**) attempts to load entire tensors into VMEM (SRAM) simultaneously. Since VMEM is tiny (e.g., 32MB per core), even a standard 2048x2048FP32 matrix (16MB) plus workspace quickly triggers an Out-of-Memory (OOM) error.

Tiling with BlockSpec: To process LLM-sized tensors, you must partition the computation into smaller, manageable chunks. **BlockSpec** allows you to define these tiles, ensuring only the necessary data fragments are resident in VMEM at any given time.

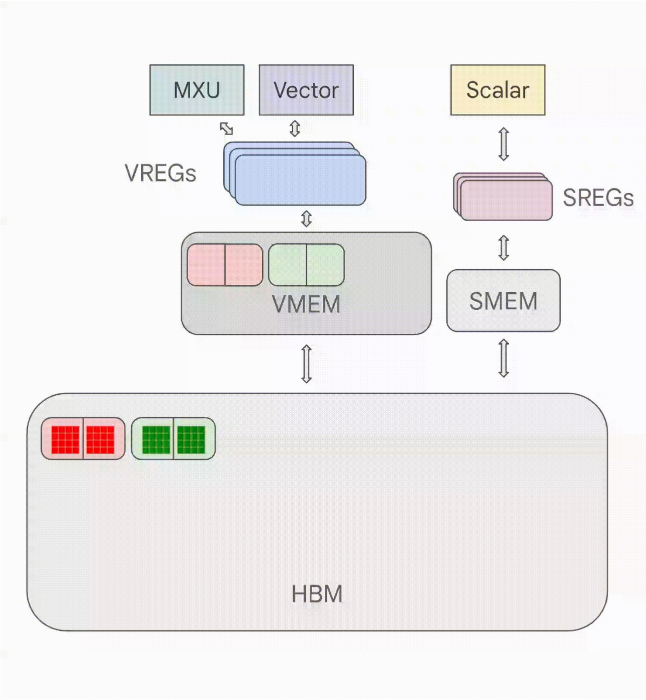
Hiding Latency via Pipelining: Once tiled, Pallas enables **software pipelining** to "hide" slow HBM transfers. While the TPU computes the *current* tile in VMEM, Pallas pre-fetches the *next* tile from HBM in the background, ensuring the compute units never sit idle.

XlaRuntimeError: RESOURCE_EXHAUSTED: **Ran out of memory in memory space vmem** while allocating on stack for %tpu_custom_call.1 = f32[2048,2048]{1,0:T(8,128)} custom-call(%args_0_.1, %args_1_.1), custom_call_target="tpu_custom_call", operand_layout_constraints={f32[2048,2048]{1,0}, f32[2048,2048]{1,0}}, metadata={op_name="jit(wrapped)/pallas_call" source_file="/tmp/ipython-input-6-1996092727.py" source_line=14 source_end_line=17 source_column=6 source_end_column=9}. Scoped allocation with size 48.01M and limit 32.00M exceeded scoped vmem limit by 16.01M. It should not be possible to run out of scoped vmem - please file a bug against XLA.

Pipelining in TPU

- **Efficiency through Parallelism:** TPUs use pipelining to overlap data movement with computation to eliminate idle time.
- **VPU vs. MXU Roles:** While the **MXU** (Matrix Execution Unit) handles large multiplications, the **Vector Processing Unit (VPU)** performs element-wise operations like **addition** and activation functions.
- **Streamlined Data Flow:** High-Bandwidth Memory (HBM) copies data to Vector Memory (**VMEM**), the VPU/MXU executes the op, and results stream back to HBM.
- **Bottleneck Reduction:** Continuous overlapping keeps execution units saturated, preventing memory-bound delays during complex ML workloads.

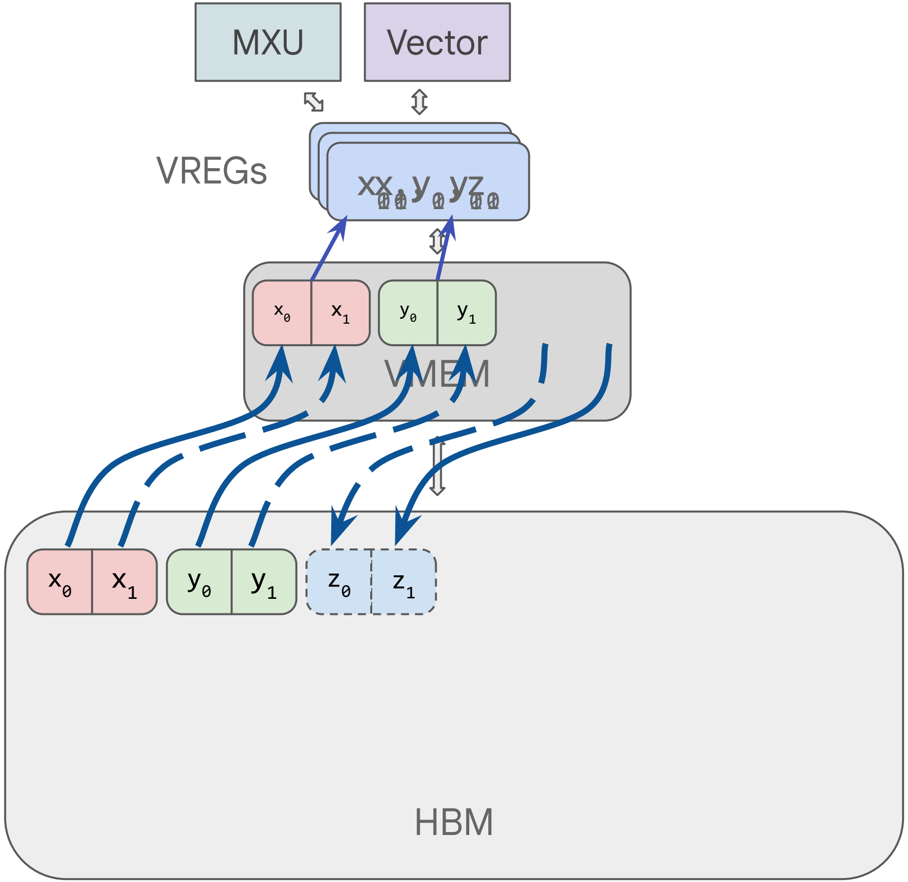
Pipelining - Addition



Pipelining Example

Idea: overlap loading/storing with compute by tiling inputs/outputs

- Allocate output buffer for z
- Allocate VMEM scratch space
- Copy x_0, y_0 from HBM into VMEM
- Start copying x_1, y_1 from HBM into VMEM
- Load x_0, y_0 into VREGs
- Add x_0 and y_0 using vector core
- Store z_0 in VMEM
- Start copying z_0 from VMEM into HBM
- Wait until x_1, y_1 are done copying into VMEM
- Load x_1, y_1 into VREGs
- Add x_1 and y_1 into using vector core



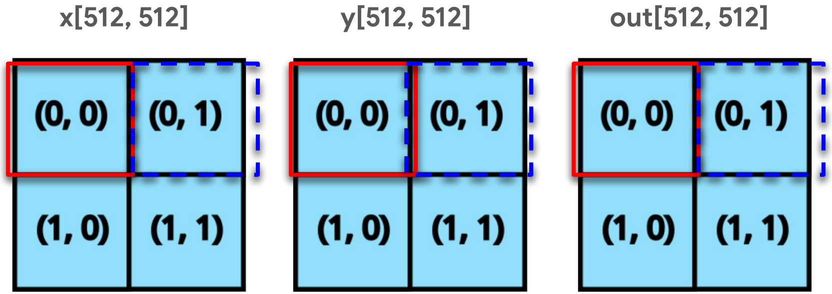
Pallas Pipelining API

The Pallas Pipelining API automates the management of multiple buffers and asynchronous memory transfers, hiding HBM latency by overlapping data movement with active computation.

Component	Role in Pipelining	Key Features
Grid	Defines the Structure of the Pipeline	Defines the loop structure as a tuple (N, M, ...). It dictates the total iteration space, where the kernel is invoked prod(grid) times to solve the global problem.
BlockSpecs (pl.BlockSpec)	Handles Data Communication	Manages data orchestration by defining the block_shape and index_map. It specifies which HBM data slice is copied to VMEM for each grid index.
Kernel	Specifies the Computation Stage	The computational stage that processes a single block. It operates directly on VMEM buffers and uses pl.program_id to identify its current position in the grid.
Pallas Call (pl.pallas_call)	Main Entry Point & Orchestration	The orchestration entry point that binds the kernel, grid, and BlockSpecs together into a single executable pipeline.

Pipelined Sum Kernel

```
def add_matrices_pipelined_param(
    x: jax.Array, y: jax.Array, *, bm: int = 256, bn: int = 256
) -> jax.Array:
  m, n = x.shape
  block_spec = pl.BlockSpec((bm, bn), lambda i, j: (i, j))
  return pl.pallas_call(
      add_matrices_kernel,
      out_shape=x,
      in_specs=[block_spec, block_spec],
      out_specs=block_spec,
      grid=(m // bm, n // bn),
  )(x, y)

x, y = jnp.ones((512, 512)), jnp.ones((512, 512))

np.testing.assert_array_equal(
    add_matrices_pipelined_param(x, y, bm=256, bn=256), x + y
)
```



Block Computation in Pallas

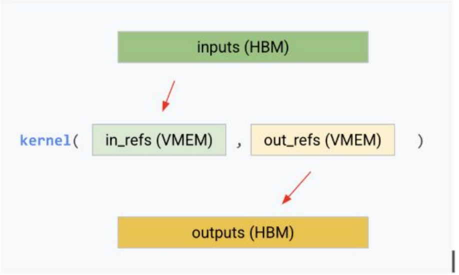
Grid Parallelism: The **grid** parameter defines the iteration space, launching the kernel **prod(grid)** times. This structure allows Pallas to parallelize independent array programs across TPU cores.

BlockSpec Partitioning: **BlockSpec** partitions global tensors into smaller blocks that fit in VMEM. It uses **index maps** to determine exactly which data slice corresponds to each coordinate in the grid.

Pipelined Scheduling: **pallas_call** acts as the orchestrator, scheduling kernel launches and constructing a pipeline that overlaps HBM data transfers with active computation to maximize throughput.

Pallas call scheduler

```
for i in range(grid[0]):
 for j in range(grid[1]):
  ...
  for k in range(grid[-1]):
   in_refs = get_in_refs(in_specs, (i, j, ..., k), *inputs)
   out_refs = get_out_refs(out_specs, (i, j, ..., k))
   kernel((i, j, ..., k), *in_refs *out_refs)
   outputs = update(out_specs, (i, j, ..., k), outputs, out_refs)
```



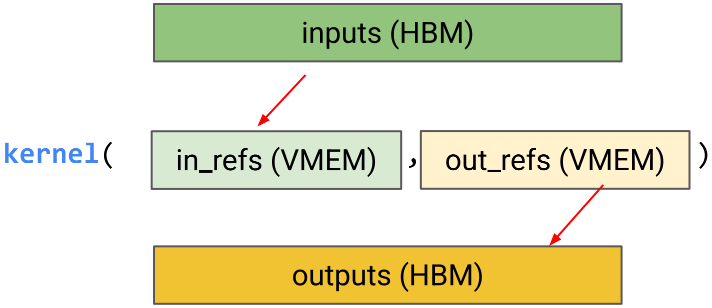
pallas_call: The Pipeline Orchestrator

For-Loop Scheduler: At its core, **pallas_call** acts as a high-performance loop scheduler. It interprets the **grid** as a set of nested loops, orchestrating the execution of kernel instances across the defined iteration space.

Asynchronous Overlap: It automatically emits a pipeline that overlaps **HBM** ↔ **VMEM** data transfers with active computation. This ensures that while one block is being processed by the MXU, the next is already being fetched from memory.

Buffer Reuse & Efficiency: To minimize memory overhead, pallas_call reuses the same **VMEM buffers** for consecutive kernel instances if the in_specs and out_specs remain consistent, effectively eliminating redundant allocations and copies.

```
for i in range(grid[0]):
 for j in range(grid[1]):
  ...
  for k in range(grid[-1]):
   in_refs = get_in_refs(in_specs, (i, j, ..., k), *inputs)
   out_refs = get_out_refs(out_specs, (i, j, ..., k))
   kernel((i, j, ..., k), *in_refs *out_refs)
   outputs = update(out_specs, (i, j, ..., k), outputs, out_refs)
```



Pallas Output Aliasing: In-Place Computation

Input-Output Aliasing: By mapping an output index to an input index (e.g., {2: 0}), Pallas instructs XLA to reuse the input's memory buffer for the output, eliminating the need for new allocations.

In-Place Updates: The kernel overwrites the original input data directly (e.g., out[...] = out[...] + x[...]), performing a true in-place operation that preserves memory space.

Zero-Copy Efficiency: This optimization slashes HBM bandwidth usage and reduces latency by removing the overhead of redundant data copies between memory locations.

```
total_shape = (4096, 4096)
block_shape = (1024, 1024)

def multi_output_kernel(x_ref, y_ref, z1_ref, z1_output, z2_ref):
  z1_output[...] = z1_ref[...] + x_ref[...] + y_ref[...]
  z2_ref[...] = x_ref[...] - y_ref[...]

def multi_output_pipelined(x: jax.Array, y: jax.Array, z1_initial: jax.Array):

  output_shape_struct = jax.ShapeDtypeStruct(x.shape, dtype=jnp.float32)
  output_spec = pl.BlockSpec(block_shape, index_map=lambda i, j: (i, j))

  return pl.pallas_call(
    multi_output_kernel,
    grid=tuple(total // block for (total, block) in zip(total_shape, block_shape)),
    in_specs=[
      pl.BlockSpec(block_shape, index_map=lambda i, j: (i, j)),
      pl.BlockSpec(block_shape, index_map=lambda i, j: (i, j)),
      pl.BlockSpec(block_shape, index_map=lambda i, j: (i, j))
    ],
    out_specs=(output_spec, output_spec),
    out_shape=(output_shape_struct, output_shape_struct),
    input_output_aliases={2: 0},
    debug=False,
  )(x, y, z1_initial)

x = jnp.ones(total_shape, dtype=jnp.float32)
y = jnp.ones(total_shape, dtype=jnp.float32)
z1_initial = jnp.full(total_shape, 100.0, dtype=jnp.float32)
result = multi_output_pipelined(x, y, z1_initial)
```

Two Level Tiling Strategy

Macro-Tiling (Pallas/User): Orchestrates the **HBM ↔ VMEM** boundary. You define large software blocks (e.g., **1024×1024**) to hide high-latency DMA transfers. Pallas manages the grid and pipelines the *next* HBM-to-VMEM fetch while the *current* block is being computed.

Micro-Tiling (Compiler/Hardware): Orchestrates the **VMEM ↔ VPU** boundary. The compiler decomposes the large macro-block into the hardware's native **8×128** processing units. It translates the computation into a tight loop of low-level Vector Load, ALU, and Store instructions.

HBM (DMA/Macro-Tiling) ↔VMEM (Vector Load/Store Micro-Tiling) VREGS →VPU

Compiler MLIR & Hardware Alignment

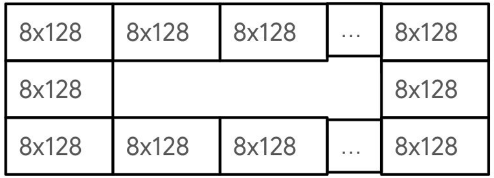
The MLIR Bridge: MLIR is a compiler framework that translates your Python code into hardware instructions through a series of "dialects" that understand both high-level logic and low-level TPU limits.

Vector Abstraction: Instead of complex loops, the compiler uses a simplified vector.load to treat large blocks (e.g., 1024x1024) as single units, letting the backend handle the tedious hardware scheduling.

Physical Layout (T(8,128)): Data is organized in memory to match the hardware's native **8×128** processing size. This alignment ensures that every memory transfer feeds the VPU at peak efficiency.

Uniform Padding: Pallas automatically pads irregular matrix edges with zeros. This guarantees every block is the same size, which avoids slow "if-else" branching logic in the hardware.

f32[32768,32768]{1,0:T(8,128)}




```
module @add_matrices_pipelined_kernel {
  func.func @main(%arg0: i32, %arg1: i32, %arg2: memref<1024x1024xf32, #tpu.
  memory_space<vmem>>, %arg3: memref<1024x1024xf32, #tpu.memory_space<vmem>>,
  %arg4: memref<1024x1024xf32, #tpu.memory_space<vmem>>) attributes
  {dimension_semantics = [#tpu.dimension_semantics<arbitrary>, #tpu.
  dimension_semantics<arbitrary>], iteration_bounds = array<i64: 32, 32>,
  scalar_prefetch = 0 : i64, scratch_operands = 0 : i64, window_params =
  [{transform_indices = @transform_0, window_bounds = array<i64: 1024, 1024>},
  {transform_indices = @transform_1, window_bounds = array<i64: 1024, 1024>},
  {transform_indices = @transform_2, window_bounds = array<i64: 1024, 1024>}]} {
    %c0 = arith.constant 0 : index
    %c0_0 = arith.constant 0 : index
    %0 = vector.load %arg2[%c0, %c0_0] : memref<1024x1024xf32, #tpu.
    memory_space<vmem>>, vector<1024x1024xf32>
    %c0_1 = arith.constant 0 : index
    %c0_2 = arith.constant 0 : index
    %1 = vector.load %arg3[%c0_1, %c0_2] : memref<1024x1024xf32, #tpu.
    memory_space<vmem>>, vector<1024x1024xf32>
    %2 = arith.addf %0, %1 : vector<1024x1024xf32>
    %c0_3 = arith.constant 0 : index
    %c0_4 = arith.constant 0 : index
    %3 = vector.load %arg4[%c0_3, %c0_4] : memref<1024x1024xf32, #tpu.
    memory_space<vmem>>, vector<1024x1024xf32>
    tpu.vector_store %arg4[%c0_3, %c0_4], %2 {strides = array<i32>} :
    memref<1024x1024xf32, #tpu.memory_space<vmem>>, vector<1024x1024xf32>,
    return
  }
  func.func @transform_0(%arg0: i32, %arg1: i32) -> (i32, i32) {
    %c0_i32 = arith.constant 0 : i32
    return %arg0, %arg1 : i32, i32
  }
  func.func @transform_1(%arg0: i32, %arg1: i32) -> (i32, i32) {
    %c0_i32 = arith.constant 0 : i32
    return %arg0, %arg1 : i32, i32
  }
  func.func @transform_2(%arg0: i32, %arg1: i32) -> (i32, i32) {
    %c0_i32 = arith.constant 0 : i32
    return %arg0, %arg1 : i32, i32
  }
}
```

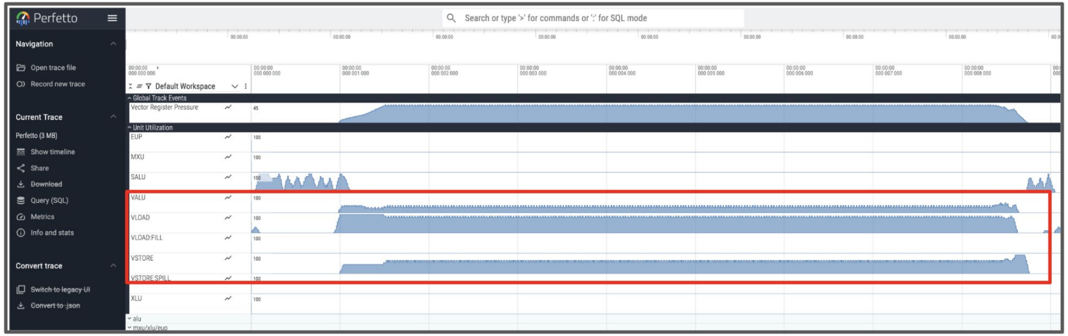
- Load x,y and result blocks [1024, 1024] from VMEM to regs
- Perform sum (VPU) (8x128 tiles)
- Finally write sum to VMEM

Block Spec - index map transformations for x, y and result

VPU Pipelining

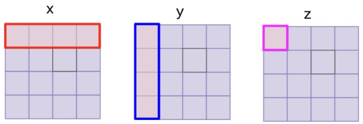
- VLOAD **(Vector Load):** Fetching inputs (X and Y) for the **next chunk (8x128) (N+1)** from Vector Memory (VMEM) and moving them into the Vector Registers (VRegs).
- VALU **(Vector ALU):** Performing the core element-wise computation (the X+Y) for the **current chunk (N)** on the Vector Processor Unit.
- VSTORE **(Vector Store):** Writing the computed result (O) for the **previously completed chunk (N–1)** from VRegs back to VMEM.

The O=X+Y addition is a **VMem bandwidth-bound** operation. With an operational intensity of only 1 FLOP/12 Bytes (requiring two loads for every one store), data movement saturates the memory. This is confirmed by the 100% utilization of the VLOAD and VSTORE units, which starves the VALU, keeping its utilization low (e.g., 30%–50%).



BlockWise Matrix Multiplication

```
def matmul_small(x: np.ndarray, y: np.ndarray) -> np.ndarray:
  m, k, n = x.shape[0], x.shape[1], y.shape[0]
  return np.matmul(x, y)

def block_matmul(
    x: np.ndarray,
    y: np.ndarray,
    *,
    bm: int = 256,
    bk: int = 256,
    bn: int = 256,
) -> np.ndarray:
  m, k = x.shape
  _, n = y.shape

  z = np.zeros((m, n), dtype=x.dtype)
  for m_i in range(m // bm):
    for n_i in range(n // bn):
      for k_i in range(k // bk):
        m_slice = slice(m_i * bm, (m_i + 1) * bm)
        k_slice = slice(k_i * bk, (k_i + 1) * bk)
        n_slice = slice(n_i * bn, (n_i + 1) * bn)
        x_block = x[m_slice, k_slice]
        y_block = y[k_slice, n_slice]
        z[m_slice, n_slice] += matmul_small(x_block, y_block)
  return z
```



Decomposition: The core idea is to break down a large matmul(X,Y) of size (m, k)×(k, n) into many smaller, manageable block multiplications.

Three Nested Loops: The process is governed by three nested loops corresponding to the block dimensions of the output and the summation axis:

- **Outer loop (M)** iterates over output row blocks (m // bm)
- **Middle loop (N)** iterates over output column blocks (n // bn)
- **Inner loop (K)** iterates over the summation axis blocks (k // bk), where the result is accumulated

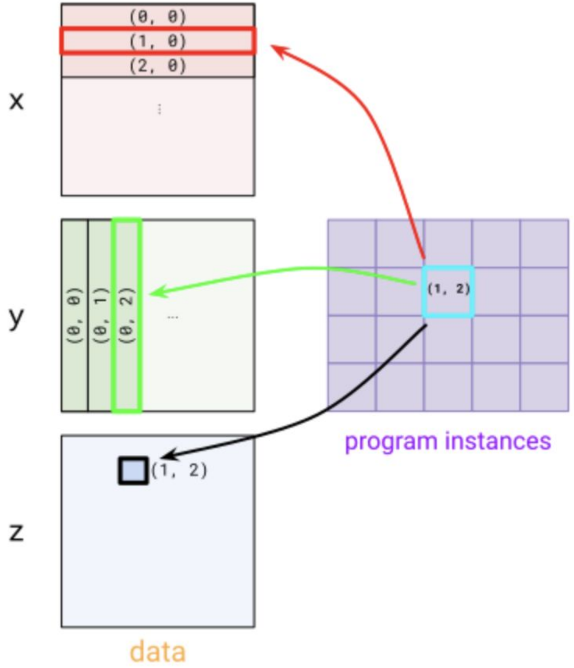
Data reuse and locality: Keep the current Z block in the fast VMEM for its entire accumulation phase

Pallas Fundamentals

Grid: This defines a parallel execution space.
grid=(2, 2) means the kernel will be invoked 4 times, with each instance processing a different logical slice of the data. TPUs process this grid sequentially in lexicographical order.

BlockSpec: This is the core mechanism for memory management. It partitions large global tensors in HBM into smaller blocks and maps them to specific grid coordinates. Pallas uses BlockSpec to implicitly manage DMA transfers between HBM and VMEM, enabling pipelining .

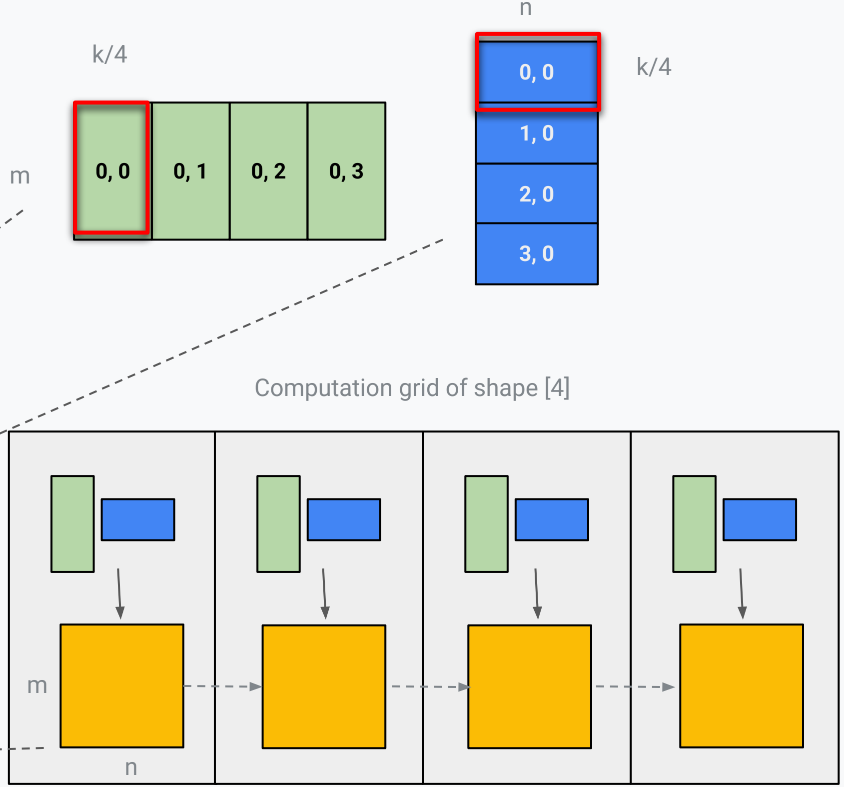
Pipelining: Pallas overlaps data loading/storing from HBM to VMEM with the actual computation performed by the MXU. This is crucial for keeping the MXU busy and avoiding memory-bound performance.



1D Grid Pallas Matmul Example

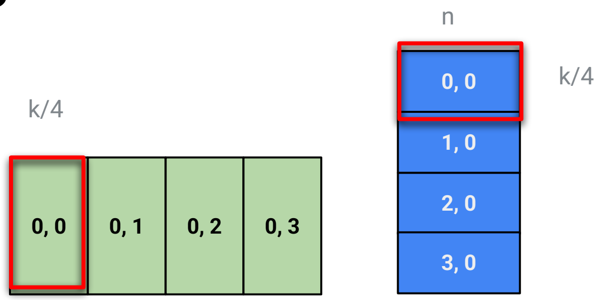
The 1D grid primarily divides the inner accumulation dimension (k)

```python
def pallas_matmul(x, y): x: f32[m, n], y: f32[n, k], out: f32[m, k]

  def kernel(x_ref, y_ref, out_ref):
    # x_ref: f32[m, n // num_blocks], y_ref: f32[n // num_blocks, k]
    # out_ref: f32[m, k]
    x_block = x_ref[...]
    y_block = y_ref[...]
    # accumulate the result
    out_ref[...] = out_ref[...] + jnp.dot(x_block, y_block)

  m, k, n = x.shape[0], x.shape[1], y.shape[1]
  num_blocks = 4

  return pl.pallas_call(
      kernel,
      grid=(num_blocks,),  # `num_blocks` kernel calls in total
      in_specs=[
          pl.BlockSpec(
            index_map=lambda i: (0, i),
            block_shape=(m, k // num_blocks)), # x
          pl.BlockSpec(
            index_map=lambda i: (i, 0),
            block_shape=(k // num_blocks, n)), # y
      ],
      out_shape=jax.ShapeDtypeStruct(shape=(m, n), dtype=x.dtype),
      out_specs=pl.BlockSpec(
          index_map=lambda i: (0, 0),  # out
          block_shape=(m, n),
      ),
  )(x, y)
```

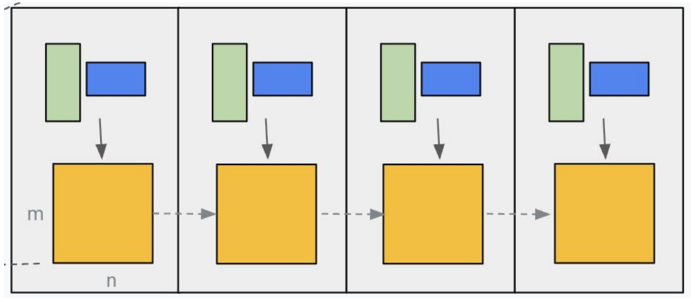


1D Grid Matmul: Primary Challenges

The VMEM Constraint: This technique requires the **entire output matrix Z(M, N)** to reside in VMEM simultaneously. For large-scale matrices, the output footprint consumes the available VMEM, leaving insufficient space for input buffers and triggering **Out-of-Memory (OOM)** errors.



Zero Output Parallelism: Because the M and N dimensions are unpartitioned, the execution grid is forced into a serial accumulation. This prevents the workload from being distributed across multiple independent **TensorCores**, capping the compute throughput to a fraction of the hardware's potential.



2D Grid Pallas Matmul

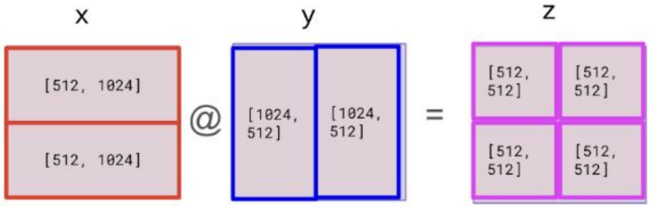
Abstracted Kernel: Expresses core matrix logic in a single Python line, delegating hardware-specific scheduling to the Pallas runtime.

pallas_call Orchestrator: Serves as the primary scheduler that binds the kernel to the TPU and configures the pipelined execution environment.

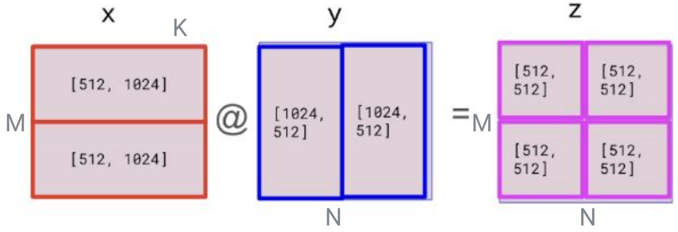
grid=(2, 2) Parallelism: Launches four simultaneous kernel instances, scaling the computation across multiple independent TPU compute units.

BlockSpec Data Mapping: Uses **lambda index maps** to route specific HBM data slices to VMEM based on grid coordinates, enabling efficient tiling and data reuse.

```
def matmul_kernel(x_ref, y_ref, z_ref):
  z_ref[...] = x_ref[...] @ y_ref[...]

def matmul(x: jax.Array, y: jax.Array):
  return pl.pallas_call(
    matmul_kernel,
    out_shape=jax.ShapeDtypeStruct((x.shape[0], y.shape[1]), x.dtype),
    grid=(2, 2),
    in_specs=[
      pl.BlockSpec((x.shape[0] // 2, x.shape[1]), lambda i, j: (i, 0)),
      pl.BlockSpec((y.shape[0], y.shape[1] // 2), lambda i, j: (0, j))
    ],
    out_specs=pl.BlockSpec(
      (x.shape[0] // 2, y.shape[1] // 2), lambda i, j: (i, j),
    )
  )(x, y)

k1, k2 = jax.random.split(jax.random.key(0))
x = jax.random.normal(k1, (1024, 1024))
y = jax.random.normal(k2, (1024, 1024))
z = matmul(x, y)
```



2D-Grid MatMul Challenges



The 1D Limit: The 1D grid successfully handled the summation by tiling only the reduction axis (K), but it failed to parallelize the output plane (MxN).

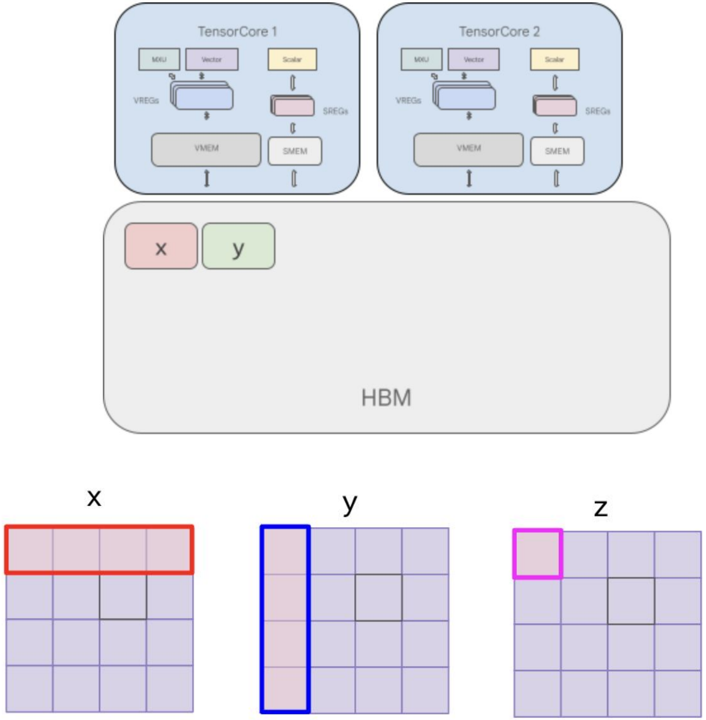
The 2D Challenge (VMEM Overflow): While a 2D grid grid=(2,2) parallelizes the output, it assumes the entire shared dimension (K) can fit in VMEM. For large-scale models, the blocks $X_{i,full}$ and $Y_{full,j}$ are too massive for on-chip memory, leading to immediate **VMEM Overflow**.

The 3D Solution: To handle large K dimensions without crashing, a **3D Grid** is required. By adding the accumulation axis as the third dimension, Pallas can tile M, N, *and* K simultaneously, ensuring every data chunk fits comfortably within VMEM limits while maintaining full output parallelism.

3D Grid Matrix Multiplication Example

```
import jax
import jax.numpy as jnp
from jax.experimental import pallas as pl
from jax.experimental.pallas import tpu as pltpu

def matmul_kernel(x_ref, y_ref, z_ref):
  @pl.when(pl.program_id(2) == 0)
  def _():
    z_ref[...] = jnp.zeros_like(z_ref)
  z_ref[...] += x_ref[...] @ y_ref[...]

def matmul(x: jax.Array, y: jax.Array, bm: int, bk: int, bn: int) -> jax.Array:
  m, k = x.shape
  _, n = y.shape
  grid = (m // bm, n // bn, k // bk)
  return pl.pallas_call(
      matmul_kernel,
      out_shape=jax.ShapeDtypeStruct((m, n), x.dtype),
      grid = grid,
      in_specs=[
          pl.BlockSpec((bm, bk), lambda i, j, k: (i, k)),
          pl.BlockSpec((bk, bn), lambda i, j, k: (k, j))
      ],
      out_specs=pl.BlockSpec((bm, bn), lambda i, j, k: (i, j)),
      compiler_params=pltpu.TPUCompilerParams(
          dimension_semantics=("parallel", "parallel", "arbitrary")),
      debug=False,
  )(x, y)

m, k, n = 4096, 4096, 4096
x = jnp.ones((m, k))
y = jnp.ones((k, n))
z = matmul(x, y, 1024, 1024, 1024)
```

MegaCore



grid = [4, 4, 4] (program instances)

3D Grid Matmul: Full Parallelism & Accumulation

The **3D Grid (m,n,k)** is the standard, scalable implementation for block matrix multiplication, tiling all three dimensions of the problem: the output height (M), the output width (N), and the summation axis K

Pallas Component	Role in 3D Tiling	Key Mechanism
Grid	grid=(m//bm, n//bn, k//bk)	Using **grid=(m//bm, n//bn, k//bk)** defines a 3D iteration space. The first two axes (M, N) parallelize the output blocks, while the third axis (K) tiles the reduction dimension into smaller, VMEM-safe chunks.
Output (Z)	Accumulator Initialization	The output block Z_{i,j} is initialized to zero in VMEM only at the start of the K-loop **(pl.program_id(2) == 0)**. This ensures partial sums are cleanly accumulated within the core's local memory.
Output (Z)	Final Write-back	Pallas optimizes HBM bandwidth by only writing the final accumulated Z_{i,j} block back to main memory once the entire K-axis traversal is complete, avoiding redundant intermediate stores.
Scheduling	Parallelism vs. Serialism	While the M and N axes are distributed across physical cores for **parallel execution**, the K axis is processed **serially** on each core. This maximizes the temporal locality of the X and Y tiles within that core's VMEM.

Matrix Multiplication Performance: FLOPs vs. Bandwidth

Key Metrics for Matmul (m,k)@(k,n)

Metric	Calculation	Growth Rate	
Floating Point Operations (FLOPs)	The total number of calculations required.	**FLOPs≈2·m·k·n**	Cubic **O(N^3)**
Minimum Memory Bandwidth Usage	The minimum size of inputs (X, Y) plus output (Z) that must be transferred between HBM and VMEM	**BW Usage≈(m·k+k·n+m·n)·4 bytes** (for float32)	Quadratic **O(N^2)**

The differing growth rates mean the ratio of FLOPs to BW Usage increases as the matrix size grows. This ratio, called **Arithmetic Intensity** (FLOPs/Byte), determines whether the kernel is limited by the processor or the memory.

The Arithmetic Intensity Ratio

Arithmetic Intensity (FLOPs/Byte) **> Chip Capacity : Compute-Bound** (Ideal): The processor (FLOP/s) is the bottleneck. Hardware is fully utilized, waiting for computation to finish

Arithmetic Intensity (FLOPs/Byte) **< Chip Capacity : Memory-Bound** : Memory bandwidth is the bottleneck. Compute units are idling while waiting for data transfers (HBM↔VMEM)

Ironwood

- Peak FLOP Rate per TensorCore = **1028.75 TFLOP/s**
- Peak HBM Bandwidth per TensorCore: 3433 GiB/s = 3433 GiB/s × 1.07374 ≈ **3685.5 GB/s**
- Ironwood **Arithmetic Intensity** = FLOP/s / BW = **1028.75/3.685** ≈ **279.16** FLOPs/Byte

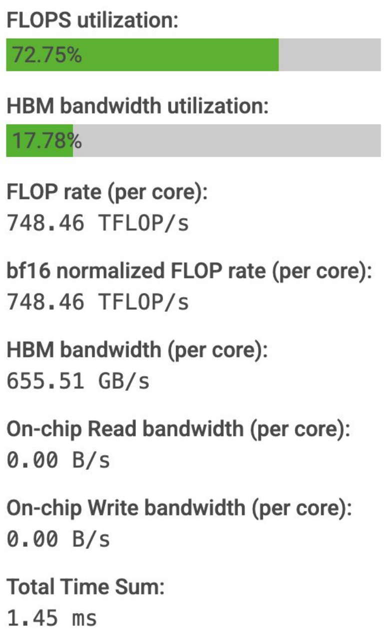
Square Matrix

- Matmul Intensity (M = K = N) : (2.M.M.M)/(M.M + M.M+ M.M) * 4 = **M/6** FLOPs/Byte
- Minimum Compute-Bound Size : M/6 > 279
    - M > (279 * 6) > **1674**

GiB=2^30 GB=10^9

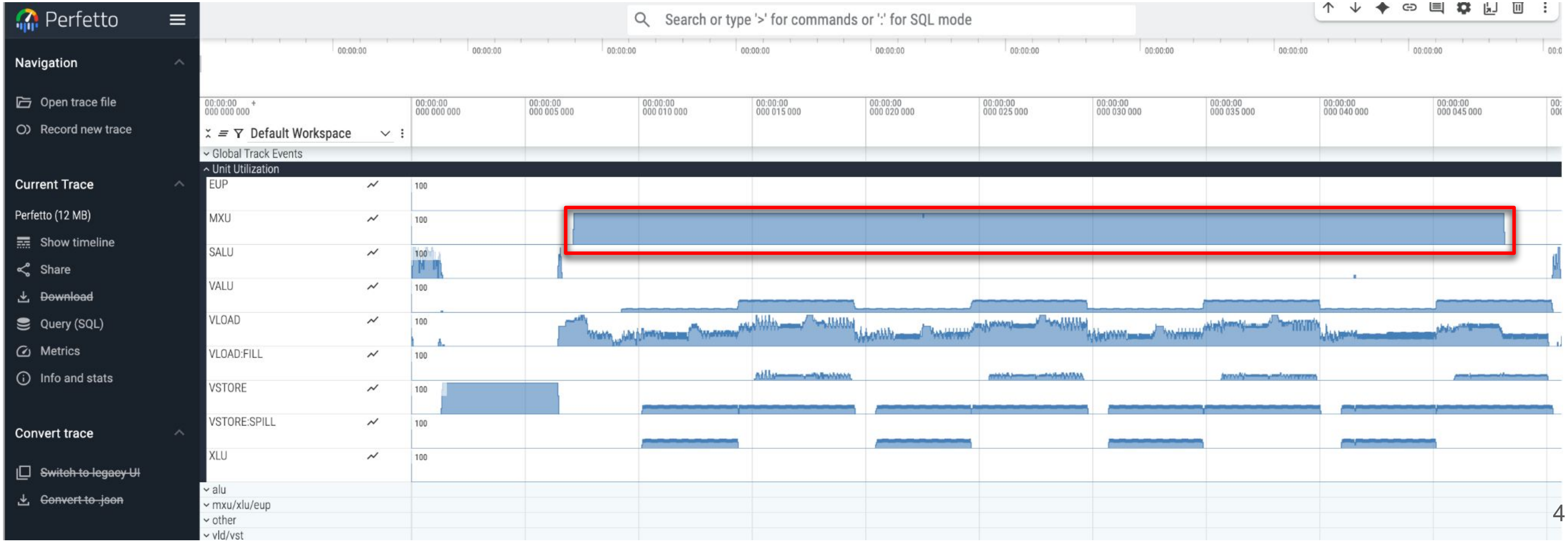
Performance Analysis: (4096,7168)×(7168,18432)

- Lets analyse matmul **(4096,7168)×(7168,18432)**
  - No pallas kernel, just jnp.dot(x, y) ⟶ dot_general
  - FLOPs = 2 x 4096 x 7168 x 18432 = 1.087 TFLOPs
- Ironwood
  - Peak FLOP Rate per TensorCore = **1028.75 TFLOP/s**
  - Peak HBM Bandwidth per TensorCore: 3433 GiB/s ≈ **3685.5 GB/s**
- (4096,7168)×(7168,18432) ⟶ 1.45 ms
  - **Achieved FLOP/s** = 1.087×10^{12} FLOPs / 1.45×10^{-3} = 748.46 TFLOPS/s
  - **Flops Utilization** = 748.46/1028.75 = **72.75%**
- Achieved HBM Bandwidth
  - HBM bandwidth per core ≈ 655.51 GB/s
  - **HBM bandwidth utilization** = 655.51/3685.5 = **17.78%**



Visualizing Pallas matmul kernel (4096, 7168) @ (7168, 18432)

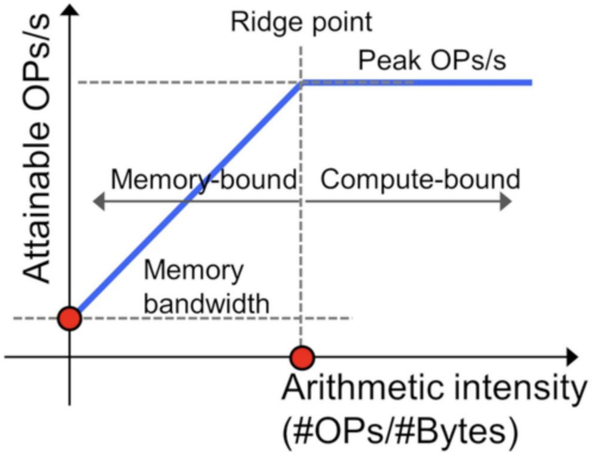
The visualization confirms the highly optimized 1024 tile Pallas kernel achieved a **Compute-Bound** state, using VLOAD/VSTORE overlap to keep the MXU continuously active and effectively hide the memory latency from the 504 kernel executions.



Compute vs Memory Bound

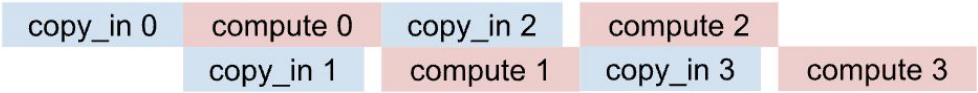
- TPU performance with LLMs is optimized by balancing compute-bound and memory-bound operations.
- **Compute-bound:**
  - The MXU is the bottleneck, fully used for matrix multiplications (attention, feedforward) in LLMs.
  - Example: LLMs with very large layers.
- **Memory-bound:**
  - HBM to VMEM data transfer is the bottleneck.
  - The MXU waits to load large tensors (Q, K, V, weights).
  - Example: Attention with long sequences.
- High arithmetic intensity (FLOPs per byte) is preferred for LLM efficiency on TPUs.

Roofline Model

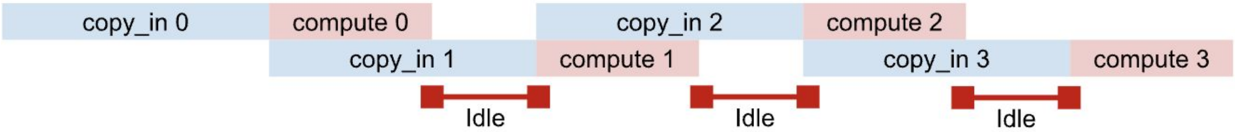


Compute Bound vs Memory Bound

Compute Bound: The bottleneck is the processing speed (FLOP/s) of TPU. The kernel is limited by how fast it can perform calculations.

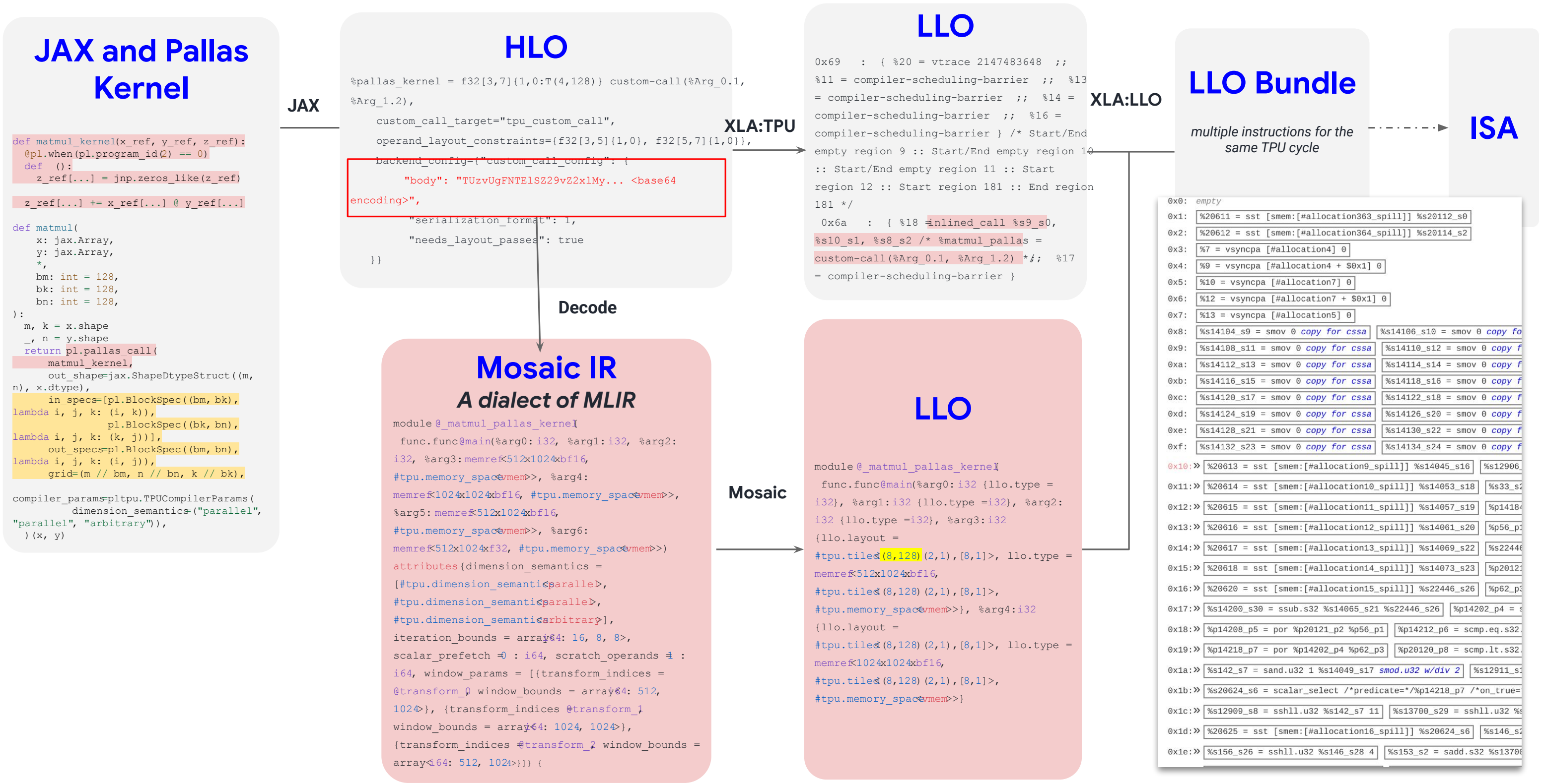


Memory Bound: The bottleneck is memory access (latency LL or bandwidth BB). The processor spends time idle, waiting for data to be transferred.



Ideal scenario: The goal is achieved when the program is compute-bound, meaning the hardware is fully utilized and the runtime is dominated by total required FLOPs divided by FLOP/s.

Pallas Compilation Path: An Overview



Tuning Pallas Kernels: The Critical Impact of Tile Size

Performance Comparison: Small vs. Optimized Tiles: Using the matrix size **4096×7168×18432** (Total work: **1.087 TFLOPs**), we see a dramatic difference in performance based on tile selection:

- IronwoodFish : Peak FLOP Rate per TensorCore = **1028.75 TFLOP/s**

Run	Tile Size	Matmul Time	Achieved TFLOP/s	FLOPs Utilization	Pallas Gird	Kernel invocations
Run 1 (Sub-Optimal)	bm=512, bk=512, bn=512	4.63 ms	233.6 TFLOP/s	22.72%	(8, 36, 14)	4032
Run 2 (Optimized)	bm=1024, bk=1024, bn=1024	1.68 ms	644.6 TFLOP/s	62.70%	(4, 18, 7)	504

- The change in tile size alone resulted in the kernel running **2.75 times faster**
- The primary goal of **Pallas tuning** is to minimize instruction overhead by making the execution grid as small as possible. This is achieved through the continuous process of finding the largest possible tile sizes **(bm,bn,bk)** that can fully saturate the TensorCore's VMEM capacity, ultimately pushing FLOPs utilization closer to the physical limits of the processor.

(4096, 7168) @ (7168, 18432)

Fusion: The Core Concept

Fusion: Fusion combines multiple sequential operations (e.g., **Matmul → ReLU → Add**) into a single unified kernel launch. Instead of treating each operation as a separate task, the accelerator processes the entire chain in one pass.

Breaking the Memory Wall: In a non-fused sequence, every op must read inputs from **HBM** and write intermediate results back to it. For element-wise operations like ReLU, the time spent moving data across the "memory wall" often exceeds the time spent on actual computation.

The Cost of "Round-trips": Without fusion, a Matmul followed by an Activation requires a full round-trip to memory. Even if the Matmul is extremely fast, the subsequent read/write of the large Z matrix forces the hardware to idle, wasting significant energy and throughput.

Fused Activation Function in the Kernel

VMEM as Local Accumulator: The z_ref buffer maintains data residency within **VMEM**. By keeping the intermediate matrix product local, the system avoids the "memory wall" associated with constant HBM round-trips.

In-Place Execution: Once the accumulation is finished, the activation function runs **in-place** directly on the final block inside VMEM. This ensures that the activation result overwrites the temporary data without requiring additional memory allocation.

Single Write-Back: Efficiency is maximized because only one final write-back to **HBM** occurs. By executing the entire chain (Matmul + Activation) before moving data to main memory, you significantly reduce HBM bandwidth consumption and kernel latency.

```
def matmul_kernel(x_ref, y_ref, z_ref, nsteps, activation):
  # 1. Initialization
  @pl.when(pl.program_id(2) == 0)
  def _():
    z_ref[...] = jnp.zeros_like(z_ref)

  # 2. Accumulation
  z_ref[...] += x_ref[...] @ y_ref[...]

  # 3. Fused Activation
  @pl.when(pl.program_id(2) == nsteps - 1)
  def _():
    z_ref[...] = activation(z_ref[...]).astype(z_ref.dtype)
```

Manual Pipelining: The Overlap Advantage

Pallas's **pallas_call** automatically overlaps data transfer with computation, but manual Direct Memory Access (DMA) is **sometimes required** to achieve absolute peak hardware utilization

- **Explicit Timing Control:** Manual DMA gives the programmer control over ***when*** the next load starts. We can use **pl.make_async_copy(...)** to trigger the next block load immediately after heavy compute begins, achieving the maximal essential overlap.
- **Deeper Pipelining:** Allows for complex circular buffer schemes (e.g., three or four slots) and non-standard tiling, which exceed the capabilities of the automatic double-buffering from pallas_call
- **Guaranteed Locality:** Provides full control over VMEM buffers to prevent premature eviction, ensuring blocks remain resident to keep the kernel running at peak FLOPs utilization.

Customizing the pallas_call Signature

To transition from automatic to manual DMA, the structure of the pl.pallas_call must be modified to pass high-level **HBM references and scratch buffers** into the kernel.

```
out = pl.pallas_call(
    matmul_kernel_dma,
    grid_spec=pltpu.PrefetchScalarGridSpec(
      num_scalar_prefetch=1,
      grid=(m // bm, n // bn, k // bk),  # `num_blocks` kernel calls in total
      in_specs=[
          pl.BlockSpec(memory_space=pltpu.TPUMemorySpace.ANY), # x
          pl.BlockSpec(memory_space=pltpu.TPUMemorySpace.ANY), # y
      ],
      out_specs=pl.BlockSpec((bm, bn), lambda i, j, k, _: (i, j)),
      scratch_shapes=[
          pltpu.VMEM((2, bm, bk), x.dtype), # VMEM for x, 2 for double buffering
          pltpu.VMEM((2, bk, bn), y.dtype), # VMEM for y, 2 for double buffering
          pltpu.SemaphoreType.DMA,
      ],
    ),
    out_shape=jax.ShapeDtypeStruct(shape=(m, k), dtype=x.dtype),
)(jnp.zeros((1,), dtype=jnp.int32), x, y)
return out
```

This allows a non-array scalar, the buffer_index, to be passed into the kernel via SMEM(Scalar Memory), controlling the alternating buffers

The kernel receives HBM pointers (x_hbm_ref,y_hbm_ref) and must manually slice the data using Pallas's pl.ds(start, size) function

Double Buffering VMEM: The key is allocating two spaces for both X and Y blocks within the scratch_shapes argument of pl.pallas_call

Sequential DMA: The Cost of No Pipelining

No Overlap: The core issue is the immediate synchronization: copy.start() is followed directly by copy.wait()

Forced Idle: The processor (MXU) is forced to **idle** for the entire duration of the slow HBM data transfer.

Memory-Bound: This wasted time confirms the operation is severely **Memory-Bound**, as the MXU is always waiting for the memory pipe.

Low Utilization: The result is extremely low efficiency (23.4%), wasting 76% of the core's compute potential.

- (4096, 7168) @ (7168, 18432) with (1024, 1024, 1024) tiles

```
def matmul_kernel_dma(x_hbm_ref, y_hbm_ref, out_ref, x_vmem_ref, y_vmem_ref, sem):

    i = pl.program_id(0)
    j = pl.program_id(1)
    k = pl.program_id(2)

    copy_x = pltpu.make_async_copy(
        x_hbm_ref.at[pl.ds(i*bm, bm), pl.ds(k*bk, bk)], x_vmem_ref, sem)
    copy_y = pltpu.make_async_copy(
        y_hbm_ref.at[pl.ds(k*bk, bk), pl.ds(j*bn, bn)], y_vmem_ref, sem)

    copy_x.start()
    copy_y.start()
    copy_x.wait()
    copy_y.wait()

    @pl.when(pl.program_id(2) == 0)
    def _():
      out_ref[...] = jnp.zeros_like(out_ref)

    # Now the content on the VMEM is ready
    x_block = x_vmem_ref[...]
    y_block = y_vmem_ref[...]
    out_block = jnp.matmul(x_block, y_block)
    out_ref[...] = out_ref[...] + out_block  # accumulate the result
```

Overlapping I/O and Compute

1. Trigger Next Copy: copy_next.start() is executed early in the loop to initiate the load for the next block into the other buffer, maximizing the overlap window.

2. Wait for Current Block: copy_x.wait() and copy_y.wait() forces the MXU to pause only if the DMA has not yet finished loading the current block.

3. Compute Phase: jnp.matmul() executes on the ready buffer's data, which is now guaranteed to be local, while the DMA engine works in the background.

4. Final Write-Back: The output out_ref (the accumulator) is only written from VMEM to HBM once at the very end of the 3D loop, maintaining high FLOPs intensity.

```python
def matmul_kernel_dma(buffer_index_ref, x_hbm_ref, y_hbm_ref, out_ref, x_vmem_ref, y_vmem_ref, sem):

    i = pl.program_id(0)
    j = pl.program_id(1)
    k = pl.program_id(2)

    num_programs_i = pl.num_programs(axis=0)
    num_programs_j = pl.num_programs(axis=1)
    num_programs_k = pl.num_programs(axis=2)

    def get_next_program_id(i, j, k, ni, nj, nk):
      """Concise next program ID using jax.lax.cond."""
      next_k = (k + 1) % nk
      next_j = jax.lax.cond(next_k == 0, lambda _: (j + 1) % nj, lambda _: j, None)
      next_i = jax.lax.cond((next_k == 0) & (next_j == 0), lambda _: (i + 1) % ni, lambda _: i, None)
      return next_i, next_j, next_k

    @pl.when(pl.program_id(2) == 0)
    def _():
      out_ref[...] = jnp.zeros_like(out_ref)

    # Copy a slice of array from HBM to VMEM
    buffer_index = buffer_index_ref[0]
    copy_x = pltpu.make_async_copy(
        x_hbm_ref.at[pl.ds(i*bm, bm), pl.ds(k*bk, bk)], x_vmem_ref.at[buffer_index], sem)
    copy_y = pltpu.make_async_copy(
        y_hbm_ref.at[pl.ds(k*bk, bk), pl.ds(j*bn, bn)], y_vmem_ref.at[buffer_index], sem)

    @pl.when((i == 0) & (j == 0) & (k == 0))
    def async_copy_first_block():
        copy_x.start()
        copy_y.start()

    @pl.when((i < num_programs_i - 1) | (j < num_programs_j - 1) | (k < num_programs_k - 1))
    def async_copy_next_block():
        next_buffer = jnp.where(buffer_index == 0, 1, 0) # swap buffer
        buffer_index_ref[0] = next_buffer

        # Get next program invocation
        next_i, next_j, next_k = get_next_program_id(
            i, j, k, num_programs_i, num_programs_j, num_programs_k)

        copy_next_x = pltpu.make_async_copy(
            x_hbm_ref.at[pl.ds(next_i*bm, bm), pl.ds(next_k*bk, bk)],
            x_vmem_ref.at[buffer_index], sem)
        copy_next_y = pltpu.make_async_copy(
            y_hbm_ref.at[pl.ds(next_k*bk, bk), pl.ds(next_j*bn, bn)],
            y_vmem_ref.at[buffer_index], sem)

        copy_next_x.start()
        copy_next_y.start()

    # Wait for the current block transfer
    copy_x.wait()
    copy_y.wait()

    # Now the content on the VMEM is ready
    x_block = x_vmem_ref.at[buffer_index][...]
    y_block = y_vmem_ref.at[buffer_index][...]
    out_block = jnp.matmul(x_block, y_block)

    out_ref[...] = out_ref[...] + out_block  # accumulate the result
```

Explicit Pipelining

Implicit Pipelining (pl.pallas_call)

- **Automated I/O Boundaries:** Manages HBM ↔ VMEM transfers automatically at kernel entry and exit, ensuring data is resident before compute starts.
- **Managed Synchronization:** Simplifies the "Load → Compute → Store" lifecycle by handling all DMA "wait" signals and buffer state transitions behind the scenes.

Explicit Pipelining (pltpu.emit_pipeline)

- **Granular Internal Control:** Moves orchestration **inside** the kernel body, allowing you to manually trigger asynchronous memory moves while computation is active.
- **Custom Overlap Strategies:** Enables advanced Software Pipelining (like double-buffering), letting you pre-fetch "Tile N+1" while the hardware is still processing "Tile N."

```
def matmul_pipeline(x_ref, y_ref, z_ref):
  @pl.when(pl.program_id(2) == 0)
  def _():
    z_ref[...] = jnp.zeros_like(z_ref)

  z_ref[...] += x_ref[...] @ y_ref[...]

@functools.partial(jax.jit, static_argnames=('bm', 'bk', 'bn'))
def matmul(x: jax.Array, y: jax.Array, *,
    bm: int = 1024, bk: int = 1024, bn: int = 1024,
):
  m, k = x.shape
  _, n = y.shape

  def matmul_kernel(x_hbm_ref, y_hbm_ref, z_hbm_ref):
    pltpu.emit_pipeline(
      matmul_pipeline,
      grid=(m // bm, n // bn, k // bk),
      in_specs=[pl.BlockSpec((bm, bk), lambda i, j, k: (i, k)),
                pl.BlockSpec((bk, bn), lambda i, j, k: (k, j))]
      out_specs=pl.BlockSpec((bm, bn), lambda i, j, k: (i, j)),
    )(x_hbm_ref, y_hbm_ref, z_hbm_ref)

  return pl.pallas_call(
    matmul_kernel,
    out_shape=jax.ShapeDtypeStruct((m, n), x.dtype),
    in_specs=[
      pl.BlockSpec(memory_space=pltpu.MemorySpace.ANY), # x
      pl.BlockSpec(memory_space=pltpu.MemorySpace.ANY), # y
    ],
    out_specs=pl.BlockSpec(memory_space=pltpu.MemorySpace.ANY),
  )(x, y)
```

Flash Attention - Pallas Kernel

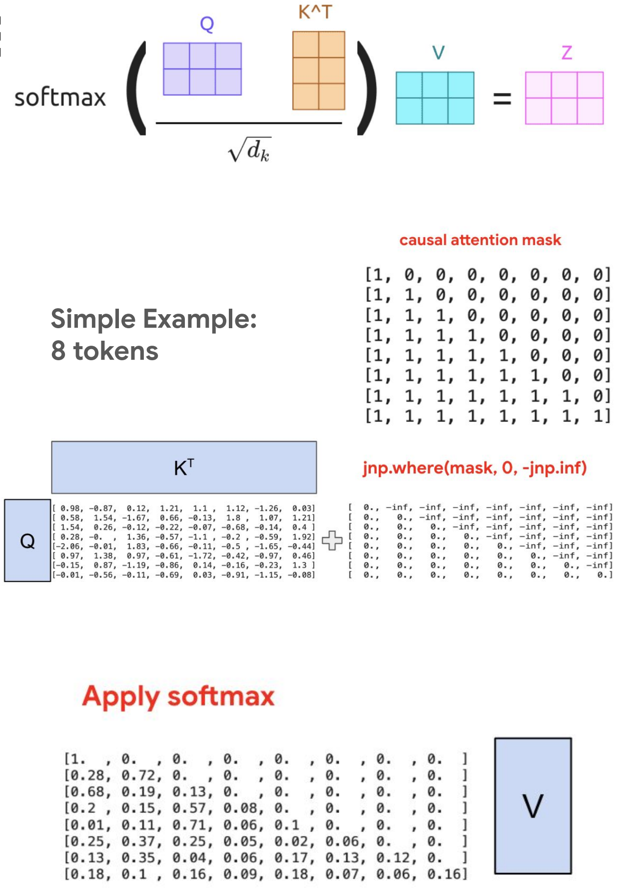
Tokamax

- Tokamax is a library of custom accelerator kernels, supporting both NVIDIA GPUs and Google TPUs.
- Tokamax provides state-of-the-art custom kernel implementations built on top of JAX and Pallas.
- Tokamax also provides tooling for users to build and autotune their own custom accelerator kernels.

- https://github.com/openxla/tokamax
- https://github.com/openxla/tokamax/tree/main/tokamax/_src/ops/experimental/tpu/splash_attention
- https://github.com/vllm-project/tpu-inference/tree/main/tpu_inference/kernels
- https://github.com/vllm-project/tpu-inference/tree/main/tpu_inference/kernels/ragged_paged_attention/v3

Recap: Simple Attention Mechanis

- **Core Mechanism:** Attention weighs input parts, enabling models to focus on relevant information by computing a weighted sum.
  - **O = softmax(QK^T / √d□) V**
    - Q = Query, K = Key, V = Value,
    - d□ = Key dimension, O = Output
- **Causal Masking:** Causal masks are applied to QK^T in decoders to prevent attending to future tokens, before the softmax function.
  - If **mask[i, j] = 0**, then **(QK^T)[i, j] = -∞**
  - Where i is the query position and j is the key pos.
- **Regularization:** Dropout (p) for regularization is applied to the softmax output before multiplying by V.
  - **O = dropout(softmax(QK^T / √d□), p) V**



Attention FLOPS

Model Params:

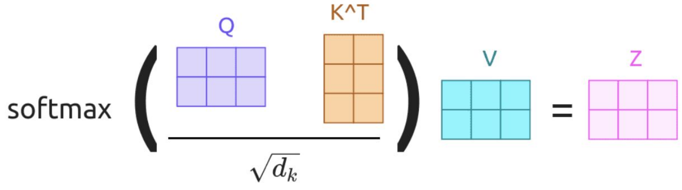
- Batch Size: 8, Seq Length: 4096, Heads: 128
- Query/Key head dim = 192, Value Dim = 128

Attention Flops:

- QK^T = 2×8×128×4096×192×4096 = 6.59 TFLOPs
- $(Q \times K^T) \times V$ = 2×8×128×4096×4096×128 = 4.39 TFLOPs
- Softmax Can be ignored (tiny FLOPS)
- Total = 6.59 + 4.39 = 10.98 TFLOPs

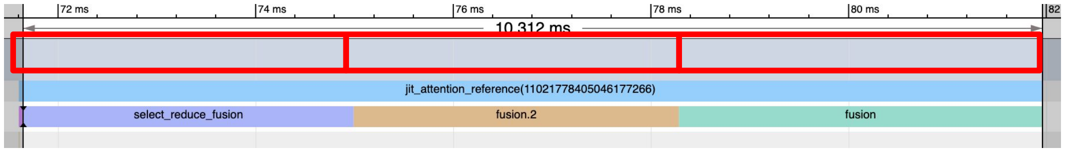
Ironwood : Peak FLOP Rate per TensorCore = **1028.75 TFLOP/s**

Custom Kernels - Motivation



The Bottleneck: Standard attention is stalled by HBM bottlenecks; moving the massive NxN matrix for Softmax and Dropout creates a "memory wall" that limits performance in long sequences.

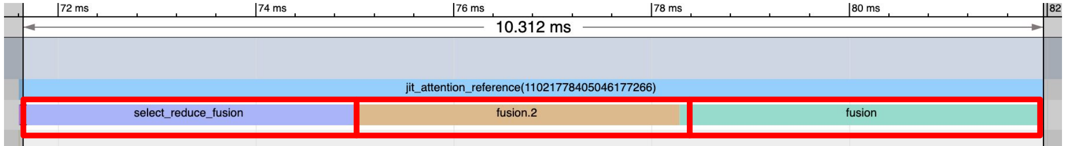
The Solution: Custom kernels use **Tiling and Fusion** to keep intermediate scores in **VMEM**. By processing the *Scale* → *Softmax* → *V-Product* chain in a single pass, the full NxN matrix is never written back to slow HBM.




```
num_heads = 128
seq_len = 4096
head_dim = 192
dtype = jnp.float32
DEFAULT_MASK_VALUE = -1e9

# 1. Create dummy input tensors [H, S, D]
key = jax.random.PRNGKey(42)
k1, k2, k3 = jax.random.split(key, 3)

q = jax.random.normal(k1, (num_heads, seq_len, head_dim), dtype=dtype)
k = jax.random.normal(k2, (num_heads, seq_len, head_dim), dtype=dtype)
v = jax.random.normal(k3, (num_heads, seq_len, head_dim), dtype=dtype)

# 2. Create a dummy attention mask [S, S]
mask = jnp.tril(jnp.ones((seq_len, seq_len), dtype=jnp.bool_))
is_mqa_mode = False

%%xprof
output = splash_attention_kernel.attention_reference(
  q=q,
  k=k,
  v=v,
  mask=mask,
  segment_ids=None,              # No segmentation for this example
  is_mqa=is_mqa_mode,
  mask_value=DEFAULT_MASK_VALUE,
  save_residuals=False,          # Simple attention output
  attn_logits_soft_cap=None,     # No soft capping on attention logits
).block_until_ready()
```

XLA Compiler Fusions For Attention

Fusion	Description	Inputs	Key Outputs & Shapes
Fusion 1: Logits & Max-Logits (%select_reduce_fusion)	Computes Q·K^T and M. Calculates the raw similarity scores (L), applies the attention mask, and determines the Max-Logits vector (M) across the sequence length.	Q,K [128,4096,192], Mask [4096,4096]	M(Max-Logits): [128,4096], L (Logits) [128, 4096, 4096] Total Time Avg: 3.34 ms **FLOPS Utilization: 24.10%** **HBM bandwidth utilization: 76.23%**
Fusion 2: Softmax Denominator (%fusion.2)	Computes Lsum. Stabilizes the masked logits (Lmasked−M), computes the exponential, and performs a sum-reduction to find the Softmax Denominator (Lsum).	L (Logits) [128,4096,4096], M (Max-Logits) [128,4096], Mask [4096,4096]	Lsum (Sum of Exponentials): [128,4096] Total Time Avg: 3.29ms **FLOPS Utilization: 0.19%** **HBM bandwidth utilization: 70.82%**
Fusion 3: Softmax & Output (ROOT %fusion)	Computes P and O. Re-calculates the stable numerator, divides by the denominator (Lsum) to get the Probability Matrix (P), and performs the final dot product P·V to get the output.	L [128,4096,4096], V [128,4096,192], M [128,4096], Lsum [128,4096]	P (Probability Matrix): [128,4096,4096] O (Attention Output): [128,4096,192] Total Time Avg: 3.68 ms **FLOPS Utilization: 21.97%** **HBM bandwidth utilization: 69.3%**



Custom Kernels: Why XLA's Fusions Aren't Enough for Peak Performance

- **Materializing the Attention Matrix:** XLA's multi-kernel approach must write the full, quadratically-sized Logit and Probability matrices ($O(S^2)$ > 8GB) to High Bandwidth Memory (HBM). This consumes vast amounts of memory bandwidth for temporary data.

- **Multiple HBM Round Trips** (Logits, Probabilities): Because the Softmax is split across Fusions 1, 2, and 3, the massive Logit and Probability matrices are written to HBM and then immediately read back for the next step. These multiple read/write cycles are slow and bandwidth-intensive

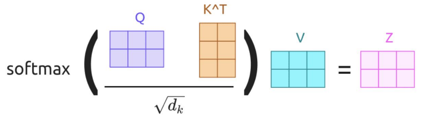
- **Storing M, L_sum to HBM**: Even the smaller normalization vectors (Max-Logits (M) and Sum of Exponentials (L_sum) are unnecessarily written to HBM after being computed and then read back by the final kernel, instead of being kept entirely on-chip.

Attention Challenges

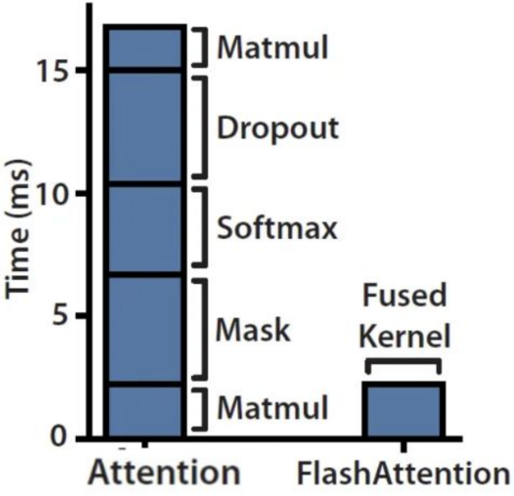
Quadratic Scaling: Standard self-attention requires storing the full NxN attention score matrix S. Because memory usage grows quadratically with sequence length, long sequences quickly exceed available hardware capacity.

The HBM Wall: High-bandwidth memory (HBM) is significantly slower than on-chip SRAM (**VMEM**). Repeatedly reading and writing the massive S matrix creates a "memory wall" that stalls the compute units.

The Flash Solution: Flash Attention eliminates this by never "materializing" the full NxN matrix in HBM. Instead, it uses **tiling** to compute the attention output in small blocks that fit entirely within fast on-chip memory.



https://arxiv.org/abs/2205.14135



Online Softmax

- **Softmax:**
  - The softmax function normalizes a vector of scores into a probability distribution.
  - A numerically stable implementation (safe softmax) is used to prevent overflow.
- **Online Softmax:**
  - Online softmax is used to process data in blocks
  - Each block does a local safe softmax calculation.
  - It tracks a shared maximum and adjusts results to ensure accuracy across all tiles.
  - Online softmax reduces memory usage and enables faster parallel processing.

softmax

naive

$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$

Algorithm 1 Naive softmax

$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
  $d_j \leftarrow d_{j-1} + e^{x_j}$
end for
for $i \leftarrow 1, V$ **do**
  $y_i \leftarrow \frac{e^{x_i}}{d_V}$
end for

3 For Loops

Algorithm 2 Safe softmax

$m_0 \leftarrow -\infty$
for $k \leftarrow 1, V$ **do**
  $m_k \leftarrow \max(m_{k-1}, x_k)$
end for
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
  $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
end for
for $i \leftarrow 1, V$ **do**
  $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

2 For Loops

Algorithm 3 Safe softmax with online normalizer

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
  $m_j \leftarrow \max(m_{j-1}, x_j)$
  $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
end for
for $i \leftarrow 1, V$ **do**
  $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

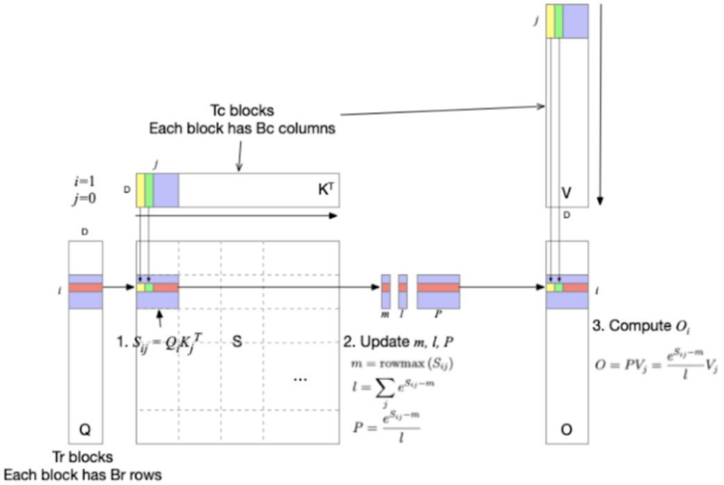
Minimizing HBM Access: The Core of Flash Attention

Bypasses the O(N^2) Memory Wall: Avoids storing the massive NxN attention matrix in HBM, eliminating the quadratic memory bottleneck that limits sequence length.

Fused Kernel Operations: Combines score calculation, softmax, and value aggregation into a single pass, drastically reducing slow data transfers between HBM and SRAM.

Tiled VMEM Processing: Breaks Q, K, and V into small blocks that fit entirely within high-speed local memory (VMEM), maximizing hardware utilization and throughput.

IO-Aware Optimization: Prioritizes reducing memory traffic over raw calculation counts, delivering significant speedups by focusing on the real hardware bottleneck: data movement.



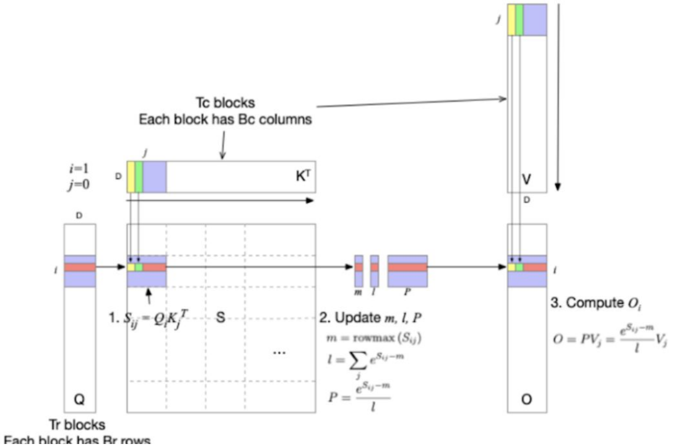
Picture Credit: https://insujang.github.io/2024-01-21/flash-attention/

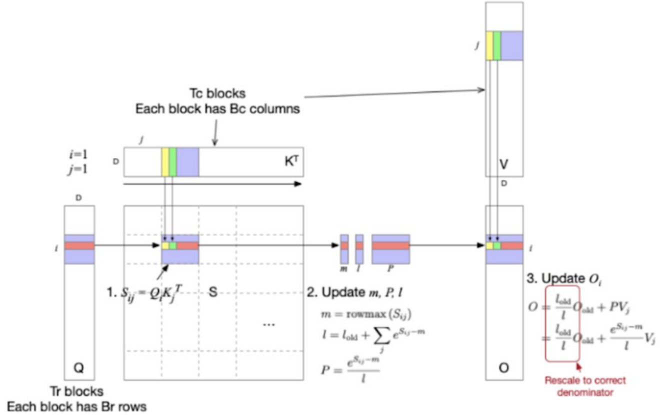
Blockwise Online Softmax

Iterative Tiling over K/V Blocks: For every Query (Q) block, the kernel iterates through all Key (K) and Value (V) blocks. This allows the TPU to compute attention scores in small, manageable strips that fit entirely within high-speed VMEM.

Dynamic Online Softmax: Each tile performs a local "safe softmax" while maintaining a running maximum (m) and sum-of-exponents (l). This incremental approach eliminates the need to materialize the full NxN attention matrix in HBM.

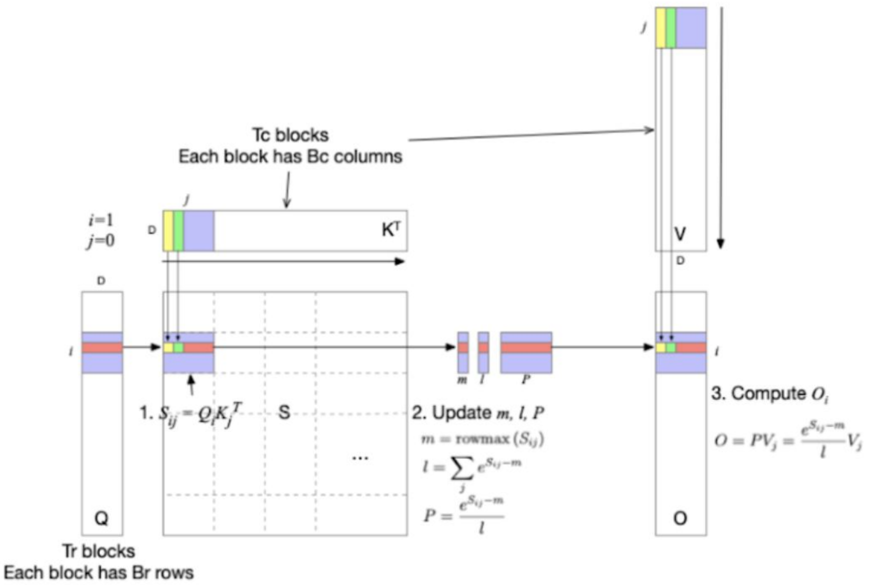
Running Output Correction: As new K/V blocks are processed, the previous partial output (O) is rescaled and adjusted to align with the updated running statistics. This ensures the final weighted sum is mathematically identical to standard softmax.

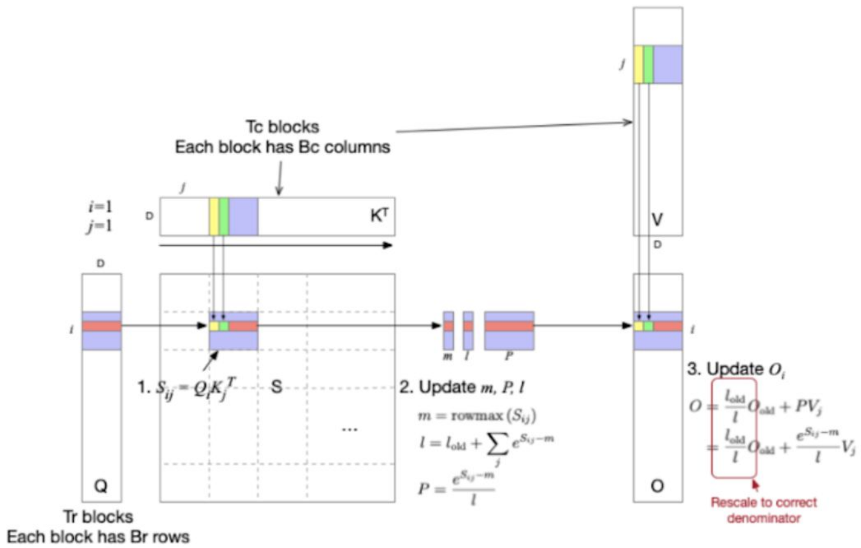




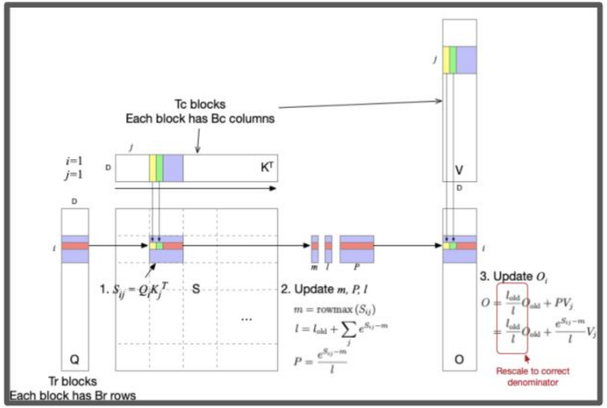
Flash Attention Summary

Flash Attention fuses attention operations and processes data in blocks, iteratively updating running statistics (L and M) to accurately compute the final output O without storing the full attention matrix in HBM.





Single Head Flash Attention Kernel




```python
@functools.partial(jax.jit, static_argnames=["br", "bc"])
def flash_attention(q, k, v, *, br: int, bc: int):
  seq_len, head_dim = q.shape
  return pl.pallas_call(
    flash_attention_kernel,
      out_shape=[
        jax.ShapeDtypeStruct((br, head_dim), q.dtype),  # l
        jax.ShapeDtypeStruct((br, head_dim), q.dtype),  # m
        jax.ShapeDtypeStruct((seq_len, head_dim), q.dtype),  # o
      ],
      in_specs=[
        pl.BlockSpec((br, head_dim), lambda i, j: (i, 0)),
        pl.BlockSpec((bc, head_dim), lambda i, j: (j, 0)),
        pl.BlockSpec((bc, head_dim), lambda i, j: (j, 0)),
      ],
      out_specs=[
        pl.BlockSpec((br, head_dim), lambda i, j: (0, 0)),  # l
        pl.BlockSpec((br, head_dim), lambda i, j: (0, 0)),  # m
        pl.BlockSpec((br, head_dim), lambda i, j: (i, 0)),  # o
      ],
      grid=(seq_len // br, seq_len // bc),
  )(q, k, v)[2]
```

```python
def flash_attention_kernel(q_ref, k_ref, v_ref, m_ref, l_ref, o_ref):

  @pl.when(pl.program_id(1) == 0)
  def _():
    neg_inf = -jnp.inf
    o_ref[...] = jnp.zeros_like(o_ref)
    m_ref[...] = jnp.full_like(m_ref, neg_inf)
    l_ref[...] = jnp.zeros_like(l_ref)

  q, k, v = q_ref[...], k_ref[...], v_ref[...]
  m_prev, l_prev = m_ref[...], l_ref[...]

  qk = jnp.dot(q, k.T)
  m_curr = qk.max(axis=-1)
  s_curr = jnp.exp(qk - m_curr[..., None])
  l_curr = jax.lax.broadcast_in_dim(s_curr.sum(axis=-1), l_prev.shape, (0,))
  o_curr = jnp.dot(s_curr, v) / l_curr

  m_curr = jax.lax.broadcast_in_dim(m_curr, m_prev.shape, (0,))
  m_next = jnp.maximum(m_prev, m_curr)
  alpha = jnp.exp(m_prev - m_next)
  beta = jnp.exp(m_curr - m_next)
  l_next = alpha * l_prev + beta * l_curr

  m_ref[...], l_ref[...] = m_next, l_next
  o_ref[...] = (l_prev * alpha * o_ref[...] + l_curr * beta * o_curr) / l_next
```

Flash Attention: Key Ops and Delayed Normalization

Dynamic Scaling (alpha): The core mechanism is the Alpha factor, calculated as exp(m_prev - m_next). This term tracks the change in the maximum logit (M) across blocks, ensuring numerical stability.

Re-scales Accumulation: This alpha factor is applied directly to the previous accumulated sums (l_prev, o_ref) before adding the current block's output. This accurately re-scales all past results relative to the new exponent reference point.

Additive Updates: Within the main loop, the partial results (l_curr, o_curr) are added to the running totals using only fast **multiplication and addition** operations.

Delayed Normalization: The final, slow division operation (normalization by L) is **intentionally skipped** in every block and executed only once at the very end of the loop, which drastically boosts accelerator throughput.

```
def flash_attention_kernel_so(q_ref, k_ref, v_ref, m_ref, l_ref, o_ref, grid_width):

  @pl.when(pl.program_id(2) == 0)
  def _():
    neg_inf = -jnp.inf
    o_ref[...] = jnp.zeros_like(o_ref)
    m_ref[...] = jnp.full_like(m_ref, neg_inf)
    l_ref[...] = jnp.zeros_like(l_ref)

  q, k, v = q_ref[...], k_ref[...], v_ref[...]
  m_prev, l_prev = m_ref[...], l_ref[...]

  # 1. matmul(qk), max, alpha scaling factor
  qk = jnp.dot(q, k.T)
  m_curr = qk.max(axis=-1)[:, None]
  m_next = jnp.maximum(m_prev, m_curr)
  alpha = jnp.exp(m_prev - m_next)

  # 2. Softmax update & track l_next
  s_curr = jnp.exp(qk - m_next)
  l_curr = s_curr.sum(axis=-1)[..., None]
  l_next = l_curr + alpha * l_prev

  # 3. SV matmul
  o_curr = jnp.dot(s_curr, v)
  o_ref[...] = alpha * o_ref[...] + o_curr

  m_ref[...], l_ref[...] = m_next, l_next

  # 4. Delay normalization until the end
  @pl.when(pl.program_id(2) == grid_width - 1)
  def end():
    o_ref[...] = (o_ref[...]/ l_next[...]).astype(o_ref.dtype)
```

$$\alpha = \text{jnp.exp}(\mathrm{M}_{\text{prev}} - \mathrm{M}_{\text{next}}) \equiv \frac{e^{\mathrm{M}_{\text{prev}}}}{e^{\mathrm{M}_{\text{next}}}}$$

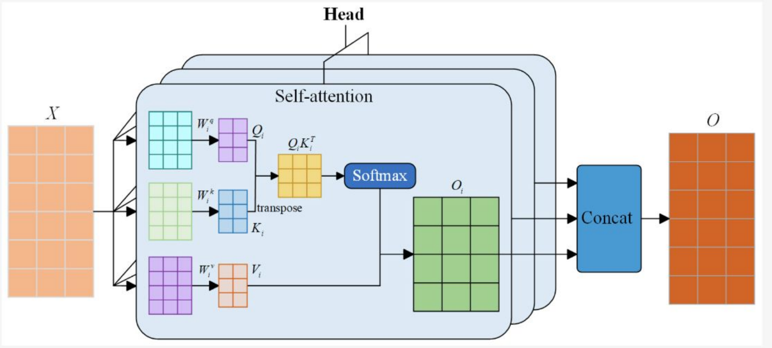
Multi-Head Attention

Subspace Parallelism: Projects Q, K, and V into multiple independent heads, allowing the model to attend to different representation subspaces (e.g., syntax vs. semantics) simultaneously.

Massive Scaling: Independent heads compute attention in parallel; **DeepSeek-V3** utilizes **128 heads** to capture intricate, high-dimensional dependencies across sequences.

Integration & Projection: Head outputs are concatenated and passed through a final linear projection, fusing diverse contextual insights into a single unified vector.

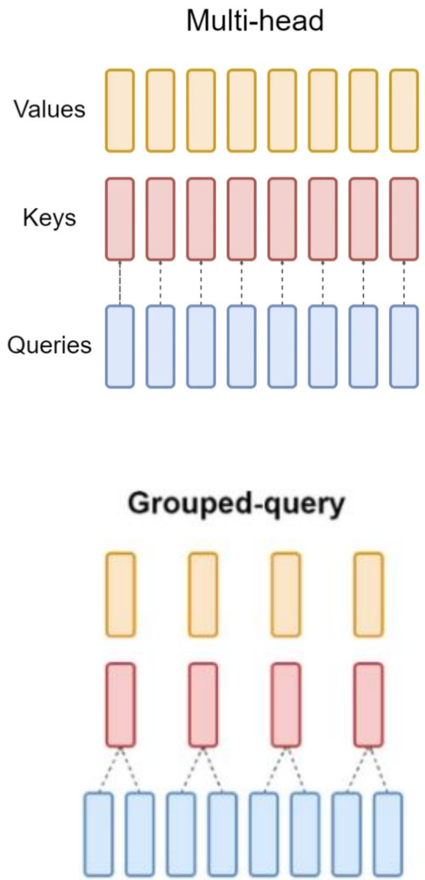
Rich Dependency Modeling: By diversifying the focus of each head, MHA provides significantly higher modeling capacity than a single-headed attention mechanism.



- **Query (q):** bf16[8,128,4096,192]
- **Key (k):** bf16[8,128,4096,192]
- **Value (v):** bf16[8,128,4096,128]
- **Output (o):** f32[8,2048,128]

Batch Size: 8, Seq Length: 4096
Attention Heads: 128
Query/Key head dim: 192, Value Dim: 128

Grouped Query Attention



Optimized Head Sharing: Multiple query heads share a single set of Key (K) and Value (V) projections, serving as a high-performance middle ground between Multi-Head and Multi-Query Attention.

Reduced Memory Footprint: Significantly lowers the KV cache storage requirements in HBM by only computing and storing one set of KV pairs per group.

Enhanced Throughput: Decreases memory bandwidth pressure during inference, enabling larger batch sizes and much longer sequence processing without hitting the "memory wall."

Parameter Efficiency: Maintains high modeling capacity while reducing redundant projections, allowing for more efficient hardware utilization on TPUs

MultiHead Attention (JAX Code)

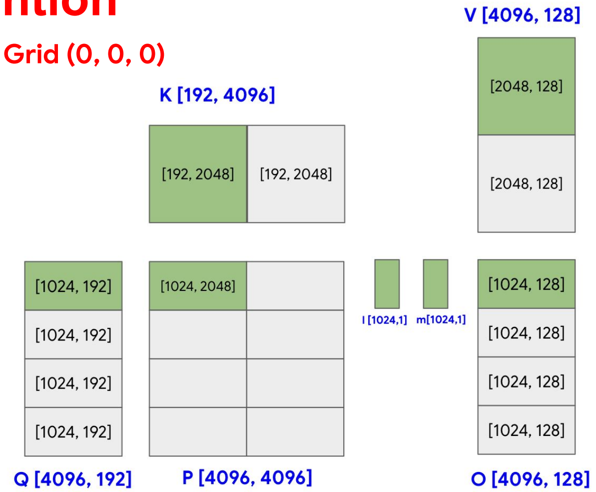
```python
def _attention_reference_impl(
    q: jax.Array,
    k: jax.Array,
    v: jax.Array,
    mask: jax.Array,
    segment_ids: SegmentIds | None,
    mask_value: float,
    save_residuals: bool,
    attn_logits_soft_cap: float | None,
) -> SplashCustomReturnType:
  logits = jnp.einsum("sd,td->st", q.astype(jnp.float32), k.astype(jnp.float32))

  if segment_ids is not None:
    mask = jnp.logical_and(
        mask, segment_ids.q[:, None] == segment_ids.kv[None, :]
    )

  if attn_logits_soft_cap is not None:
    logits = jnp.tanh(logits / attn_logits_soft_cap)
    logits = logits * attn_logits_soft_cap

  logits = jnp.where(mask, logits, mask_value)
  m = logits.max(axis=-1)
  s = jnp.exp(logits - m[..., None])
  l = s.sum(axis=-1)
  p = s / l[..., None]

  o = jnp.einsum("st,td->sd", p, v.astype(jnp.float32))

  if save_residuals:
    logsumexp = m + jnp.log(l)
    return o, (logsumexp, m)
  return o
```

```python
@partial(
    jax.jit,
    static_argnames=[
        "mask_value",
        "save_residuals",
        "attn_logits_soft_cap",
        "is_mqa",
    ],
)
def attention_reference(
    q: jax.Array,
    k: jax.Array,
    v: jax.Array,
    mask: jax.Array,
    segment_ids: SegmentIds | None = None,
    *,
    is_mqa: bool,
    mask_value: float = DEFAULT_MASK_VALUE,
    save_residuals: bool = False,
    attn_logits_soft_cap: float | None = None,
):
  """A JIT-compiled reference implementation of attention, handles MQA and MHA."""
  attn_impl = partial(
      _attention_reference_impl,
      mask_value=mask_value,
      save_residuals=save_residuals,
      attn_logits_soft_cap=attn_logits_soft_cap,
  )

  if is_mqa:
    func = jax.vmap(attn_impl, in_axes=(0, None, None, None, None))
  else:
    kv_heads, q_heads = k.shape[0], q.shape[0]
    assert q_heads % kv_heads == 0, (q_heads, kv_heads)
    if kv_heads < q_heads:
      q_heads_per_kv = q_heads // kv_heads
      k = jnp.repeat(k, repeats=q_heads_per_kv, axis=0)
      v = jnp.repeat(v, repeats=q_heads_per_kv, axis=0)

    func = jax.vmap(attn_impl, in_axes=(0, 0, 0, None, None))

  out = func(q, k, v, mask, segment_ids)
  return out
```

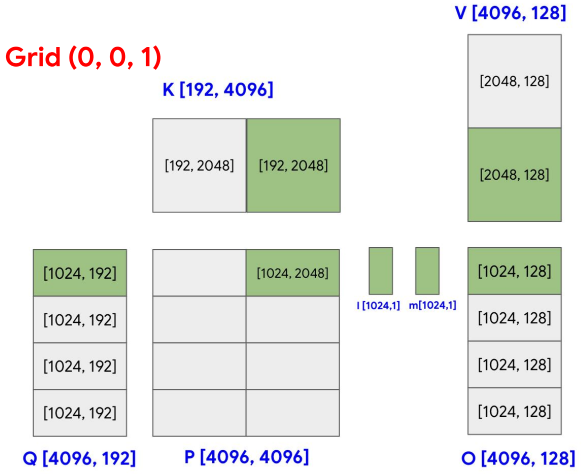
```python
def flash_attention_kernel_so(q_ref, k_ref, v_ref, m_ref, l_ref, o_ref, grid_width):

  @pl.when(pl.program_id(2) == 0)
  def _():
    neg_inf = -jnp.inf
    o_ref[...] = jnp.zeros_like(o_ref)
    m_ref[...] = jnp.full_like(m_ref, neg_inf)
    l_ref[...] = jnp.zeros_like(l_ref)

  q, k, v = q_ref[...], k_ref[...], v_ref[...]
  m_prev, l_prev = m_ref[...], l_ref[...]

  qk = jnp.dot(q, k.T)

  m_curr = qk.max(axis=-1)[:, None]
  m_next = jnp.maximum(m_prev, m_curr)
  s_curr = jnp.exp(qk - m_next)
  l_curr = s_curr.sum(axis=-1)[..., None]
  alpha = jnp.exp(m_prev - m_next)
  l_next = l_curr + alpha * l_prev
  m_ref[...], l_ref[...] = m_next, l_next

  o_curr = jnp.dot(s_curr, v)
  o_ref[...] = alpha * o_ref[...] + o_curr

  @pl.when(pl.program_id(2) == grid_width - 1)
  def end():
    o_ref[...] = (o_ref[...]/ l_next[...]).astype(o_ref.dtype)

@functools.partial(jax.jit, static_argnames=["br", "bc"])
def flash_attention_so(q, k, v, *, br: int, bc: int):
  num_heads, seq_len, q_head_dim = q.shape
  _, _, v_head_dim = v.shape
  return pl.pallas_call(
      functools.partial(flash_attention_kernel_so, grid_width=seq_len/bc),
      out_shape=[
        jax.ShapeDtypeStruct((br, 1), q.dtype),  # l
        jax.ShapeDtypeStruct((br, 1), q.dtype),  # m
        jax.ShapeDtypeStruct((num_heads, seq_len, v_head_dim), q.dtype),  # o
      ],
      in_specs=[
        pl.BlockSpec((None, br, q_head_dim), lambda h, i, j: (h, i, 0)),
        pl.BlockSpec((None, bc, q_head_dim), lambda h, i, j: (h, j, 0)),
        pl.BlockSpec((None, bc, v_head_dim), lambda h, i, j: (h, j, 0)),
      ],
      out_specs=[
        pl.BlockSpec((br, 1), lambda h, i, j: (0, 0)),  # l
        pl.BlockSpec((br, 1), lambda h, i, j: (0, 0)),  # m
        pl.BlockSpec((None, br, v_head_dim), lambda h, i, j: (h, i, 0)),  # o
      ],
      grid=(num_heads, seq_len // br, seq_len // bc),
  )(q, k, v)[2]

from jax import random
num_heads, seq_len, q_head_dim, v_head_dim = 128, 4096, 192, 128
k1, k2, k3 = random.split(random.PRNGKey(0), 3)
q = random.normal(k1, (num_heads, seq_len, q_head_dim))
k = random.normal(k2, (num_heads, seq_len, q_head_dim))
v = random.normal(k3, (num_heads, seq_len, v_head_dim))

out = flash_attention_so(q, k, v, br=1024, bc=2048).block_until_ready()
```

Multi-Head Attention

1. **Blockwise compute**, maximum stabilization.
2. **Update** global max (m) and sum (l).
3. **Merge** outputs using exponential scaling.



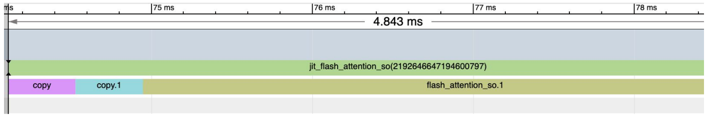
1. **Grid** schedules kernel iterations (heads, seq_len/br, seq_len/bc)
2. **Input specs** tile Q, K and V into blocks.
3. **Output specs** accumulate L, M, and write O

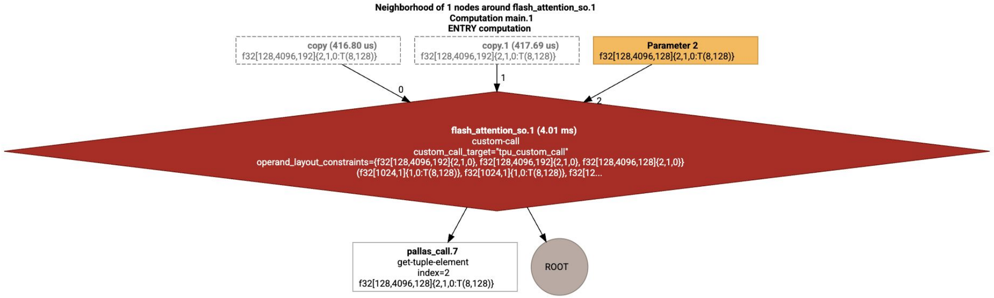


br = 1024, bc = 2048
grid = (128, 4096/1024, 4096/2048)
grid = (128, 4, 2)

Multi-Head Attention MFU

- **Attention FLOPS q/k [128, 4096, 192] and v[128, 4096, 128]**
    - FLOPs_Q K^T= 2 x 4096 x 192 x 4096 = 6.43 TFLOPs
    - FLOPs_SV = 2 x 4096 x 4096 x 128 = 4.30 TFLOPs
    - Total FLOPs = 128 x (FLOPs_Q K^T + FLOPs_SV) = 1373.4 TFLOPs
- **MFU Utilization**
    - Execution Time = 4 ms
    - Achieved FLOPs/s = Total FLOPs/Execution Time = 1373.4/4 = 343.35 TFLOPSs/s
    - Ironwood Peak FLOP Rate per TensorCore: 1028.75 TFLOP/s
    - **MFU Utilization = (342.5/1028.75) x 100% = 33.29%**





Pacchetto - Debugging Pallas Kernels

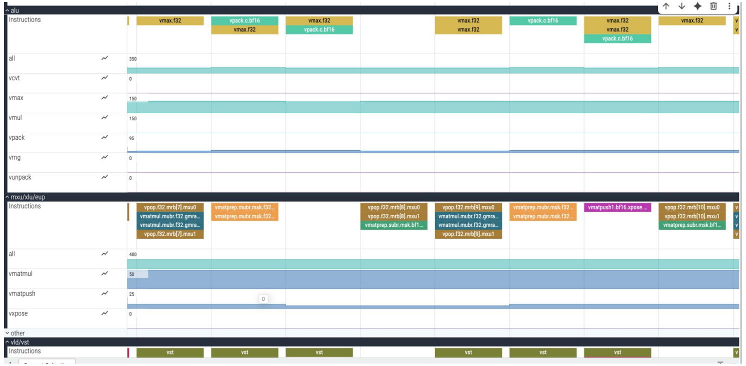
LLO Bundle Visualizer: A high-fidelity tool designed to analyze Low-Level Optimizer (LLO) bundles. It maps complex TPU instructions onto the Perfetto trace viewer, providing a granular timeline of kernel execution.

Hardware-Level Insights: Visualizes the precise utilization of the TensorCore, specifically tracking the activity of the MXU (Matrix Multiply Unit) and VPU. It allows developers to see exactly when compute units are idling or stalled.

Memory & Register Analysis: Tracks data flow across the TPU memory hierarchy and monitors register pressure. This is critical for identifying "spills" or inefficient memory movement that can degrade performance.

Kernel Bottleneck Detection: A vital tool for optimizing complex, hand-written kernels like **FlashAttention** or GMM. It helps engineers identify if a kernel is compute-bound (MXU limited) or memory-bound (HBM/VMEM bandwidth limited).

Pacchetto Trace Structure



This trace visualizes the **three main execution engines** on the accelerator chip and their maximum capacity (the scale on the left axis).

Trace Section	Functional Unit (FU)	Key Instructions	Maximum Capacity (Example Scale)	Notes
ALU (Top)	Vector/Scalar Unit (XLU)	vmax, vpack, vadd, vmov	≈350 OPC	Softmax & Activation: Performs all element-wise math, non-linear functions (ReLU, exp), and data formatting.
MXU/XLU/EUP (Middle)	Matrix Unit (MXU) & Control Pipes	vmatmul, vmatprep, vpop, vmatpush	≈400 OPC	Core Computation: Executes the high-throughput QK^T and SV matrix multiplications.
VLD/VST (Bottom)	Load/Store Unit (LSU)	vld, vst	-	Memory I/O: Transfers data between on-chip memory (VMEM) and the register file.

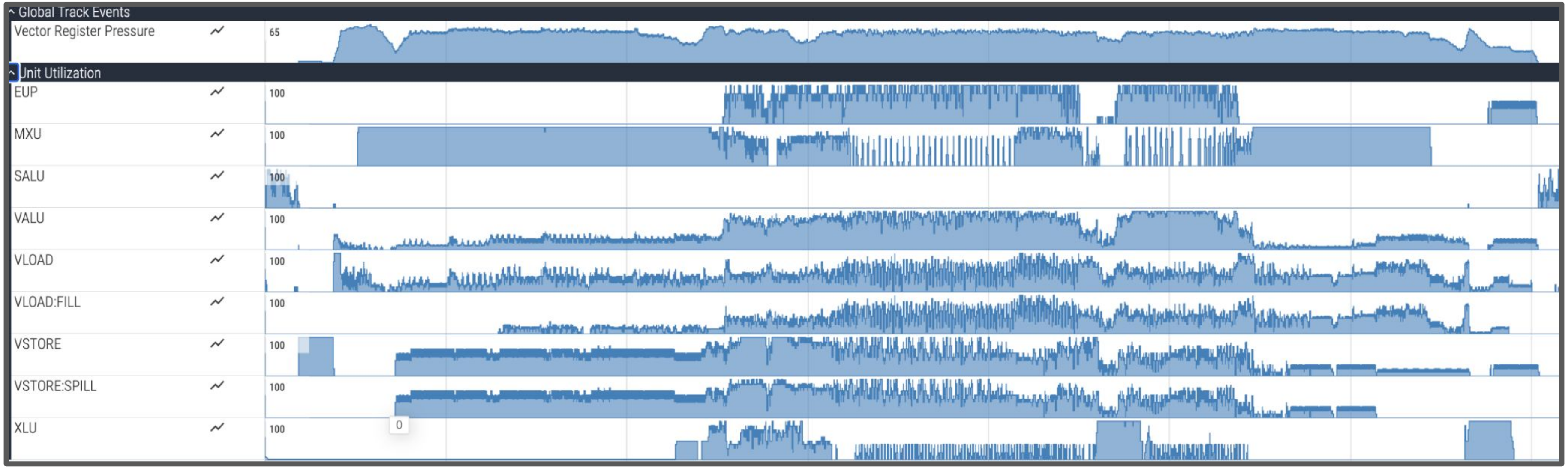
Reading Pipelining and Parallelism

The most critical information in the trace is how the compiler **hides latency** by overlapping the three execution phases across cycles.

Trace Feature	Interpretation	Performance Insight
Instruction-Level Parallelism	Instructions scheduled horizontally within a narrow time slice (a "packet" or "bundle") execute in the same clock cycle.	Example: **vmatmul.mxu0** and **vmatmul.mxu1** running concurrently means the single Matrix operation is split and executed simultaneously across the accelerator's two MXUs.
MXU → ALU Pipelining	Producer-Consumer Handshake: A vpop instruction (MXU finish) is immediately followed or overlapped by a vmax or vpack instruction (ALU start).	Softmax Fusion: The ALU starts processing the first tiles of the QK^T matrix (e.g., finding Mcurr) before the MXU has even finished calculating the entire matrix.
Latency Hiding (Setup)	Overlapping Operations: vmatprep and **vmatpush** (setup for the next **vmatmul**) are scheduled while the current vmatmul is running.	Sustained Throughput: The MXU's input buffers are continuously replenished, ensuring that when the vmatmul instruction finishes, the hardware can immediately launch the next one without stalling.
Vertical Alignment (VLD/VST)	Input/Output Bandwidth: High density of vld (Load) operations before the vmatmul and dense vst (Store) operations after the vmax show memory access is closely managed to feed the compute units.	Memory Bottleneck Detection: If the compute bars drop or become sparse due to a cluster of VLDs or VSTs, it suggests a memory bandwidth bottleneck.

Multi-Head Attention - Pacchetto Trace

Despite achieving high MXU compute saturation and successful pipeline fusion, the program is critically **memory bound** due to extreme **Vector Register Pressure** that forces frequent, latency-inducing VSTORE.SPILL activity and intermittent I/O stalls



Performance Bottleneck Observations from Trace

Sporadic MXU Stalls: The Matrix Multiply Unit shows high-frequency idling, signaling data starvation where the compute engine is consistently waiting on the pipeline.

Saturated Register Pressure: Vector registers are at 100% capacity, triggering constant, expensive VSTORE.SPILL operations to offload data and free up register space.

Memory-Bound Fragmentation: Disjointed VLOAD/VSTORE activity confirms the kernel is bound by HBM bandwidth, with compute units stalled during data transfers.

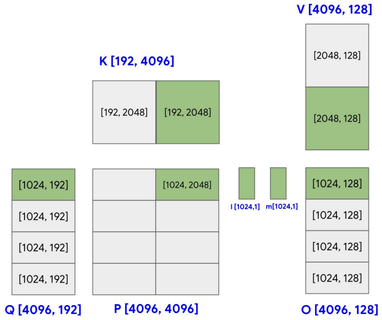
Inefficient Compute Overlap: Intermittent VALU activity indicates that vector math (like Softmax) is failing to hide behind the MXU's matrix operations.

Low Sustained MFU: The profile lacks a "steady state," showing high-peak bursts rather than the sustained utilization required for high Model FLOPs Utilization.

Tuning Block Sizes for Peak MXU Sustained Utilization

- **Optimal Block Size Search:** Systematically test and profile the largest possible br, bc values that fit within VMEM to **maximize the compute-to-load ratio** and sustain the MXU.

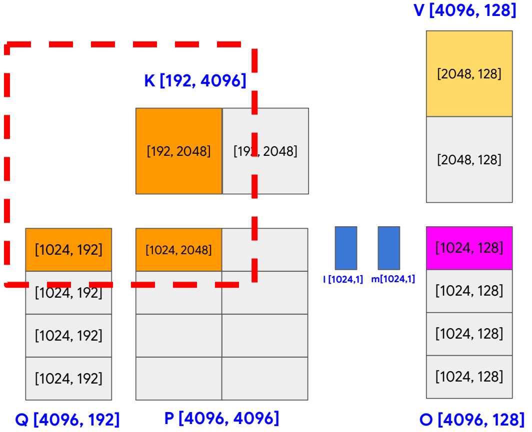
Multi-Head Attention Q/K: [128, 4096, 192] and V: [128, 4096, 128] Peak FLOP Rate per Ironwood TensorCore: 1028.75 TFLOP/s				
Block Sizes (br,bc)	**Grid**	**Execution Time (ms)**	**Achieved TFLOPs/s**	**MFU Utilization**
1024, 2048	(128, 4, 2)	4.03	343.35	33.37%
2048, 2048	(128, 2, 2)	3.36	408.75	39.73%
4096, 4096		N/A	N/A	Memory Limit Exceeded

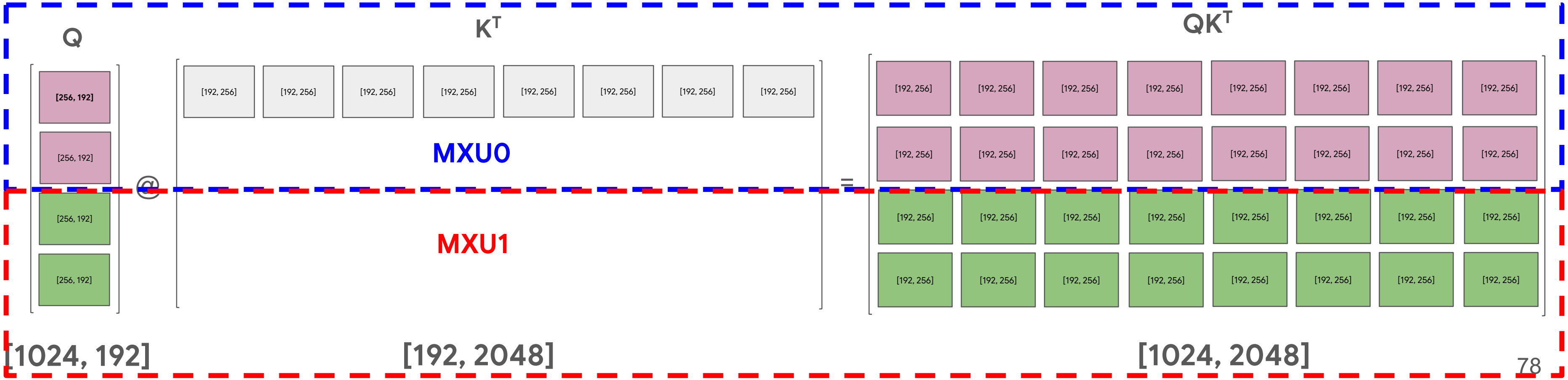


Increasing the Query block size (br) from 1024 to 2048 (Scenario 1 to 2) resulted in a significant 19% performance gain (343.35 to 408.75 TFLOPs/s), demonstrating that the kernel was indeed **Q-block size limited.**

Pallas Block Tiling & MXU Configuration

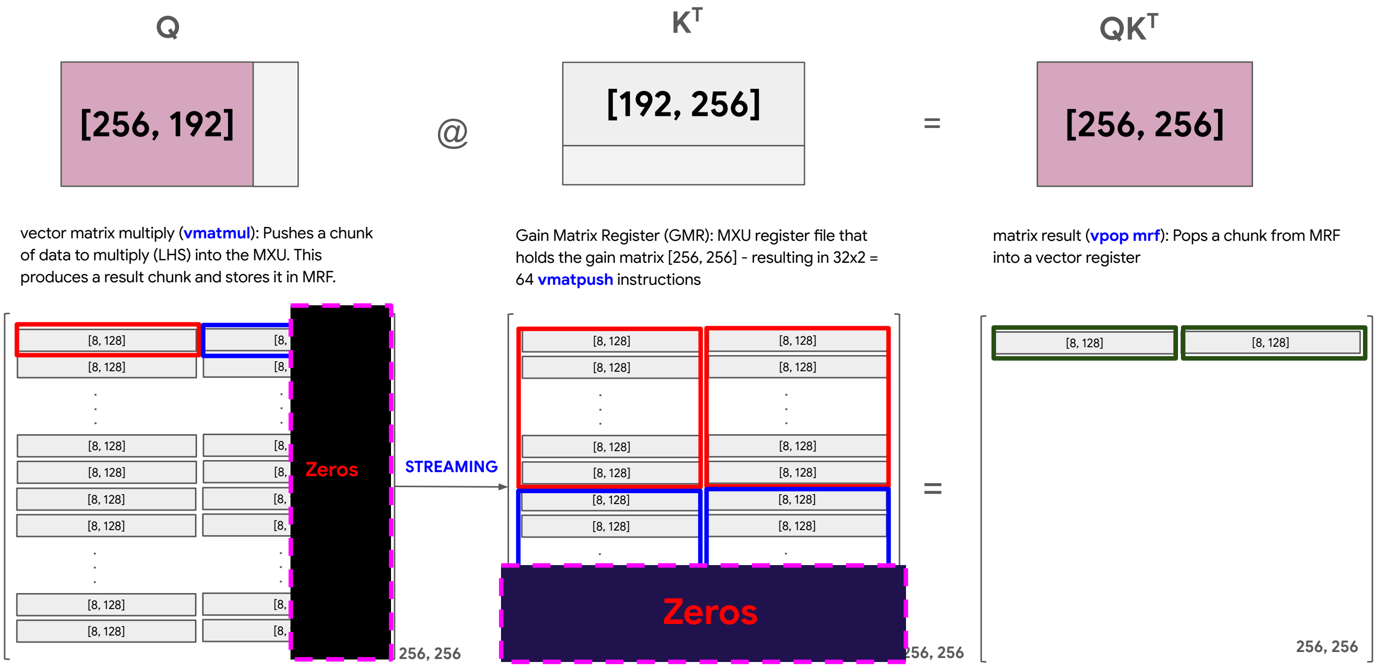
- **Pallas Block:** The kernel processes a Query block Q∈[1024,192] and a Key block K∈[2048,192]. The inner loop iterates over 8 blocks of K (j=0 to 7).
- **MXU Unit:** Ironwood uses **DUAL** [256,256][256, 256] **MXUs**
- **Dual MXU Parallelism (Row-Based):** The 32 output tiles are split by Q rows (16 tiles/MXU) to ensure balance and data locality:
  - **MXU 0:** Handles the top 512 rows of Q
  - **MXU 1:** Handles the bottom 512 rows of Q





MXU Data Alignment & Implicit Zero Padding

The LLO implicitly pads the Q and K 192-dimension to the required 256 size. **K (Gain Register):** K^T uses ≈ 48 explicit vmatpush instructions. The remaining ¼ set to zero by vmatprep (implicit zeroing), saving 16 redundant push operations.



MXU/VALU Pipelining

MXU/VALU pipelining achieves **zero memory Softmax fusion** by transferring matrix tiles directly via **registers**, completely bypassing slow VMEM access.

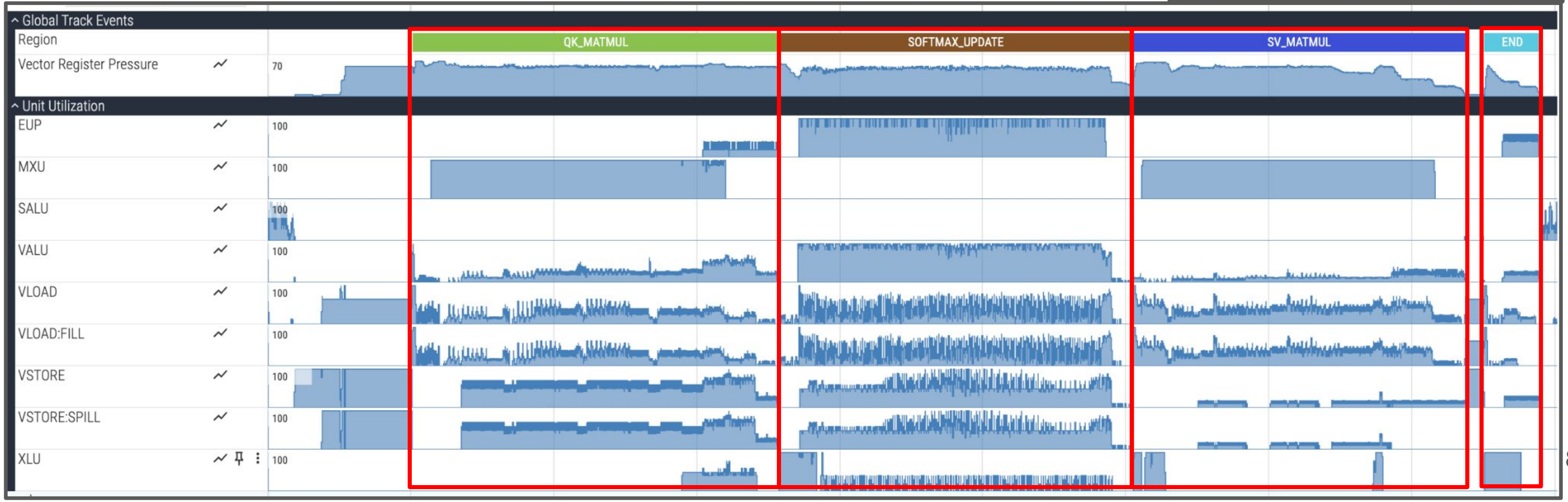
```
qk = jnp.dot(q, k.T)

m_curr = qk.max(axis=-1)[:, None]
m_next = jnp.maximum(m_prev, m_curr)
s_curr = jnp.exp(qk - m_next)
l_curr = s_curr.sum(axis=-1)[..., None]
alpha = jnp.exp(m_prev - m_next)
l_next = l_curr + alpha * l_prev
m_ref[...], l_ref[...] = m_next, l_next

o_curr = jnp.dot(s_curr, v)
o_ref[...] = alpha * o_ref[...] + o_curr

@pl.when(pl.program_id(2) == grid_width - 1)
def end():
  o_ref[...] = (o_ref[...]/ l_next[...]).astype(o_ref.dtype)
```

Phase	Core Code Operation	Hardware Unit	LLO Instruction Flow	Key Action & Fusion
1. Softmax Numerator Prep	qk=QK^T ⟶ Mcurr, Scurr	MXU ⟶ VALU/XLU	vmatmul⟶vpop⟶vmax⟶vsub/vexp	QK Pipelining: VALU instantly consumes QK tiles to compute the max/exp terms (M, S).
2. L-Term Update (Scalar)	α=exp(Mprev−Mnext) ⟶ lnext=lcurr+α∗lprev	VALU/XLU (+SALU)	vsub⟶vexp⟶vmul⟶vadd	Critical Fusion: L-term update is pure vector/scalar math, running completely on the VALU concurrently with the main MXU/IO flow.
3. Output Accumulation	Ocurr=Scurr·V ⟶ Oref.update(·)	VALU ⟶ MXU ⟶ VALU	vpack⟶vmatmul⟶vpop⟶vmul⟶vadd	SV Pipelining: VALU output (S) immediately feeds the MXU for the second matrix multiply (SV), and the result is accumulated by the VALU.

Named Scopes and implicit barriers

```
q, k, v = q_ref[...], k_ref[...], v_ref[...]
m_prev, l_prev = m_ref[...], l_ref[...]

with named_scope("QK_MATMUL"):
  qk = jnp.dot(q, k.T)
  m_curr = qk.max(axis=-1)[:, None]
  m_next = jnp.maximum(m_prev, m_curr)
  alpha = jnp.exp(m_prev - m_next)

with named_scope("SOFTMAX_UPDATE"):
  s_curr = jnp.exp(qk - m_next)
  l_curr = s_curr.sum(axis=-1)[..., None]
  l_next = l_curr + alpha * l_prev

with named_scope("SV_MATMUL"):
  o_curr = jnp.dot(s_curr, v)
  o_ref[...] = alpha * o_ref[...] + o_curr

m_ref[...], l_ref[...] = m_next, l_next

@pl.when(pl.program_id(2) == grid_width - 1)
@named_scope("END")
def end():
  o_ref[...] = (o_ref[...]/ l_next[...])
```

Named scopes confirmed three distinct, non-overlapping execution phases because the huge intermediate qk^T matrix acts as a compulsory synchronization point. This structure is an **implicit barrier**: the MXU must write the entire matrix to slower VMEM, and the VALU must read it back before the SV multiplication can commence.



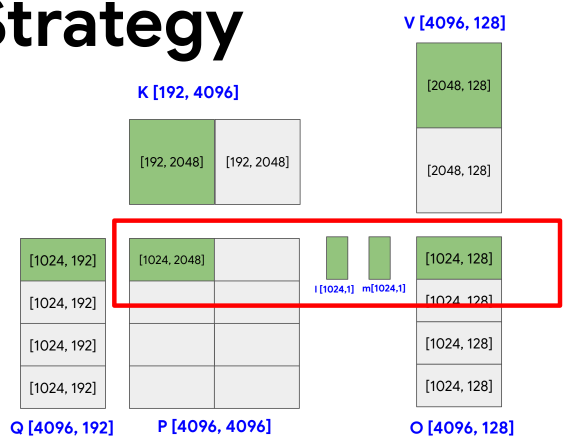
Micro-Tiling: The Memory Locality Strategy

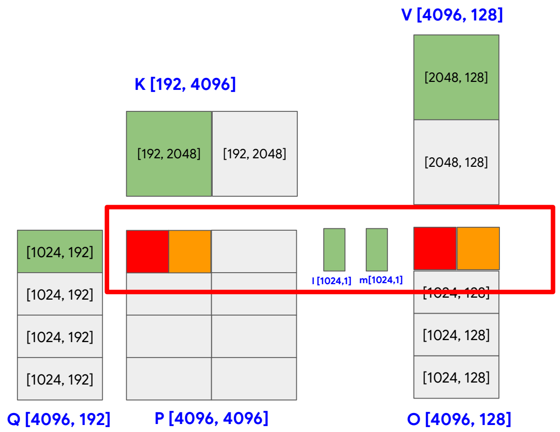
Explicit Matrix Decomposition: Strategically breaks down massive global operations (e.g., [1024, 2048]) into a series of smaller, iterative tiles (e.g., [1024, 1024]) to match the physical constraints of the hardware.

Register-File Optimization: Identifies the precise block size that fits entirely within the processor's **fastest on-chip registers**, ensuring data stays as close to the compute units as possible.

Elimination of Memory Spills: By keeping intermediate values in registers, it prevents the system from triggering **VSTORE.SPILL** operations—expensive transfers that offload data to slower memory when registers overflow.

Zero-Stall Pipeline Fusion: Enables continuous saturation of the **MXU** and **VALU** by guaranteeing that the next set of operands is already staged in registers, effectively eliminating pipeline stalls.

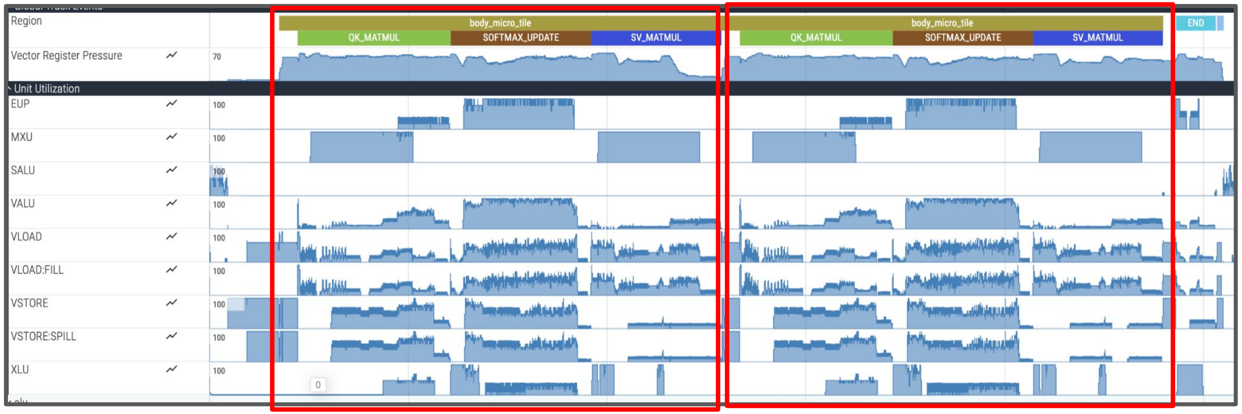




Micro-tiling Pacchetto Trace

- **Micro-tiling** breaks the required QK^T block of [1024, 2048] into two manageable [1024, 1024] ([1024, 192] @ [192, 1024] chunks inside the kernel (based on bkv_compute=2).
- This sacrifices minimal **loop overhead** to gain the benefit of **eliminating memory stalls**, as the smaller chunks now fit entirely within the fast on-chip registers, achieving maximum throughput (Remove named_scope while testing)

```python
@named_call
def body_micro_tile(k_micro_idx, carry):
  m_prev_inner, l_prev_inner, o_prev_inner = carry

  # Calculate the slice for the current K/V micro-block
  k_slice = pl.ds(k_micro_idx * bkv_compute, bkv_compute)
  k_micro = k_ref[k_slice, :]
  v_micro = v_ref[k_slice, :]

  with named_scope("QK_MATMUL"):
    qk = jnp.dot(q, k_micro.T)
    m_curr = qk.max(axis=-1)[:, None]
    m_next = jnp.maximum(m_prev_inner, m_curr)
    alpha = jnp.exp(m_prev_inner - m_next)

  with named_scope("SOFTMAX_UPDATE"):
    s_curr = jnp.exp(qk - m_next)
    l_curr = s_curr.sum(axis=-1)[:, None]
    l_next = alpha * l_prev_inner + l_curr

  with named_scope("SV_MATMUL"):
    o_curr = jnp.dot(s_curr, v_micro)
    o_next = alpha * o_prev_inner + o_curr

  return m_next, l_next, o_next

# Initialize inner loop state with loaded M/L/O (inner micro-tiling
initial_carry = (m_prev, l_prev, o_prev)
m_final, l_final, o_final = lax.fori_loop(
    0, num_micro_iters, body_micro_tile, initial_carry, unroll=True
)

# Write back the final accumulated state
m_ref[...], l_ref[...] = m_final, l_final

@pl.when(pl.program_id(2) == grid_width - 1)
@named_scope("END")
def end():
  l_inv = 1.0 / l_final
  o_ref[...] = (o_final * l_inv).astype(o_ref.dtype)

# write back the accumulated state for the next call to read
@pl.when(pl.program_id(2) != grid_width - 1)
@named_scope("interim_write")
def interim_write():
  o_ref[...] = o_final.astype(o_ref.dtype)
```



Max Logit Estimate: Bypassing Vector Overhead

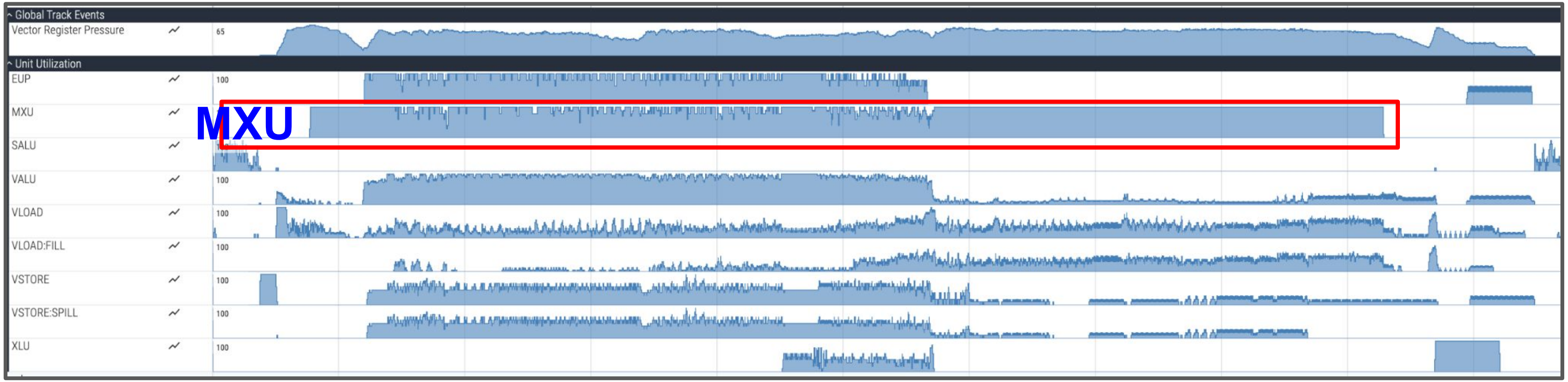
The **Max Logit Estimate MLE strategy** skips complex pipeline steps by substituting a correct constant value for the m_next calculation.

- **Bypasses Slow Reduction:** The optimization skips the slow, vector-intensive step of calculating the true maximum vmax in each block.
- **Substitutes Constant:** We substitute the dynamically calculated maximum m_max with a **pre-determined, constant** max_logit_estimate
- **Simplifies the Pipeline:** This elimination removes complex vector arithmetic operations vmax and jnp.maximum from the critical path, allowing the compiler to generate **simpler, faster machine code**.
- **Achieves Faster Throughput:** By simplifying the math, the processor spends less time on control flow and vector computation, achieving higher throughput.

```
qk = jnp.dot(q, k.T)
if not use_estimate:
  m_curr = qk.max(axis=-1)[:, None]
  m_next = jnp.maximum(m_prev, m_curr)
  alpha = jnp.exp(m_prev - m_next)
  logit_subtractor = m_next
else:
  m_next = jnp.full_like(m_prev, max_logit_estimate)
  alpha = jnp.full_like(m_prev, 1.0)
  logit_subtractor = m_next

s_curr = jnp.exp(qk - logit_subtractor)
l_curr = s_curr.sum(axis=-1)[..., None]
l_next = l_curr + alpha * l_prev
o_curr = jnp.dot(s_curr, v)
o_ref[...] = alpha * o_ref[...] + o_curr

if not use_estimate:
  m_ref[...], l_ref[...] = m_next, l_next
else:
  l_ref[...] = l_next

@pl.when(pl.program_id(2) == grid_width - 1)
def end():
  o_ref[...] = (o_ref[...]/ l_next[...]).astype(o_ref.dtype)
```

Flash Attention MFU: Logit Estimate Performance

Multi-Head Attention Q/K: [128, 4096, 192] and V: [128, 4096, 128] Peak FLOP Rate per Ironwood TensorCore: 1028.75 TFLOP/s				
Block Sizes (br,bc)	**Grid**	**Execution Time (ms)**	**Achieved TFLOPs/s**	**MFU Utilization**
1024, 2048	(128, 4, 2)	4.03	343.35	33.37%
2048, 2048	(128, 2, 2)	3.36	408.75	39.73%
2048, 2048 (MLE)	**(128, 2, 2)**	**2.64**	**520.23**	**50.27%**
4096, 4096		N/A	N/A	Memory Limit Exceeded



Flash Attention Optimization Summary

- **Optimized I/O Block Sizes:** Outer block dimensions (br, bc) are carefully selected and **manually tuned** to maximize memory concurrency and the data transfer rate between slower HBM and on-chip memory VMEM.
- **Micro-Tiling for Register Locality:** Matrix segments are subdivided (bc=2048 ⟶ bkv_compute=1024) to guarantee intermediate data fits entirely within the **fast on-chip registers**. This is a **memory locality strategy** that eliminates VSTORE.SPILL stalls.
- **Zero-Memory Pipeline Fusion:** The MXU and VALU achieve continuous operation by transferring data tiles directly via **registers**. This handshake ensures co-saturation of both units and prevents stalls associated with buffering massive intermediate matrices.
- **Arithmetic Simplification (MLE):** The **Max Logit Estimate (MLE)** optimization allows the kernel to substitute a correct constant for the dynamic M term, entirely bypassing the expensive VALU maximum vector reduction vmax instruction.
- **Delayed Normalization & Dynamic Scaling:** The full normalization (division) is **deferred until the final block**. Instead, the alpha factor is calculated every block to dynamically **re-scale previous accumulated outputs** (L, O), maintaining numerical stability using only fast multiplication and addition.

https://github.com/openxla/tokamax/tree/main/tokamax/_src/ops/experimental/tpu/splash_attention

Splash Attention

Flash Attention - Recap

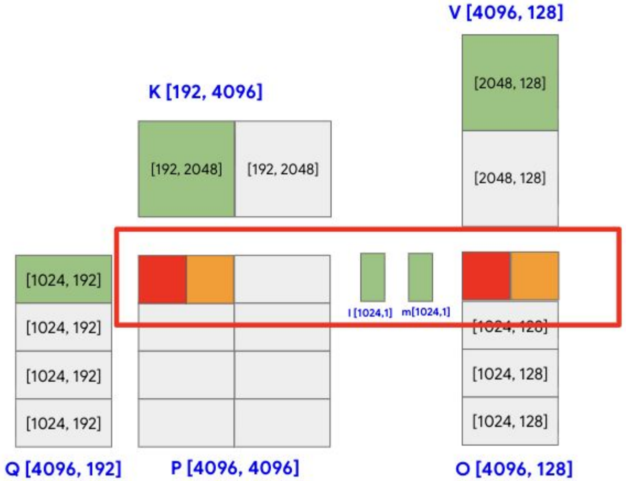
Optimized I/O Tiling: Hand-tuned block sizes to maximize HBM-to-VMEM data transfer speeds.

Micro-Tiling: Subdividing tiles to keep data in registers and eliminate memory spills.

Pipeline Fusion: Direct register handshakes between MXU and VALU to prevent execution stalls.

Arithmetic Simplification (MLE): Skipping expensive max-vector reductions using constant estimates.

Delayed Normalization: Using per-block scaling instead of division for speed and stability.




```
# Initialize inner loop state with loaded M/L/O (inner micro-tiling
initial_carry = (m_prev, l_prev, o_prev)
n_final, l_final, o_final = lax.fori_loop(
    0, num_micro_iters, body_micro_tile, initial_carry, unroll=True
)
```

```
@pl.when(pl.program_id(2) == grid_width - 1)
@named_scope("END")
def end():
  l_inv = 1.0 / l_final
  o_ref[...] = (o_final * l_inv).astype(o_ref.dtype)
```

Masks and Sparsity

Causal & Local Constraints: Masks enforce architectural rules, such as Causal Masks in decoders to prevent "looking ahead" at future tokens, and Local Attention Masks that restrict focus to a sliding window of neighboring context.

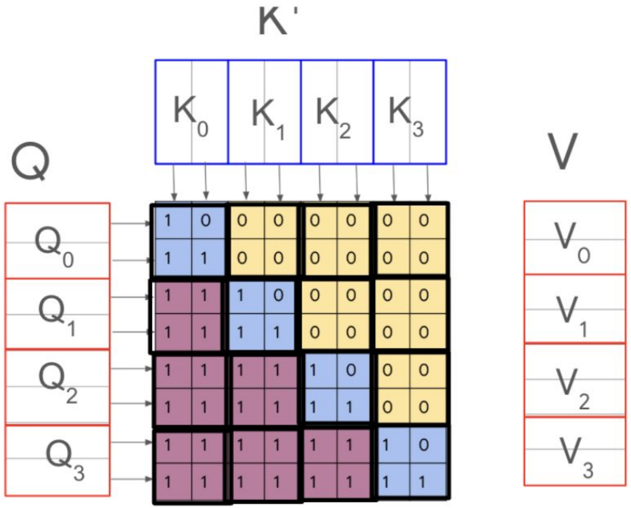
Logical Sparsity: By zeroing out specific connections, masks transform dense attention into a sparse operation where the model only processes relevant scores, significantly reducing the theoretical workload.

Computational Skipping: Hardware-aware kernels leverage this sparsity to bypass the calculation of masked regions entirely, saving both memory bandwidth and MXU cycles.

Throughput Acceleration: Transforming masks into a sparse execution map allows the hardware to skip "empty" blocks, converting logical constraints into physical speedups and reduced power consumption.

Causal Mask, seq_length = 8
Bq = 2, Bkv = 2

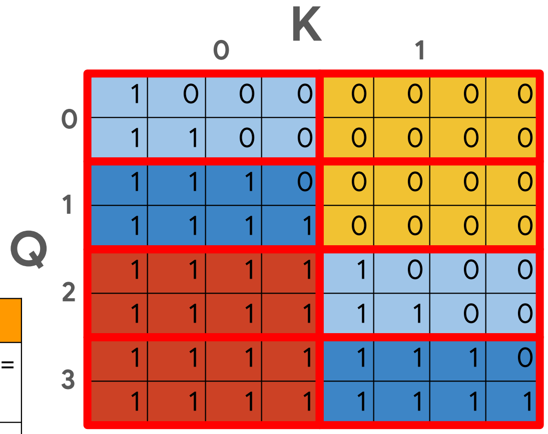
- Yellow - all zeros (no computation)
- Blue: Partial Mask
- Read: Full attention



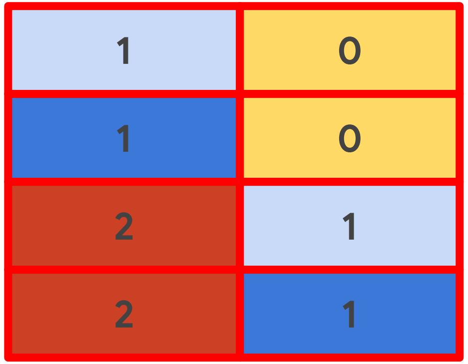
Pre-Processing Mask Information

- **Sparsity Pruning:** The [4096, 4096] mask is divided into 4x2 blocks ([1024, 2048]), and all inactive blocks are discarded, creating a compact map of only **6 active blocks** for computation.
- **Data Compression:** The system analyzes the active blocks and efficiently finds that it only needs to store **2 unique raw mask patterns** in memory (partial_mask_blocks), drastically reducing memory usage.
- **Execution Map & Prefetching:** The arrays (active_rows, mask_next) are generated to explicitly map the computation order and tell the MXU which of the 2 unique patterns to **prefetch** for zero-latency application.

Array Name	Value	Notes
Block Mask	[1 1 2 1 2 1 0 0]	Tells the MXU if it needs a mask (1 = Partial) or not (2 = Full).
active_rows Q-axis index	[0 1 2 2 3 3 0 0]	Active block row index
active_cols KV-axis index	[0 0 0 1 0 1 0 0]	Active block col index
partial_mask_blocks	Shape (2,1024,2048)	Contains the 2 unique raw mask patterns (Pattern 0 and Pattern 1).
mask_next	[0 1 0 0 1 1 0 0]	Crucial: Tells the hardware which of the 2 unique patterns to load next for each active block.
num_active_blocks	6	

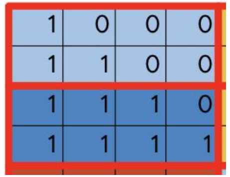


Block Sizes [1024, 2048]
Seq_length [4096]
Block Mask Shape: [4, 2]



Block Mask

Partial masks
(2 unique pattern)
Shape: [2, 1024, 2048]



Sparse Execution Metadata Structures

Array Name	Value	Notes
Block Mask	[1 1 2 1 2 1 0 0]	Tells the MXU if it needs a mask (1 = Partial) or not (2 = Full).
active_rows Q-axis index	[0 1 2 2 3 3 0 0]	Active block row index
active_cols KV-axis index	[0 0 0 1 0 1 0 0]	Active block col index
partial_mask_blocks	Shape (2,1024,2048)	Contains the 2 unique raw mask patterns (Pattern 0 and Pattern 1).
mask_next	[0 1 0 0 1 1 0 0]	Crucial: Tells the hardware which of the 2 unique patterns to load next for each active block.
num_active_blocks	6	

Block Filtering (block_mask): Categorizes QK blocks as zero, non-zero, or partial, allowing the kernel to skip empty regions and focus MXU cycles only on active data.

Coordinate Mapping (active_rows/cols): Provides the hardware with precise (Q, KV)$grid indices to calculate exact memory addresses, ensuring efficient loading of active blocks from HBM.

Mask Pipelining (mask_next): Stores indices that allow the MXU to prefetch the next required partial mask pattern with zero latency while the current computation is in flight.

SMEM-Resident Metadata: Computed once per mask and stored in SMEM, these structures enable high-speed access to the sparse execution map without redundant HBM transfers.

Tokamax Splash Attention Kernel

[splash_attention/]

```
import jax
import jax.numpy as jnp
from tokamax._src.ops.experimental.tpu.splash_attention
        import splash_attention_kernel as splash
from tokamax._src.ops.experimental.tpu.splash_attention
        import splash_attention_mask as mask_lib

(bs, q_heads, kv_heads) = (8, 128, 128)
(q_seq_len, kv_seq_len) = (4096, 4096)
(qk_head_dim, v_head_dim) = (192, 128)

mask = mask_lib.make_causal_mask((q_seq_len, kv_seq_len))
config = splash.SplashConfig(
 block_q=1024,
 block_kv=2048,
 block_kv_compute=256,
)
attn_fn = splash.make_splash_mha(mask, is_mqa=False, q_seq_shards=1, config=config)
attn_fn = jax.vmap(attn_fn, in_axes=(0, 0, 0))

key, key1, key2, key3, key4 = jax.random.split(jax.random.key(42), 5)
q = jax.random.uniform(key1, (bs, q_heads, q_seq_len, qk_head_dim), dtype=jnp.float32)
k = jax.random.uniform(key2, (bs, kv_heads, kv_seq_len, qk_head_dim), dtype=jnp.float32)
v = jax.random.uniform(key3, (bs, kv_heads, kv_seq_len, v_head_dim), dtype=jnp.float32)

o = attn_fn(q, k, v).block_until_ready()
```

Static Sparse Mapping: make_splash_mha pre-calculates the MaskInfo structure to identify active blocks (e.g., 6 of 8), allowing the hardware to skip empty regions and focus resources only on non-zero data.

Zero-Latency Prefetching: The kernel uses the mask_next array to prefetch upcoming mask metadata during active computation, ensuring the MXU pipeline remains fully saturated without memory stalls.

Splash Attention (Sparse + Flash = Splash)

1. **Identifying Inactive Regions:** The Mask Processing Function analyzes the mask to identify and skip all entirely zero (all-zero) blocks that the query should not attend to. This immediately prunes the vast majority of unnecessary computation.

2. **Execution Metadata:** The function computes auxiliary data structures (like block_mask, active_rows, and active_cols) that serve as the sparse execution roadmap, defining the precise coordinates of every active block to be computed.

3. **Fine-Grained Masking:** The system applies fine-grained masking only where necessary (on partial_blocks) using compressed partial_masks and pre-computed prefetch indices (mask_next) to efficiently access the mask data.

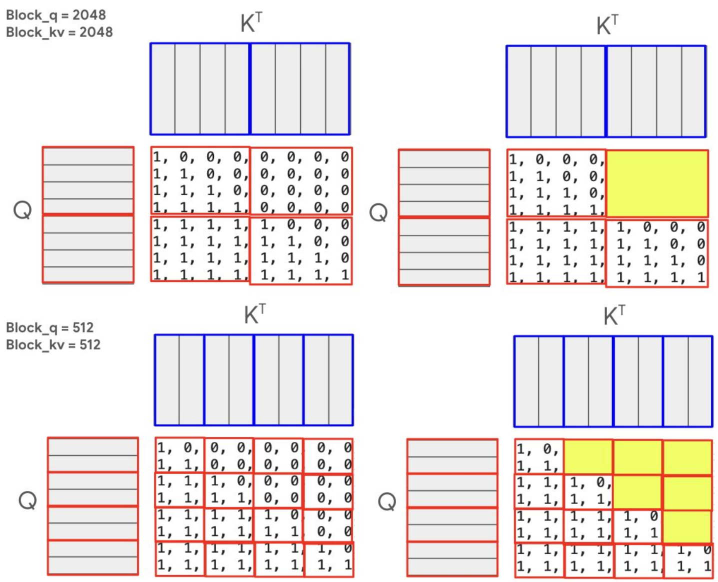
The Challenge of Causal Masking on Smaller Seq Lengths

The MFU vs. Efficiency Trade-off: Large block sizes (e.g., **2048**) are required to saturate Ironwood's massive MXU for peak throughput, but they force the hardware to calculate large "fill-in" areas of zeros within a causal mask.

Granularity vs. Utilization: Smaller blocks (e.g., **512**) better prune the causal mask by skipping more empty regions, yet they often collapse **Model FLOPs Utilization (MFU)** by failing to keep hardware pipelines full.

The "Sweet Spot" Search: Optimal performance requires finding a block size that is large enough to maintain high compute density but small enough to minimize wasted computation on masked-out tokens.

Memory vs. Compute Bottlenecks: On shorter sequences, the management overhead of small blocks can shift the bottleneck from raw compute to memory bandwidth, further complicating the scaling strategy.



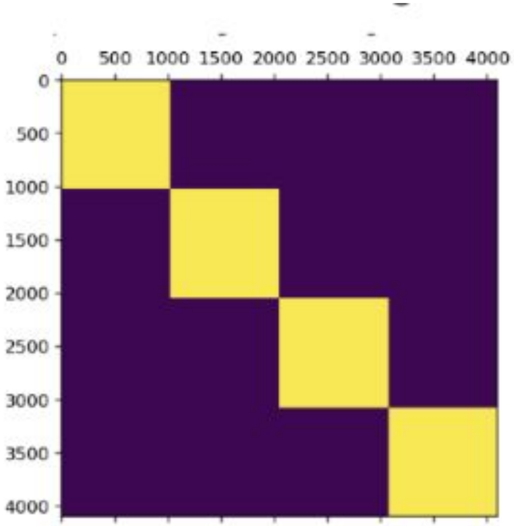
Segment IDs

Sequence Partitioning: Assigns 1D integer arrays to Q and KV tokens to identify independent samples within a "packed" sequence, preventing unrelated data from interacting.

Identity-Based Masking: Enforces a strict rule where tokens only attend to others with matching IDs, effectively eliminating information leakage between concatenated samples.

Block-Diagonal Sparsity: Naturally creates a structural pattern where only clusters along the diagonal are active, while the rest of the matrix remains empty.

Skip-Computation Optimization: High-performance kernels use these IDs to bypass "off-diagonal" blocks entirely, saving significant memory bandwidth and MXU cycles.



Segment ID Masks and Joint Masking

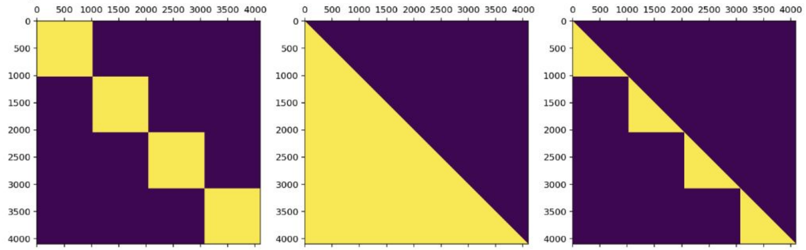
Logical Conjunction: Combines segment IDs with **causal** or **local** masks via a bitwise **AND** operation. Attention is only permitted if all mask conditions are simultaneously met.

Enforced Isolation: Restricts attention to within-sample tokens, ensuring that even if a token is "local" or "causal," it cannot attend to data belonging to a different segment ID.

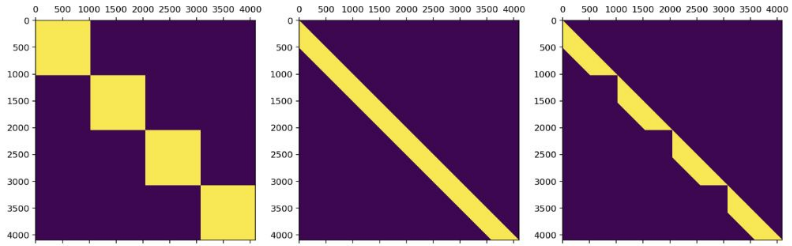
Compound Sparsity: Merging masks increases the total number of empty blocks (e.g., a **Local + Segment** mask), creating a highly constrained diagonal that maximizes skipped computations.

Unified Execution Map: The combined results are baked into a single MaskInfo structure, allowing the hardware to follow one optimized sparse path rather than evaluating multiple masks.

segment mask [4096, 4096] with 4 segments,
causal mask, segment mask & causal mask



segment mask [4096, 4096] with 4 segments,
Local windows mask, segment mask & local_window_mask



Splash Attention Pallas Call

- **Parallel Execution Grid**
  - grid=(num_q_heads, num_active_blocks)
- **Tiling Sizes**
  - bq, bkv, and bkv_compute define the memory blocks and compute chunk sizes used by the kernel's tiling logic
- **Masking/Sparse Info (SMEM)**
  - fwd_mask_info.{ block_mask, mask_next, active_rows, active_cols, partial_mask_blocks} pass the pre-calculated tables needed by the next_nonzero function to enable sparse and segmented attention traversal - SMEM
- **Inputs (VMEM)**
  - q, k, and v
  - q_segment_ids, kv_segment_ids
  - q_sequence
    - q_sequence holds the **absolute token indices** for the current Query block. If the full sequence has L=8192 and the block starts at index 2048, the tensor contains 2048,2049,...,4095.

```python
if fwd_mask_info.num_active_blocks is not None:
  grid_size = fwd_mask_info.num_active_blocks[0]
else:
  grid_size = kv_steps * (q_seq_len // bq)

grid = (num_q_heads, grid_size)

with jax.named_scope(kernel_name):
  all_out = pl.pallas_call(
      partial(
          flash_attention_kernel,
          mask_value=mask_value,
          kv_steps=kv_steps,
          bq=bq,
          bkv=bkv,
          bkv_compute=bkv_compute,
          head_dim_v=head_dim_v,
          # note: fuse_reciprocal can only be False if save_residuals is True
          # fuse_reciprocal = (config.fuse_reciprocal or not save_residuals)
          fuse_reciprocal=fuse_reciprocal,
          config=config,
          mask_function=mask_function,
      ),
      grid_spec=pltpu.PrefetchScalarGridSpec(
          num_scalar_prefetch=6,
          in_specs=in_specs,
          out_specs=out_specs,
          grid=grid,
          scratch_shapes=[
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # m_scratch
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # l_scratch
              pltpu.VMEM((bq, head_dim_v), jnp.float32),  # o_scratch
          ],
      ),
      compiler_params=pltpu.CompilerParams(
          dimension_semantics=("parallel", "arbitrary"),
      ),
      out_shape=out_shapes,
      name=kernel_name,
      cost_estimate=cost_estimate,
      interpret=config.interpret,
      metadata=metadata,
  )(
      fwd_mask_info.active_rows,
      fwd_mask_info.active_cols,
      fwd_mask_info.mask_next,
      bounds_start,
      bounds_end,
      fwd_mask_info.block_mask,
      q if q_layout == QKVLayout.HEAD_DIM_MINOR else q.swapaxes(-1, -2),
      k if k_layout == QKVLayout.HEAD_DIM_MINOR else k.swapaxes(-1, -2),
      v if v_layout == QKVLayout.HEAD_DIM_MINOR else v.swapaxes(-1, -2),
      q_segment_ids,
      kv_segment_ids,
      fwd_mask_info.partial_mask_blocks,
      q_sequence,
      max_logit_value,
  )
```

Pallas Call: Execution Grid & Inputs

Grid Definition: The grid is defined by two dimensions: (num_q_heads, grid_size). Grid_size is set dynamically using the pre-computed sparse metadata: fwd_mask_info.num_active_blocks[0]. This ensures threads are only launched for the active, non-zero regions of the mask, maximizing efficiency

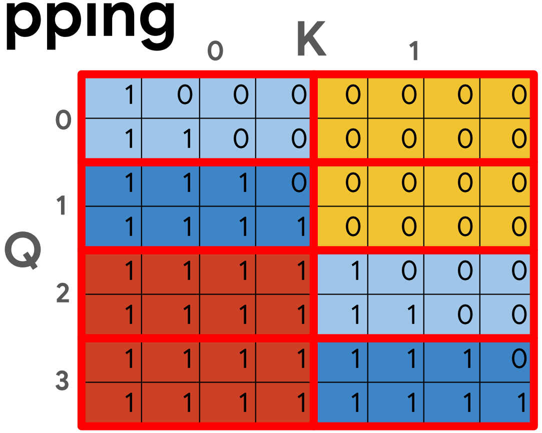
SMEM Inputs: num_scalar_prefetch=6 This instructs the hardware to aggressively prefetch the critical scalar/metadata inputs (e.g., active_rows, mask_next) into the fastest SMEM *before* the thread begins computing.

VMEM Inputs: (Q, K, V) and necessary metadata (segment IDs and partial_mask_blocks) that must be moved from HBM to VMEM for the kernel to execute.

```python
if fwd_mask_info.num_active_blocks is not None:
  grid_size = fwd_mask_info.num_active_blocks[0]
else:
  grid_size = kv_steps * (q_seq_len // bq)

grid = (num_q_heads, grid_size)

with jax.named_scope(kernel_name):
  all_out = pl.pallas_call(
      partial(
          flash_attention_kernel,
          mask_value=mask_value,
          kv_steps=kv_steps,
          bq=bq,
          bkv=bkv,
          bkv_compute=bkv_compute,
          head_dim_v=head_dim_v,
          # note: fuse_reciprocal can only be False if save_residuals is True
          # fuse_reciprocal = (config.fuse_reciprocal or not save_residuals)
          fuse_reciprocal=fuse_reciprocal,
          config=config,
          mask_function=mask_function,
      ),
      grid_spec=pltpu.PrefetchScalarGridSpec(
          num_scalar_prefetch=6,
          in_specs=in_specs,
          out_specs=out_specs,
          grid=grid,
          scratch_shapes=[
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # m_scratch
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # l_scratch
              pltpu.VMEM((bq, head_dim_v), jnp.float32),  # o_scratch
          ],
      ),
      compiler_params=pltpu.CompilerParams(
          dimension_semantics=("parallel", "arbitrary"),
      ),
      out_shape=out_shapes,
      name=kernel_name,
      cost_estimate=cost_estimate,
      interpret=config.interpret,
      metadata=metadata,
  )(
      fwd_mask_info.active_rows,
      fwd_mask_info.active_cols,
      fwd_mask_info.mask_next,
      bounds_start,
      bounds_end,
      fwd_mask_info.block_mask,
      q if q_layout == QKVLayout.HEAD_DIM_MINOR else q.swapaxes(-1, -2),
      k if k_layout == QKVLayout.HEAD_DIM_MINOR else k.swapaxes(-1, -2),
      v if v_layout == QKVLayout.HEAD_DIM_MINOR else v.swapaxes(-1, -2),
      q_segment_ids,
      kv_segment_ids,
      fwd_mask_info.partial_mask_blocks,
      q_sequence,
      max_logit_value,
  )
```

Block Sizes [1024, 2048]
Seq_length [4096]
Block Mask Shape: [4, 2]

Pallas Indexing: QKV Block Address Mapping



Address Lookup Inputs: The index_map receives seven arguments: (h, grid_idx, rows_ref, cols_ref, *refs). This consists of the Head Index (h), the linear Program ID (grid_idx), and several pre-fetched arrays (rows_ref, cols_ref, etc. 6 of them)

Linear to 2D Mapping: The core function (_unravel) converts the linear Program ID (grid_idx) into the 2D block coordinates. It does this by reading the pre-calculated i (Q-Row Index) and j(KV-Column Index) values from the sparse metadata arrays (rows_ref and cols_ref) at the grid_idx position.

Query (Q) Specification: The q_index_map loads Query data based only on its row. It uses the **Head Index (h)** and the calculated **Q-Row Block Index (i)** to determine the block's starting memory address. The column index (j) is discarded.

Key/Value (K, V) Specification: the k_index_map and v_index_map load data based on the column. They use the **KV-Head Index (h_kv)** and the calculated **KV-Column Block Index (j)** to find the K/V block's starting address.

Block Mask	[1 1 2 1 2 1 0 0]
active_rows Q-axis index	[0 1 2 2 3 3 0 0]
active_cols KV-axis index	[0 0 0 1 0 1 0 0]
partial_mask_blocks	Shape (2,1024,2048)
mask_next	[0 1 0 0 1 1 0 0]
num_active_blocks	6

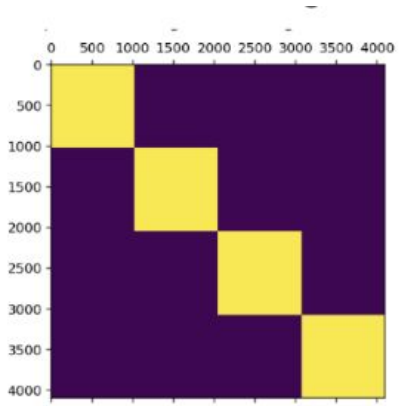
Splash Attention: Auxiliary Input Blocks

Segment IDs (Q & KV): Starting as simple {seq_len}, these IDs enforce segment boundaries to prevent cross-attention. They are sliced into bq(Q) and bkv(KV) blocks for segment-matching checks inside the kernel.

Partial Mask Blocks: This large pre-stitched array (e.g., shape [2, 1024, 2048]) holds specific element-wise masks. The mask_index_map selects and retrieves the **exact [bq , bkv] mask slice** for the current block, controlling fine-grained sparsity

Q Sequence Index: This array, originally shaped (seq_len), provides the global index of each Q token. It is sliced by bq and used exclusively for **dynamic masking**, allowing an internal function to calculate the attention mask based on token positions.

Example: 4 segments



partial_mask_blocks	Shape (2,1024,2048)
mask_next	[0 1 0 0 1 1 0 0]

Splash Output: Attention and Residuals

Final Attention Output (O): Output is written out in blocks corresponding to the input data, with a shape of (Head, head_dim_v). It represents the weighted sum of V blocks and is the final, normalized output of the attention mechanism.

LogSumExp (L Residual): Softmax denominator is written out in blocks of size (Head, bq, _), aligning with the Q-sequence block size (bq) (saved for the backward pass's numerical stability)

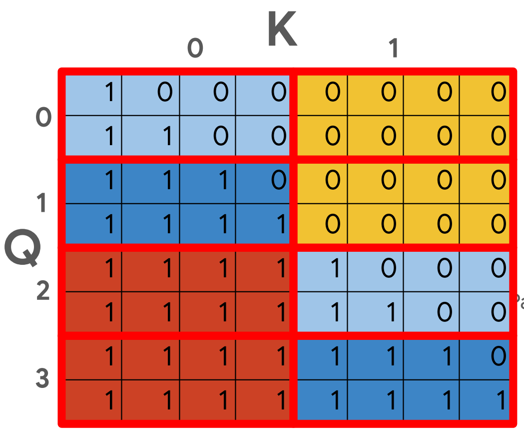
Max Logits (M Residual): Maximum logit per row for numerical shifting is also written out in blocks of size (Head, bq, _),, aligning with the Q-sequence block size bq. This residual is critical for stable gradient computation.

Scratch buffers (o_scratch, m_scratch, l_scratch) are mandatory VMEM-based accumulators that store the running total and max/log-sum of the attention calculation across K/V blocks before the final, complete result is written to the global output tensor.

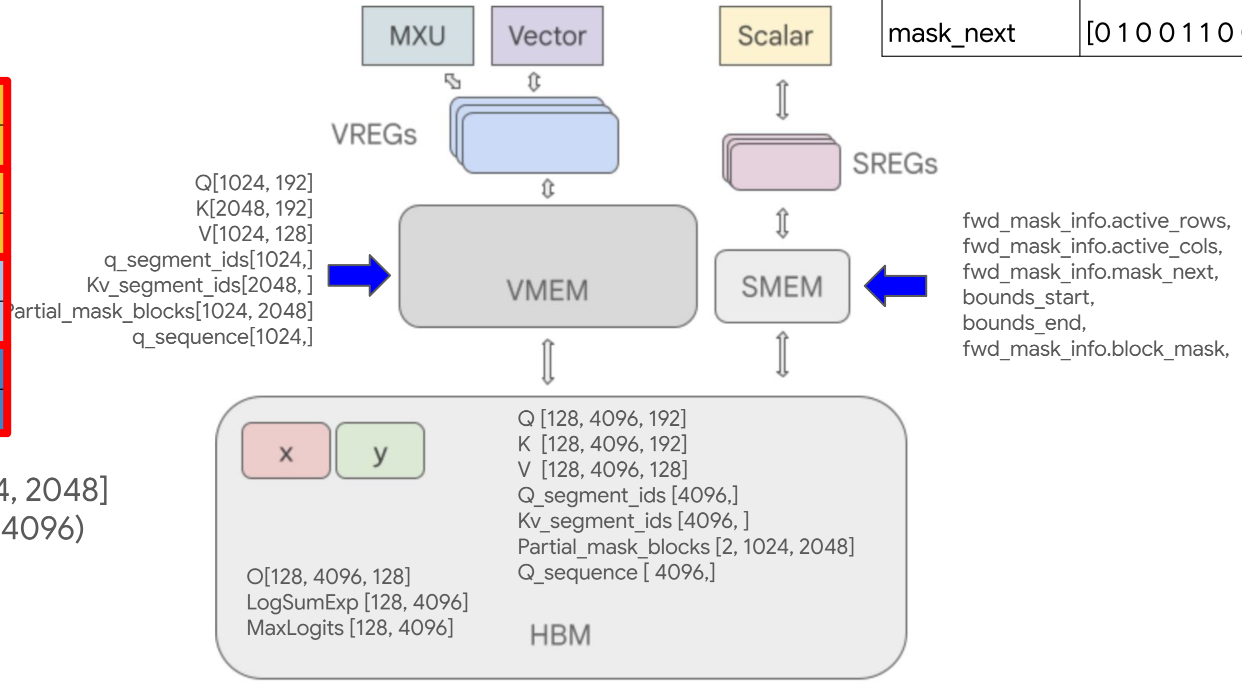
```
if fwd_mask_info.num_active_blocks is not None:
  grid_size = fwd_mask_info.num_active_blocks[0]
else:
  grid_size = kv_steps * (q_seq_len // bq)

grid = (num_q_heads, grid_size)

with jax.named_scope(kernel_name):
  all_out = pl.pallas_call(
      partial(
          flash_attention_kernel,
          mask_value=mask_value,
          kv_steps=kv_steps,
          bq=bq,
          bkv=bkv,
          bkv_compute=bkv_compute,
          head_dim_v=head_dim_v,
          # note: fuse_reciprocal can only be False if save_residuals is True
          # fuse_reciprocal = (config.fuse_reciprocal or not save_residuals)
          fuse_reciprocal=fuse_reciprocal,
          config=config,
          mask_function=mask_function,
      ),
      grid_spec=pltpu.PrefetchScalarGridSpec(
          num_scalar_prefetch=6,
          in_specs=in_specs,
          out_specs=out_specs,
          grid=grid,
          scratch_shapes=[
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # m_scratch
              pltpu.VMEM((bq, NUM_LANES), jnp.float32),  # l_scratch
              pltpu.VMEM((bq, head_dim_v), jnp.float32),  # o_scratch
          ],
      ),
      compiler_params=pltpu.CompilerParams(
          dimension_semantics=("parallel", "arbitrary"),
      ),
      out_shape=out_shapes,
      name=kernel_name,
      cost_estimate=cost_estimate,
      interpret=config.interpret,
      metadata=metadata,
  )(
      fwd_mask_info.active_rows,
      fwd_mask_info.active_cols,
      fwd_mask_info.mask_next,
      bounds_start,
      bounds_end,
      fwd_mask_info.block_mask,
      q if q_layout == QKVLayout.HEAD_DIM_MINOR else q.swapaxes(-1, -2),
      k if k_layout == QKVLayout.HEAD_DIM_MINOR else k.swapaxes(-1, -2),
      v if v_layout == QKVLayout.HEAD_DIM_MINOR else v.swapaxes(-1, -2),
      q_segment_ids,
      kv_segment_ids,
      fwd_mask_info.partial_mask_blocks,
      q_sequence,
      max_logit_value,
  )
```

Splash Attention Pallas Call Inputs/Outputs

Block Mask	[1 1 2 1 2 1 0 0]
active_rows Q-axis index	[0 1 2 2 3 3 0 0]
active_cols KV-axis index	[0 0 0 1 0 1 0 0]
mask_next	[0 1 0 0 1 1 0 0]



Here each block represents [1024, 2048] slice (q_seq_len:4096, k_seq_len:4096)



Splash Attention Kernel Outline

Kernel Control Flow

```
grid_idx = pl.program_id(1)
h = pl.program_id(0)

should_not_mask = block_mask_ref[grid_idx].astype(jnp.int32) != 1
should_initialize = bounds_start_ref[grid_idx].astype(jnp.bool_)
should_write = bounds_end_ref[grid_idx].astype(jnp.bool_)
j = active_cols_ref[grid_idx].astype(jnp.int32)
```

Initialization

```
@pl.when(should_initialize)
def init():
  o_scratch_ref[...] = jnp.zeros_like(o_scratch_ref)
  m_scratch_ref[...] = jnp.full_like(m_scratch_ref, max_logit_estimate)
  l_scratch_ref[...] = jnp.zeros_like(l_scratch_ref)
```

```
def body(kv_compute_index, _, has_partial_mask=False):
    ...
```

Accumulation Loop

```
@pl.when(should_not_mask)
def _():
  lax.fori_loop(0, num_iters, body, None, unroll=True)

@pl.when(~should_not_mask)
def _():
  lax.fori_loop( 0, num_iters, partial(body, has_partial_mask=True), None, unroll=True)
```

Final Write

```
@pl.when(should_write)
def end():
  l_inv = pltpu.repeat(1.0 /  l_scratch_ref[...], head_dim_v_repeats, axis=1)
  o_ref[...] = (o_scratch_ref[...] * l_inv).astype(o_ref.dtype)
  logsumexp = m_scratch_ref[...] + log(l)
  logsumexp_ref[...] = logsumexp.astype(logsumexp_ref.dtype)
  max_logits_ref[...] = m_scratch_ref[...].astype(max_logits_ref.dtype
```

```
def flash_attention_kernel(
    # Prefetched inputs
    active_rows_ref,
    active_cols_ref,
    mask_next_ref,
    bounds_start_ref,
    bounds_end_ref,
    block_mask_ref,
    # Inputs
    q_ref,
    k_ref,
    v_ref,
    q_segment_ids_ref,
    kv_segment_ids_ref,
    mask_ref,
    q_sequence_ref,
    max_logit_value_ref,
    # Outputs
    o_ref,
    logsumexp_ref,
    max_logits_ref,
    # Scratch
    m_scratch_ref,
    l_scratch_ref,
    o_scratch_ref,
    *,
    mask_value: float,
    kv_steps: int,
    bq: int,
    bkv: int,
    bkv_compute: int,
    head_dim_v: int,
    mask_function: MaskFuncti
    fuse_reciprocal: bool,  #
    config: SplashConfig,
```

Kernel Execution Flow

Kernel Control Flow: The kernel receives the program index (grid_idx) which is mapped directly to the bounds_start and bounds_end flags, dictating when initialization and final writing should occur for the Q-block being processed.

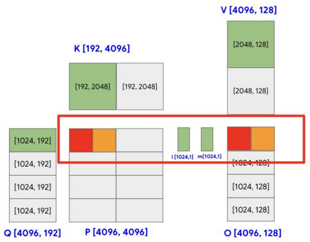
Initialization (init): Triggered by bounds_start, this executes the @pl.when(should_initialize) block to O, L M scratch accumulators in VMEM, preparing them for the accumulation phase of a new Q-block.

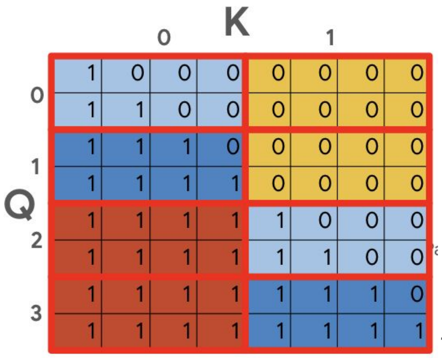
Accumulation Loop: The loop iterates over K/V blocks, with its precise inner logic controlled by block_mask_ref: it either runs the standard accumulation (if should_not_mask is true) or applies an element-wise mask (if should_not_mask is false).

Final Write (end): Triggered by bounds_end, this executes the @pl.when(should_write) block to normalize the completed output and write the final O,logsumexp, and max_logits from VMEM to HBM.

Index (grid_idx)	0	1	2	3	4	5	6	7
active_rows	0	1	2	2	3	3	0	0
bounds_start	T	T	T	F	T	F	T	F
bounds_end	T	T	F	T	F	T	F	T

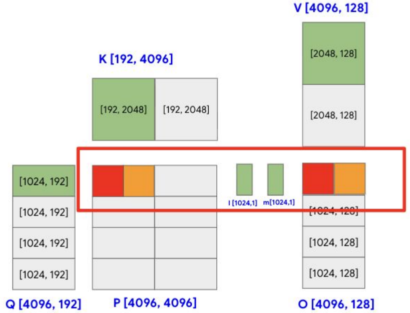
Block Row Index	grid_idx Range	Action
Row 0	0	Starts and ends immediately. Accumulation completes.
Row 1	1	Starts and ends immediately. Accumulation completes.
Row 2	2 to 3	Starts at index 2, accumulates over 2 blocks, ends at index 3.
Row 3	4 to 5	Starts at index 4, accumulates over 2 blocks, ends at index 5.





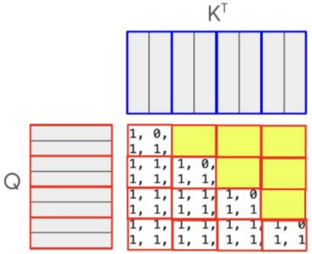
Flash Attention Kernel Body

1. Logits Computation
   - qk = lax.dot_general(q, k, qk_dims, preferred_element_type=float32)
2. Masking and Conditioning
   - qk = apply_mask_and_soft_cap() # Next Slide
3. Updating max logit
   - m_curr = qk.max(axis=-1)[:, None]
   - m_next = jnp.maximum(m_prev, m_curr)
4. Normalizing Logits and Summing Contribution
   - s_curr = jnp.exp(qk - pltpu.repeat(m_next, bkv_repeats, axis=1)
   - l_curr = jax.lax.broadcast_in_dim(s_curr.sum(axis=-1), l_prev.shape, (0,))
5. Scaling and Final Sum Update
   - alpha = jnp.exp(m_prev - m_next)
   - l_next = l_curr + alpha * l_prev
6. Weighted sum for the current segment
   - o_curr = lax.dot_general(s_curr, v, sv_dims)
7. Output Accumulator Scaling and Update
   - alpha_o = pltpu.repeat(alpha, head_dim_v_repeats, axis=1)[..., :o_scratch_ref.shape[-1]]
   - o_scratch_ref[:] = alpha_o * o_scratch_ref[:] + o_curr

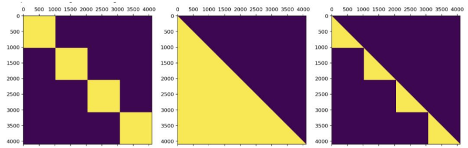


Applying Masks and Soft Cap to Attention Logits

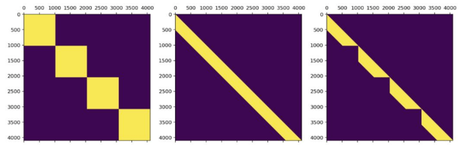
- **Compute masks**
  - **Pre-defined/Static Masks:** Applies explicit, boolean arrays (loaded beforehand) to mask out invalid connections, such as padding tokens or pre-set causal dependencies.
  - **Dynamic/Index Masks:** Computes masks on the fly using reconstructed global sequence indices (q_sequence and k_sequence) for tasks like implementing sliding window or local causal attention patterns.
  - **Segment ID Masks:** Enforces validity for **packed sequences** by ensuring that a query can only attend to key/value tokens that belong to the same logical segment (q_ids==kv_ids).
- **Apply cap logits**
  - logits = jnp.tanh(qk / attn_logits_soft_cap)
  - logits = logits * attn_logits_soft_cap
- **Apply Mask**
  - mask = functools.reduce(jnp.logical_and, masks)
  - qk = cap_logits(qk)
  - qk = jnp.where(mask, qk, mask_value)



segment mask [4096, 4096] with 4 segments,
causal mask, segment mask & causal mask



segment mask [4096, 4096] with 4 segments,
Local windows mask, segment mask & local_window_mask



Splash Attention Kernel - DeepSeekV3

Extracted Tensor Shapes **(bq = 2048, bkv = 2048)**

- **Query (q):** bf16[8,128,4096,192]
- **Key (k):** bf16[8,128,4096,192]
- **Value (v):** bf16[8,128,4096,128]
- **Output (o):** f32[8,2048,128]
- **Logsumexp:** f32[8,128,4096,128]

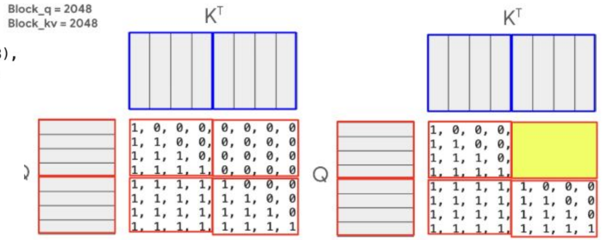
(128 query heads, batch size: 8, sequence_length: 4096, q_kv head: 192, v_dim: 128)

The custom call also includes several metadata and mask tensors with the following shapes:

- **block_mask:** s8[4] (only 4 blocks)
- **mask_next:** s8[4] (only 4 blocks)
- **partial_mask_blocks**

Block_q: 2048, Block_k=2048, block_kv_compute: 2048 results in 2 tiles on 4096 seq length


```
MaskInfo(mask_next=Array([0, 0, 0, 0], dtype=int8), active_rows=Array([0, 1, 1, 0], dtype=int8), active_cols=Array([0, 0, 1, 0], dtype=int8),
block_mask=Array([1, 2, 1, 0], dtype=int8), num_active_blocks=Array([3], dtype=int32), partial_mask_blocks=Array([[[1, 0, 0, ..., 0, 0, 0],
        [1, 1, 0, ..., 0, 0, 0],
        [1, 1, 1, ..., 0, 0, 0],
        ...,
        [1, 1, 1, ..., 1, 0, 0],
        [1, 1, 1, ..., 1, 1, 0],
        [1, 1, 1, ..., 1, 1, 1]]], dtype=int8), q_sequence=None)
```

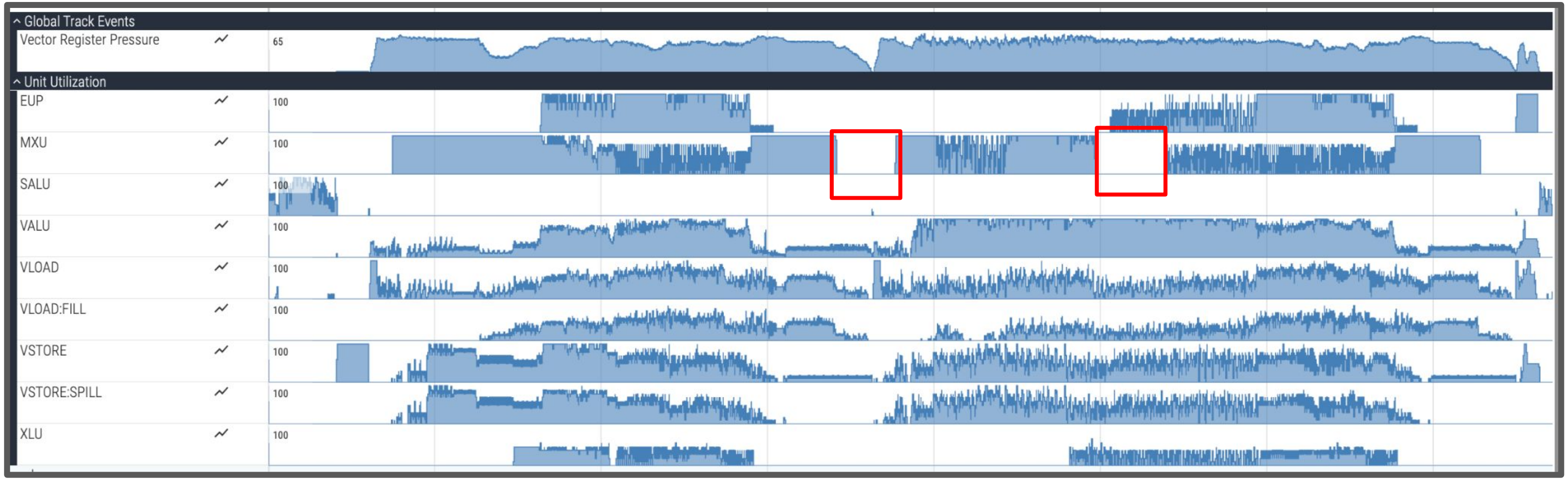


Splash Attention: Optimization Strategies Summary

Category	Optimization Technique	Concise Purpose
I/O & Tiling	Block-Wise Processing	Avoids materializing the massive attention matrix by breaking Q×KV into smaller, processable tiles (bq,bkv).
	Tuning & Micro-Tiling	Manually sets bq/bkv to the maximum size that fits in VMEM and subdivides blocks (bkv_compute) to fit intermediate results entirely into fast Vector Registers for zero-stall compute.
Sparsity	Sparse Execution Map	Pre-processes the mask once into book-keeping arrays (active_rows, mask_next) to create an execution roadmap, allowing the hardware to skip inactive blocks and save computation.
	Joint Masking	Combines all mask constraints (Causal, Dynamic, Segment IDs) using logical AND to precisely control attention boundaries and prevent cross-segment leakage.
Numerical Stability	Online Softmax & Delayed Norm	Calculates the Softmax iteratively across blocks by accumulating Max Logits (M) and LogSumExp (L) residuals. The final normalization (division) is deferred until the very end.
	Max Logit Estimate (MLE)	Replaces the expensive dynamic M calculation with a correct constant, bypassing slow vector reduction instructions to boost arithmetic throughput.
Hardware	Vector Alignment	Scalar accumulation buffers (M, L) are padded from (bq,) to (bq,128) to align data with the NUM_LANES wide memory bus, converting many slow scalar reads into one fast vector transaction.

Splash Attention Kernel - Pacchetto Trace

- Splash Attention kernel (seq_len: 4096, bq =1024, bkv =1024, and bkv_compute=1024)
- The MXU bar has several gaps and fails to maintain continuous saturation at 100%
- The kernel is failing to sustain its processing pipeline because **inconsistent VLOAD/VLOAD:FILL activity** and critical **VSTORE:SPILL** operations—caused by high **Vector Register Pressure**—prevent the Matrix Unit (MXU) from maintaining continuous, high-throughput compute



Optimization Exercises: Tiling, Arithmetic, and I/O

1. Experiment with **Optimal Tiling** by setting the memory block sizes to bq=2048 and bkv=2048 while enforcing **Micro-Tiling** within the loop by setting the compute chunk size to bkv_compute=512
2. Enable the **Arithmetic Simplification** by setting the max_logit_const (e.g., to 70) to bypass the expensive, dynamic vmax instruction in the kernel's inner loop, thereby boosting the VALU's arithmetic throughput.
3. Enable **Base-2 Exponentiation** by setting the use_base2_exp=True flag to substitute the native exp function with exp2, which often maps to a faster, specialized hardware instruction, improving the speed of the most frequently executed step in the kernel.
4. Enable **Sparsity Optimization**: To improve memory bandwidth, disable the transfer of the large partial_mask_blocks array and instead force the kernel to calculate the element-wise mask dynamically using position index logic, trading memory I/O for compute time.
5. **Scaling Validation:** Profile performance at **16K, 32K, 128K** sequence lengths.

Reading Material

- Earlier Work
  - Flash Attention Paper
  - Flash Attention - 2 Paper
  - Flash Attention - 3 Paper
- Tokamax
  - https://github.com/openxla/tokamax