https://goo.gle/2026-tpu-rfp

2026 Google Awards for Machine Learning Research and Education with TPUs

Overview

Google is requesting proposals from faculty members at academic degree-granting institutions for the 2026 academic year. Google's goal for these awards is to support the advancement of machine learning through both foundational and applied research, as well as the design and execution of courses that feature Google TPUs. Proposals are welcome for research across all domains - from core ML systems to applied sciences - and for courses at any level, including undergraduate or postgraduate.

https://goo.gle/2026-tpu-rfp

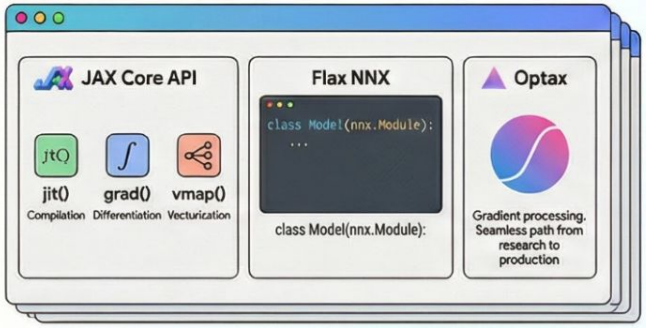
Decoding the JAX AI Stack
From Python Code to Silicon
JAX / XLA / TPU

Srinath Mandalapu
CoreML Frameworks - Google

Agenda

- **Introduction to JAX AI Stack**
  - GPT2 Training
- **TPU Architecture**
  - Memory Hierarchies (HBM/VMEM/SMEM)
  - Compute Units (MXU, VPU, XLU), VREGs,
  - DMAs (load/store)
  - Tiling, Layouts
- **XLA Compilation**
  - Lowering: Tracing, JAXpr, Stable HLO
  - Compile: HLO, LLO, VLIW Bundles, Executable
- **Sharding**
  - JIT (auto injection collectives) - Global View
  - Shard Map (Device Level Computation)

Decoding the JAX AI Stack: From Python Code to TPU Silicon



The Interoperable AI Stack

Built on JAX, Flax NNX, and Optax, providing a seamless path from research to production.

Core Philosophy: Functional Power

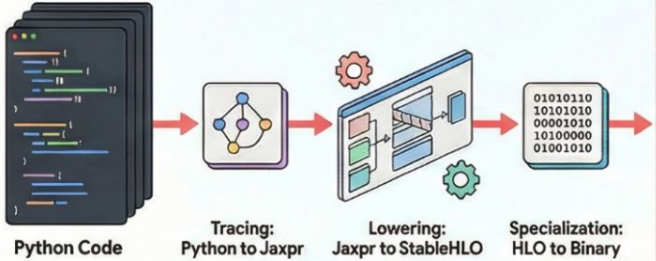
JAX achieves high performance through composable transformations: jit(), grad(), and vmap().

Flax NNX: The Pythonic Path

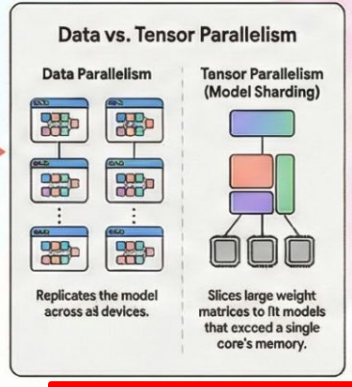
NNX uses standard Python object semantics and subclasses nnx.Module for intuitive state management and native JAX Pytree integration.

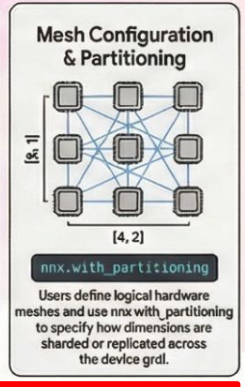
Phase 1: The Software Layer (JAX & Flax NNX)

Infographic - NotebookLM



Phase 2: The XLA Compilation Pipeline

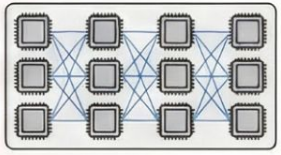




Phase 3: Distributed Execution & Sharding

Massive Scale: The Ironwood Superpod

A Superpod links 9.21e+ chips via an Optical Circuit Switching (OCS) networic providing 1.77PB of shared HBM capacity and 5.6Tb/s interconnect speeds.



Memory Hierarchy: HBM vs. VMEM

HBM

HBM Capacity: 192 GiB
HBM Bandwidth: 2,380 GB/s
Provides 100s of GIB for weights.

VMEM

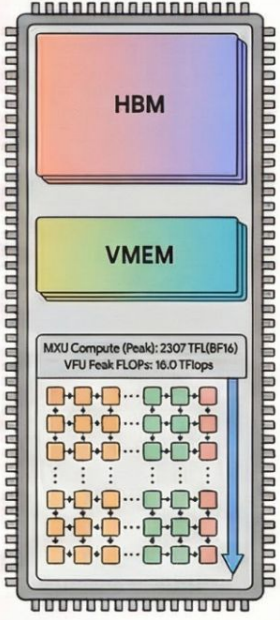
VMEM Bandwidth: O(10,000) GIB/s
Acts as a high-speed lane directly next to the execution units.

The Matrix Multiply Unit (MXU)

The heart of the TPU, uses a 256x256 systolis arrcy to perform 65,536 multiply-accumulate operations per cycle with weight-stationary date flow.

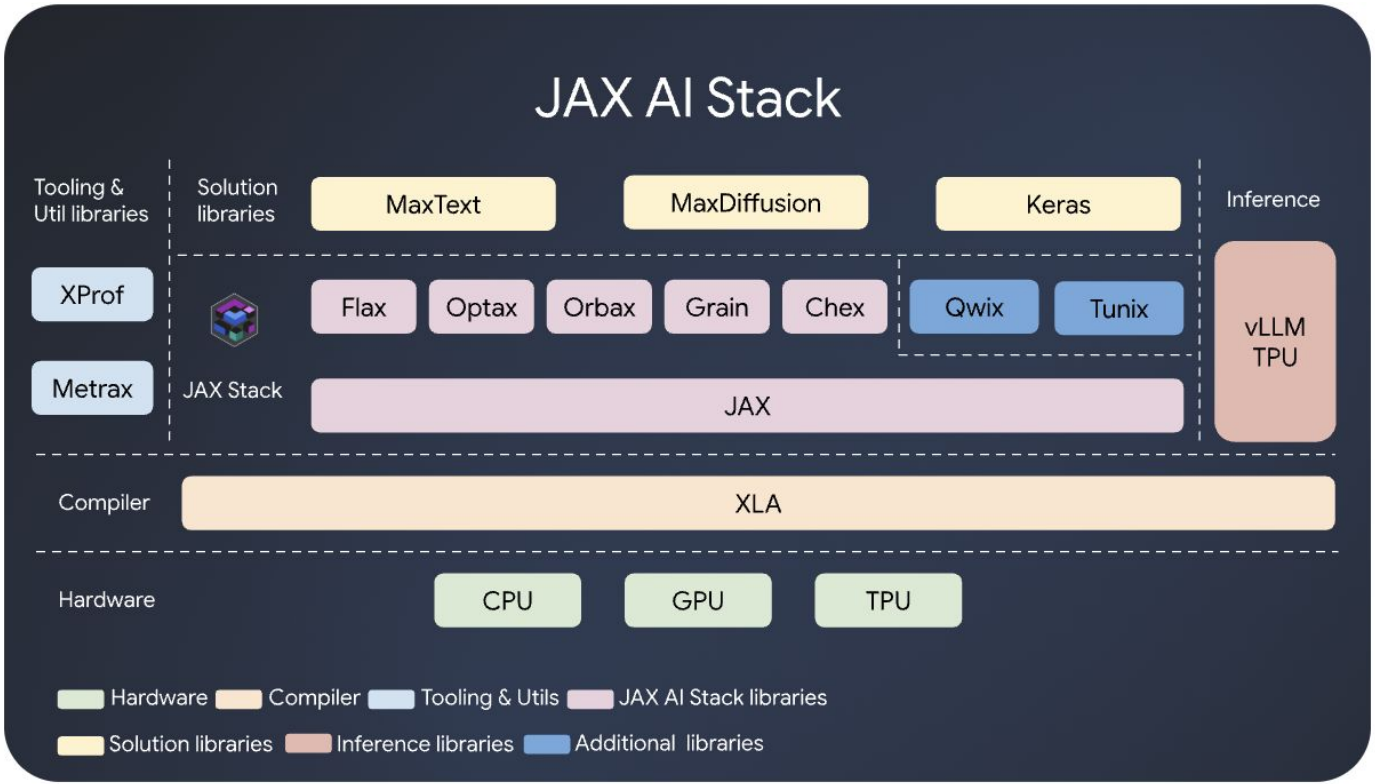
MXU Latency: The 3N-1 Rule

For an NxN matrix array, the time to complete a dense multiplication is exactly 2N-1 cycles, consisting of a FM phase and a Drain phase.



Phase 4: Inside the TPU Ironwood Hardware

JAX AI Stack



https://jaxstack.ai/

JAX Eco System - High Performance ML

Core Philosophy & Engine

- **Functional Power:** High performance via function transformations.
- **XLA Engine:** Generates hardware-optimized code using XLA compiler
- **Seamless Portability:** Uniform execution across CPU, GPU, and TPU.

The Interoperable Stack

- **JAX:** Provides the core NumPy API and function transformation layer.
- **Flax NNX & Optax:** Specialized libraries for Neural Networks and Optimization.
- **Orbax & Grain:** Handles checkpointing and high-efficiency data loading.

JAX: Accelerated Computing & Automatic Parallelism

Accelerated Numerical Computing

- JAX is a high-performance Python library built on NumPy syntax, designed for accelerators (GPUs/TPUs).
- Its core power comes from composable transformations
  - **jit()**, **grad()**, **vmap()**
- These transformations automatically compile, differentiate, and vectorize code for any numerical domain.

Seamless Scaling & Portability

- JAX is hardware-agnostic, running on CPUs, GPUs, and TPUs without modification (XLA compiler)
- Scaling from a single device to large TPU pods requires minimal boilerplate.
- Users define data sharding, and XLA manages all complex parallelism and communication.

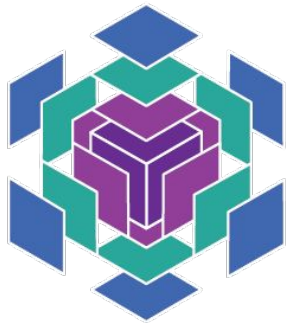
Flax NNX: The Pythonic Path to NN Development

Pythonic Object Semantics

- Flax NNX is a neural network library within the Flax ecosystem, specifically designed for JAX.
- Uses standard Python object semantics (classes, attributes, methods).
- Supports weight sharing via standard references.
- Modules defined by subclassing nnx.Module.

First-Class Pytree Integration

- Native JAX Pytrees for seamless ecosystem integration.
- Directly compatible with JAX transformations: jit, grad, vmap.
- Intuitive state management via attribute access.



Modular and Layered Design

- Layers defined in __init__, computation logic in __call__.
- Provides standard layers (nnx.Linear, nnx.Conv, etc.).

https://github.com/AI-Hypercomputer/maxtext/blob/main/src/MaxText/layers/gpt_oss.py

JAX, NNX & Optax - Simple Training Step

The Core Components - Simple Training Step

JAX: The Engine

Provides the accelerated NumPy (jnp), jax.random, and core function transformations like @jit that compile Python code for accelerators.

Flax NNX: The Blueprint

A Pythonic library to build models. nnx.Module creates stateful, object-oriented models that are native JAX Pytrees, simplifying state management.

Optax: The Optimizer

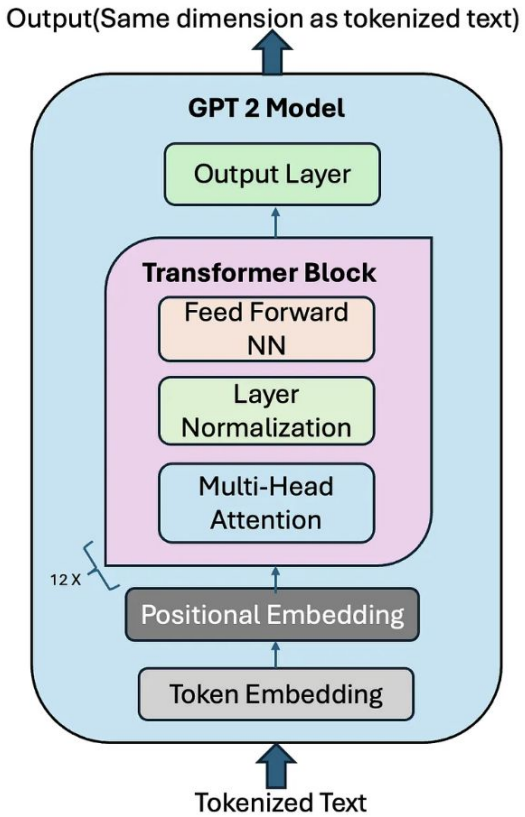
A dedicated library for gradient processing and optimization algorithms. nnx.Optimizer cleanly bundles the Optax state with the model's parameters.

```python
# Define a simple model
class LinearModel(nnx.Module):
    def __init__(self, *, rngs: nnx.Rngs):
        self.linear = nnx.Linear(in_features=1, out_features=1, rngs=rngs)

    def __call__(self, x: jax.Array):
        return self.linear(x)

# Instantiate the model
key = jax.random.PRNGKey(0)
model = LinearModel(rngs=nnx.Rngs(key))

# Create an Optax optimizer
tx = optax.sgd(learning_rate=0.01)
optimizer = nnx.Optimizer(model, tx=tx, wrt=nnx.Param)

# Dummy data
x = jnp.array([[2.0]])
y = jnp.array([[4.0]])

# Training step function
@nnx.jit
def train_step(model, optimizer, x, y):
    def loss_fn(model):
        y_pred = model(x)
        return jnp.mean((y_pred - y) ** 2)

    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)
    return loss, model

# Training loop
num_steps = 10
for i in range(num_steps):
    loss, model = train_step(model, optimizer, x, y)
    print(f"Step {i+1}, Loss: {loss}")

print("Trained model output:", model(x))
```

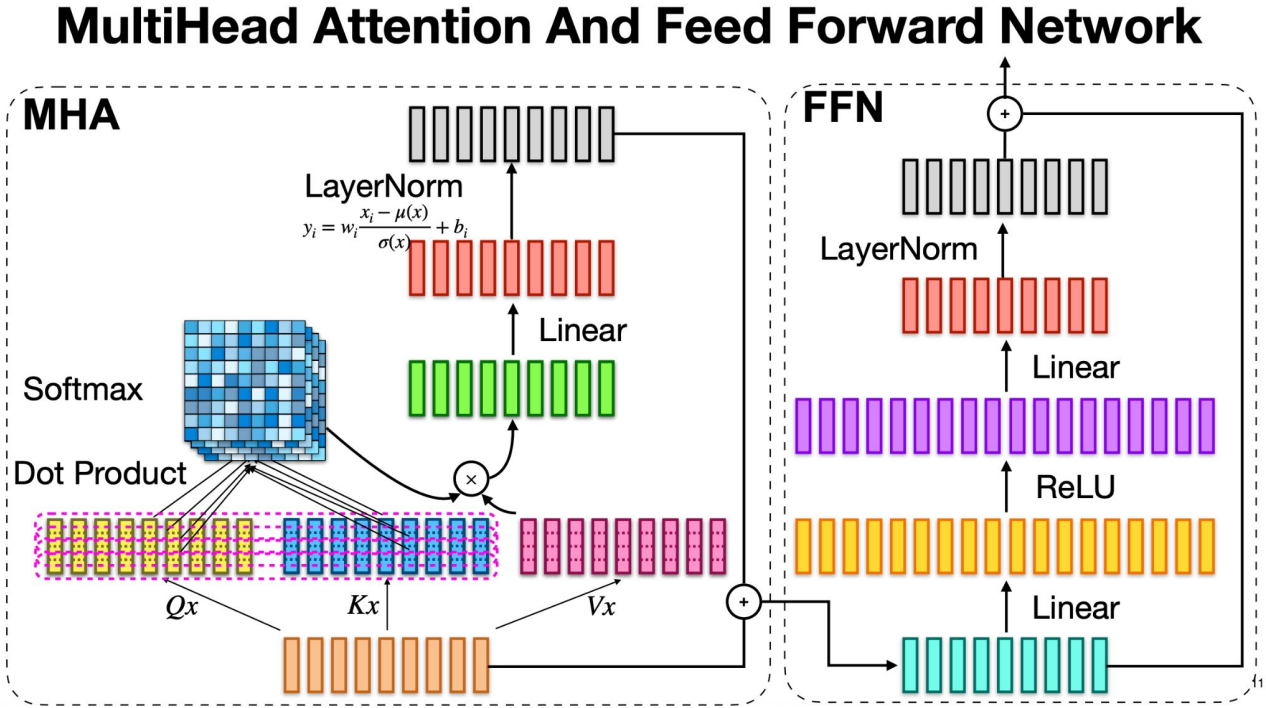
Build Transformer with JAX

- Build a Transformer with JAX
- GPT2_transformers_workshop_I...
- The Illustrated GPT-2 (Visualizing Transformer Language Models) – Jay Alammar
- Pytorch GPT2 implementation

```
num_transformer_blocks = 24
seqlen = 1024
embed_dim = 1024
num_heads = 16
feed_forward_dim = 4 * embed_dim

batch_size = 32
dropout_rate = 0.1
max_steps = 600000*12//batch_size
init_learning_rate = 5e-4
weight_decay = 1e-1
```



Transformer Block Review (Previous Lecture)



GPT 2 Training Configuration

num_transformer_blocks = 24

seqlen = 1024

embed_dim = 1024

num_heads = 16

feed_forward_dim = 4 * embed_dim

batch_size = 32 (local = 32/8 = 4)

dropout_rate = 0.1

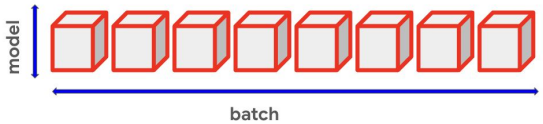
init_learning_rate = 5e-4

weight_decay = 1e-1

https://llmsystem.github.io/llmsystem2026spring/assets/files/llmsys-10-transformer-acc-5ba466406bf7296f86cd244ad0405867.pdf

Distributed Training - Data Parallelism (8-Way)

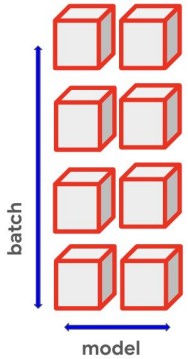
Mesh Configuration: (8, 1) → ('batch', 'model')




```
mesh = Mesh(mesh_utils.create_device_mesh((8, 1)), ('batch', 'model'))
```

- **Data Sharding:** The global batch (e.g., 32 samples) is split into 8 shards. Each TPU device receives **4 unique samples**.
- **Parameter Replication:** Because the model axis has a size of 1, the entire model state is **replicated** (mirrored) across all 8 devices.
- **Gradient Aggregation:** During the backward pass, @nnx.jit automatically inserts an **All-Reduce** operation. Gradients from all 8 devices are summed and averaged before the optimizer update to ensure all replicas stay synchronized.

Distributed Training - (Tensor + Data Parallelism)

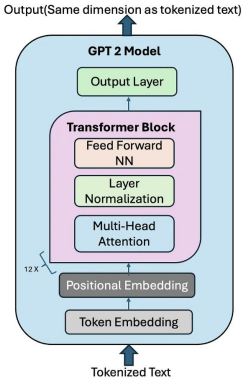


Mesh Configuration: (4, 2) → ('batch', 'model')

mesh = Mesh(mesh_utils.**create_device_mesh**((8, 1)), ('batch', 'model'))

- **Model Sharding (Tensor Parallel):** The model axis size is now 2. Large weight matrices (like MLP kernels) are split across **2 devices**. This allows for training larger models that exceed the memory of a single TPU core.
- **Data Sharding:** The batch is split 4 ways along the batch axis.
- **The Replica Group:** Each unique data shard is processed by a group of 2 devices. Within that group, the model is sharded; across the 4 groups, the data is different.
- **Communication: Intra-group:** Devices exchange partial activations (All-Gather/Reduce-Scatter) to compute layer outputs.
    - **Inter-group:** Gradients are aggregated across the 4 data-parallel replicas.

High Level GPT Architecture



Embedding Layer (TokenAndPositionEmbedding(nnx.Module))

- Combines learned **Token Embeddings** and **Position Embeddings** (sequence order).
- Input tokens are mapped to vectors and summed: x = Embed_token + Embed_pos

Transformer Block (TransformerBlock(nnx.Module))

- Normalization is applied *before* the sub-layers for improved training stability.
- **Masked Multi-Head Attention:** Enables the model to focus on previous tokens while masking future ones (causal).
- **Feed-Forward Network (MLP):** Two linear layers with **GELU** activation for non-linear processing.
- x = x + Attention(LN}(x)) followed by x = x + MLP(LN}(x)).

GPT Model Container (GPT2(nnx.Module))

- Stacks N identical Transformer Blocks (e.g., 24 blocks).
- A final LayerNorm followed by a linear projection to the vocabulary size.
- Reuses the input **Token Embedding** weights for the final output projection, reducing total parameter count and improving regularization.

Parameter Sharding with nnx.with_partitioning

nnx.with_partitioning wraps standard initializers to bind a **PartitionSpec (P)** to a variable at the moment of creation. This defines how a tensor's dimensions are distributed across the named axes of a hardware **Mesh**.

- **None**: Dimension is **replicated** across all devices on that axis.
- **'model'**: Dimension is **sharded** (split) across the devices on that axis.

Example: Weight Matrix [1024, 512]

- **Configuration:** kernel_init = **nnx.with_partitioning(init_fn, P(None, 'model'))**
- **Logical Shape:** 1024 (Rows) x 512 (Columns)

Scenario: batch: 8, model: 1 | Weight Shape: [1024, 512]

- **Full Replication:** Mapping the 512-column dimension to a size-1 'model' axis ensures every TPU device stores the entire 1024x512 matrix
- **Automatic Sync:** JAX detects replication across the 8-device 'batch' axis and triggers an **All-Reduce** during the backward pass to synchronize gradients.

Scenario: batch: 4, model: 2 | Weight Shape: [1024, 512]

- **Tensor Parallelism:** Mapping the 512-column dimension to a size-2 'model' axis physically slices the weight; each device stores only **256 columns**.
- **Hybrid Communication:** Devices perform **All-Gather** across the 'model' axis to reconstruct activations and **All-Reduce** across the 'batch' axis to synchronize gradients.

MHA Parameter Sharding ([8, 1] vs. [4, 2] Mesh)

Configuration:

- d_model = 1024 | Heads = 16 | d_head = 64

Layer	Weight Shape	Bias Shape
Q,K,V Projections	[1024, 16, 64]	[16, 64]
Output Projection	[16, 64, 1024]	[1024]

```
devices = mesh_utils.create_device_mesh((8, 1))
mesh = Mesh(devices, axis_names=('batch', 'model'))
with jax.set_mesh(mesh):
  model = GPT2(rngs=nnx.Rngs(0))
```

```
self.mha = nnx.MultiHeadAttention(
        num_heads=num_heads,
        in_features=embed_dim,
        kernel_init=nnx.with_partitioning(
          nnx.initializers.xavier_uniform(),
          P(None, 'model')),
        bias_init=nnx.with_partitioning(
          nnx.initializers.zeros_init(),
          P('model')),
      ...
)
```

Feature	Scenario A: Mesh (8, 1)	Scenario B: Mesh (4, 2)
Strategy	Pure Data Parallelism	Hybrid Tensor/Data Parallelism
Sharding	model' axis size = 1 ⟶ Full Replication	model' axis size = 2 ⟶ Tensor Parallelism
Storage	Every TPU device stores all 16 heads.	Each TPU device stores only 8 heads.
Communication	All-Reduce across 8-device 'batch' axis to sync gradients.	All-Gather across 'model' axis (combine outputs) + All-Reduce across 4 'batch' replicas.

JIT Optimized Training Step

@nnx.jit Compilation: The entire **train_step** is compiled into a single XLA graph, fusing the forward pass, backward pass, and optimizer update into a high-performance TPU execution unit.

Functional Gradients: Uses **nnx.value_and_grad** to capture the model's state, compute the next-token prediction loss, and generate the gradient Pytree in one unified call.

Integrated Communication: Because parameters are tagged with **NamedSharding**, JIT automatically injects **All-Reduce** operations to aggregate gradients across the 8-device mesh.

In-Place Updates: The **optimizer.update(grads)** call applies AdamW weight decay and momentum adjustments, directly mutating the model's sharded state on the TPU HBM.

```
@nnx.jit
def loss_fn(model, batch):
  logits = model(batch[0])
  loss = optax.softmax_cross_entropy_with_integer_labels(
    logits=logits,
    labels=batch[1]).mean()
  return loss, logits

@nnx.jit
def train_step(model: nnx.Module,
          optimizer: nnx.Optimizer,
          metrics: nnx.MultiMetric, batch):
  grad_fn = nnx.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(model, batch)
  metrics.update(loss=loss, logits=logits, lables=batch[1])
  optimizer.update(grads)
```

Training Loop

Sharded Data Fetching: Batches are fetched and distributed across the 8-device mesh using jax.device_put. The **NamedSharding(mesh, P('batch', None))** ensures each TPU device receives a unique slice of the 32-sample batch.

Execution Trigger: The **train_step** is invoked in each iteration, executing the JIT-compiled forward/backward passes and updating the replicated or sharded model weights in-place.

Periodic Metric Logging: Every 200 steps, the loop pulls the accumulated **train_metrics** to record training loss, providing real-time visibility into the convergence.

Evaluation Phase: A separate **val_data** batch is processed through the **loss_fn** (without gradient updates). This "Eval Step" monitors the model's generalization on unseen tokens and prevents overfitting.

```
while True:
 input_batch, target_batch = get_batch("train")
 train_step(model,
        optimizer, train_metrics,
        jax.device_put(
          (input_batch, target_batch),
           NamedSharding(mesh, P('batch', None)))
        )

 if step % 200 == 0:
   # Train metrics
   train_loss = float(train_metrics.compute())
   metrics_history['train_loss'].append(train_loss)
   # Eval step
   input_val_batch, target_val_batch = get_batch('val')
   loss, logits = loss_fn(
     model,
     jax.device_put(
       (input_val_batch, target_val_batch),
       NamedSharding(mesh, P('batch', None)))
     )
   val_metrics.update(val_loss=loss, logits=logits)
   val_loss = float(val_metrics.compute())
```

Checkpoint Management with Orbax

State Capture: nnx.state(model) extracts the model parameters and metadata as a Pytree, preserving the specific **NamedSharding** rules used during training.

Orbax Saving: The **PyTreeCheckpointer** asynchronously writes sharded TPU memory to disk, allowing the model to be saved without significantly stalling the training loop.

Abstract Restoration: nnx.eval_shape creates a memory-efficient "blueprint" of the model, defining the expected state structure before physical memory is allocated.

State Loading: checkpointer.restore maps the saved weights back into the state Pytree, maintaining compatibility across different TPU mesh configurations.

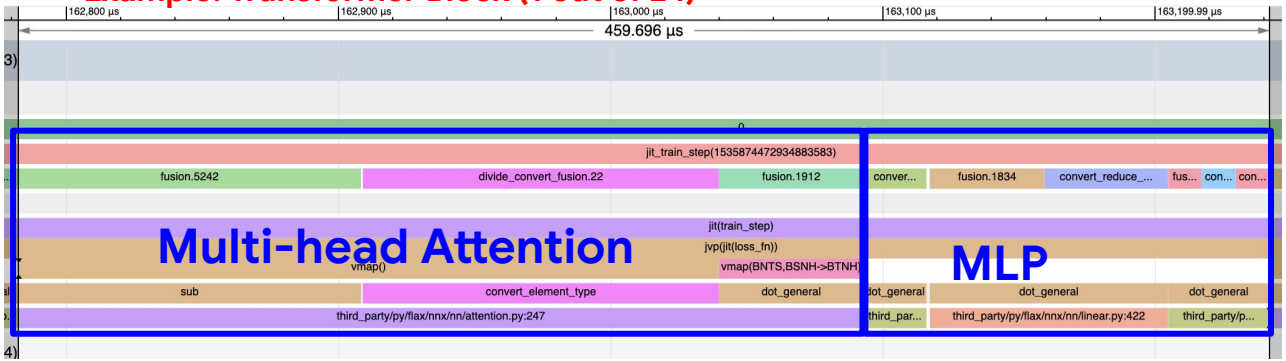
Model Update: nnx.update(model, state) populates the model instance with the restored weights, enabling immediate resumption of training or text generation.

```
checkpointer = orbax.PyTreeCheckpointer()
train_state = nnx.state(model)
checkpointer.save(checkpoint_path, train_state)
```

```
model = nnx.eval_shape(lambda:
create_model(rngs=nnx.Rngs(0)))
state = nnx.state(model)
checkpointer = orbax.PyTreeCheckpointer()
state = checkpointer.restore(checkpoint_path, item=state)
nnx.update(model, state)
```

XPROF - Training Step on one device

Example: Transformer Block (1 out of 24)




```python
def __call__(self, inputs, training: bool = False):
    input_shape = inputs.shape
    bs, seq_len, emb_sz = input_shape

    with jax.named_scope("mha_attention"):
      attention_output = self.mha(
        inputs_q=self.layer_norm1(inputs),
        mask=causal_attention_mask(seq_len),
        decode=False,
      )
    x = inputs + self.dropout1(attention_output, deterministic=not training)

    # MLP
    with jax.named_scope("mlp"):
      mlp_output = self.linear1(self.layer_norm2(x))
      mlp_output = nnx.gelu(mlp_output)
      mlp_output = self.linear2(mlp_output)
      mlp_output = self.dropout2(mlp_output, deterministic=not training)

    return x + mlp_output
```

MHA: Compiling Logic into Fused Kernels

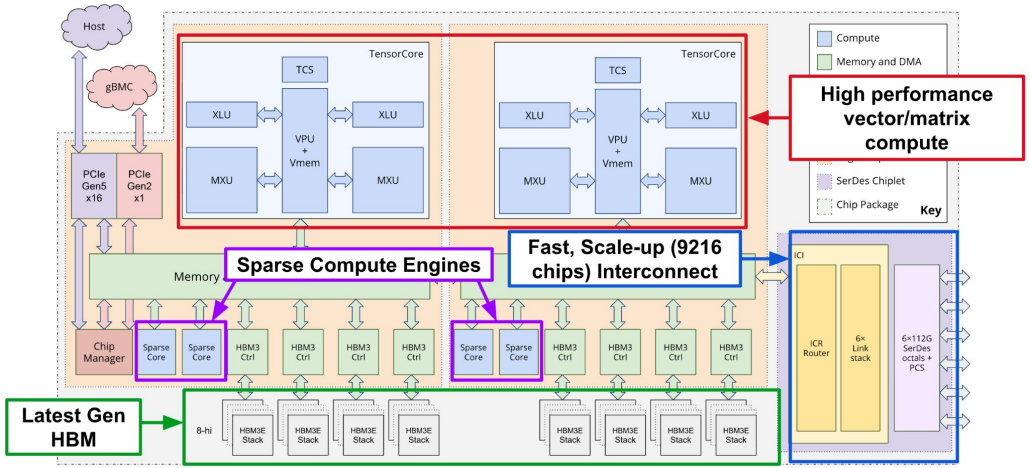
Batch Size = 4, Seq Length = 1024, Attention Heads: 16, Head Dim:64

Fusion Name	Inputs	Outputs	Python / Logical Operations	Key Notes
%convolution_add_fusion.44/45	Hidden States (X), Weights (Wq,k), Biases (bq,k)	Query (Q) or Key (K) tensors: bf16[4, 1024, 16, 64]	Q = X @ W_q + b_q K = X @ W_k + b_k	Lowers Linear layers to "Convolution" for hardware efficiency. Fuses bias addition.
%fusion.5242	Causal Mask, Query (Q), Key (K)	1. Row-wise Max: f32[4, 16, 1024] 2. Raw Scores: f32[4, 16, 1024, 1024]	Scores = (Q @ K.T) Masked = Scores + Mask RowMax = max(Masked)	Output is f32 for Softmax stability. Computes row-wise maximums to enable the "Stable Softmax" trick.
%divide_convert_fusion.22	Raw Scores, Row-wise Max values	Normalized Weights: bf16[4, 16, 1024, 1024]	Scaled = Scores / sqrt(head_dim) Weights = exp(Scaled - Max) / Sum	Completes the Scaled Dot-Product Attention. Downcasts result to bf16 to save memory for the next stage.
%fusion.1912	Attention Weights, Value (V), Value Bias (bv)	Weighted Values: bf16[4, 16, 64, 1024]	Out = Weights @ V + b_v	Performs the weighted sum of Value vectors. Fuses the V projection bias directly into the matrix multiplication.

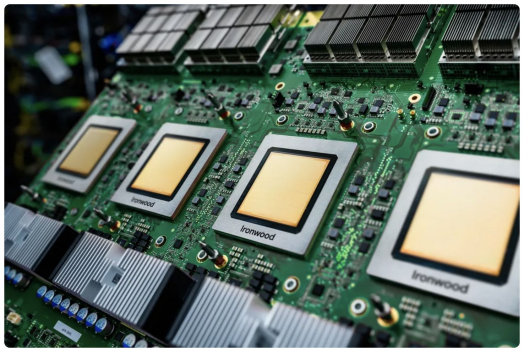
From Functional Abstractions to Fused Kernels: How XLA lowers high-level NNX/Python logic into a static HLO graph, fusing multiple attention operations into optimized kernels

TPU Ironwood

TPU Ironwood Architecture



- Each Ironwood TPU chip contains **two Tensor Cores** and **four SparseCores.**
- Each host node manages a single tray containing **4 Ironwood chips.**



Scale: 9216 chips per pod.
Peak Compute (per chip): 2307 TFLOPs (BF16) and **4614** TFLOPs (FP8).
Memory (per chip): 192 GiB HBM capacity with **7380 GBps** bandwidth.
Inter-Chip Interconnect (ICI): 1200 GB(Bytes)per sec bidirectional bandwidth per chip.
Data Center Network (DCN): 100 Gb(bits)per sec bandwidth per chip

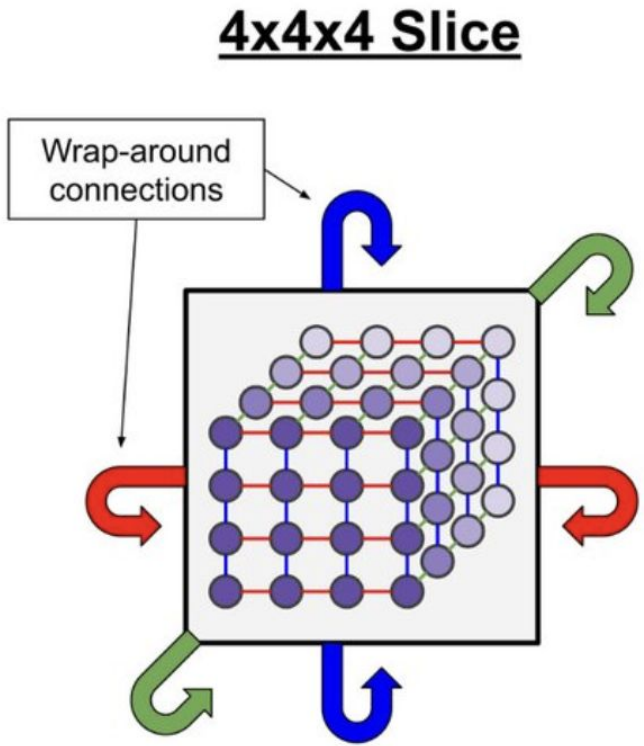
Scaling Ironwood: From Cubes to Superpods

The Fundamental Unit: 3D Torus Cube

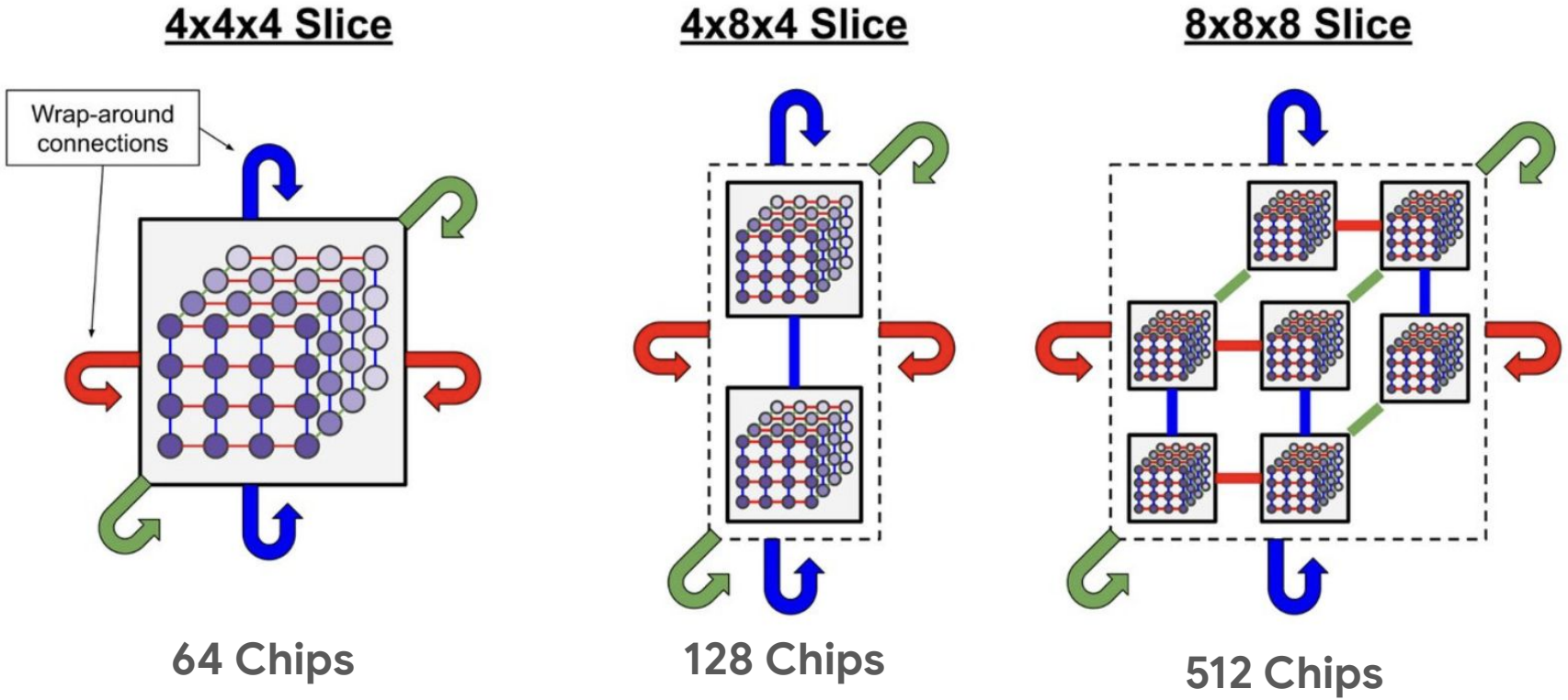
- **Structure:** A "cube" is a single rack containing 64 Ironwood chips (16 hosts).
- **Connectivity:** Each chip connects to 6 neighbors via high-speed **ICI** (Inter-Chip Interconnect).
- **Topology:** These links form a **3D Torus**, providing three distinct axes for massive parallel bandwidth.

The Scale: Optical Circuit Switching (OCS)

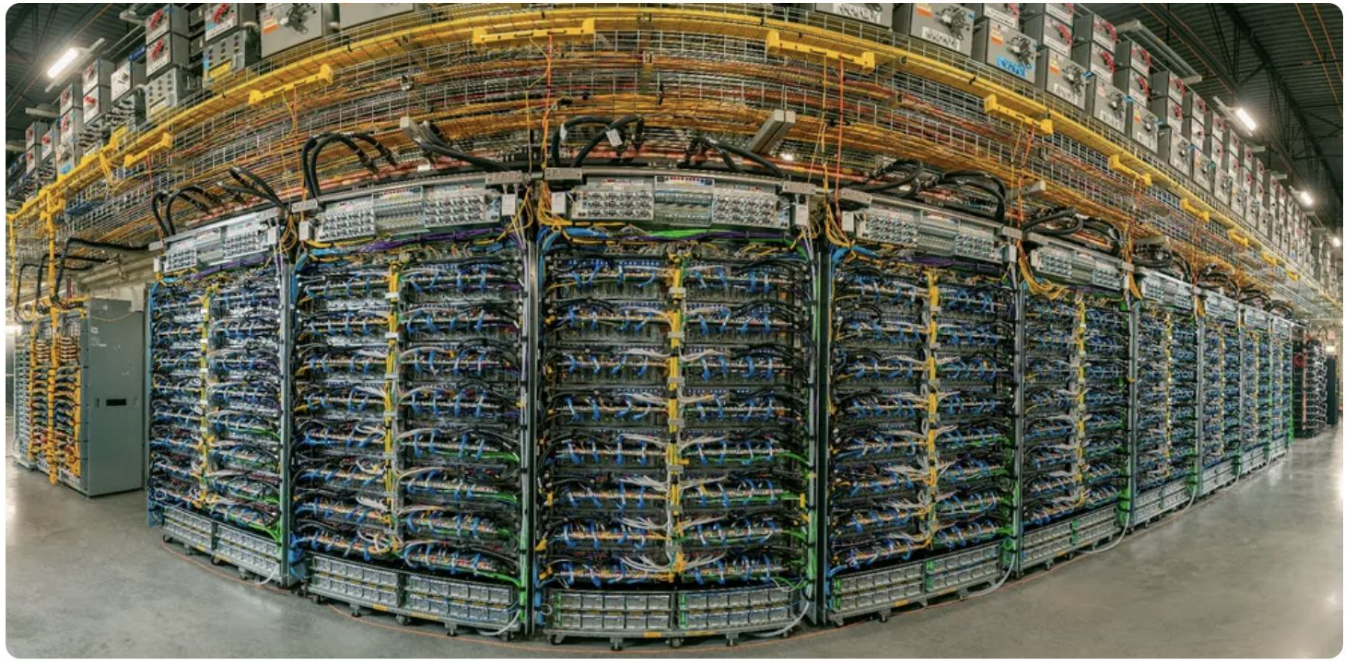
- **Modular Scaling:** Multiple cubes are linked by an **OCS network** to build "Pods" (256 chips) or "Superpods" (9,216+ chips).
- **Inter-Cube Resilience:** If a cube or link fails, the OCS optically bypasses the unhealthy unit and swaps in a spare.
- **Dynamic Slicing:** The reconfigurable OCS allows for precise, logical "slices" (e.g., 4x4x4 or 8x8x8) without a total system shutdown.



Flexible Scaling: Ironwood Topology Slices



Ironwood Superpod



1.77PB shared HBM capacity (9216 x 192 GiB/chip)

9.6Tb/s ICI network speed.

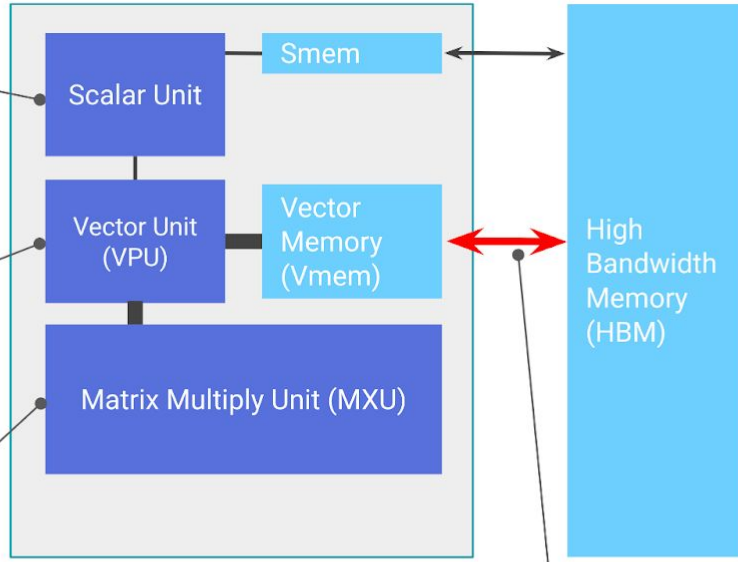
Multiple cubes are linked by an OCS network to build "Pods" (256 chips) or "Superpods" (9,216+ chips).

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.

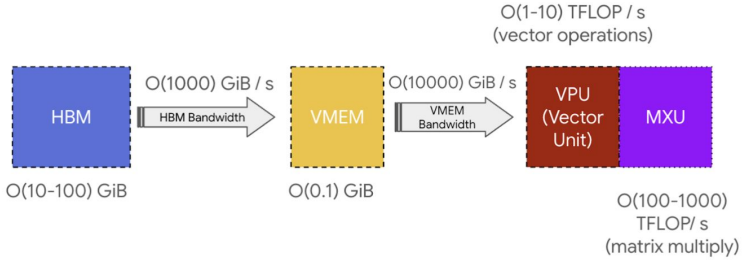


Abstract layout of a TPU TensorCore.

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).

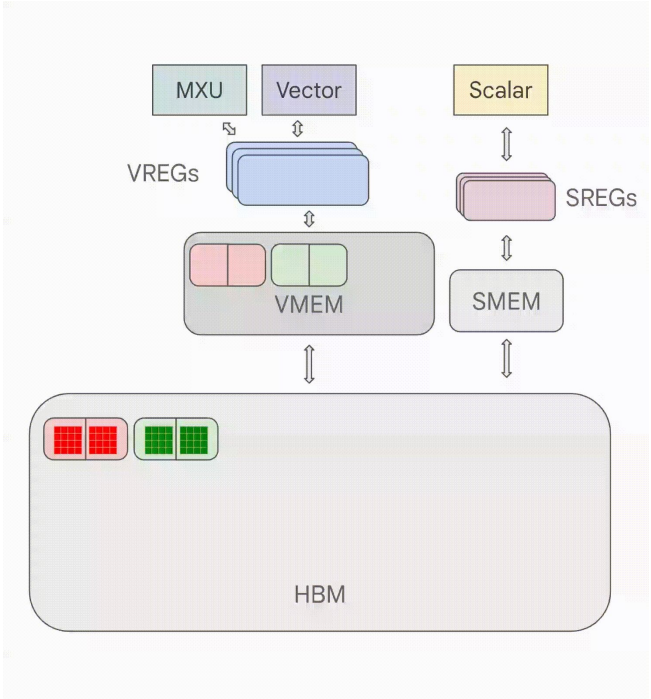
The VMEM Speed Advantage: **VMEM** provides significantly higher bandwidth than **HBM**, acting as a high-speed "lane" directly next to the execution units.

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.

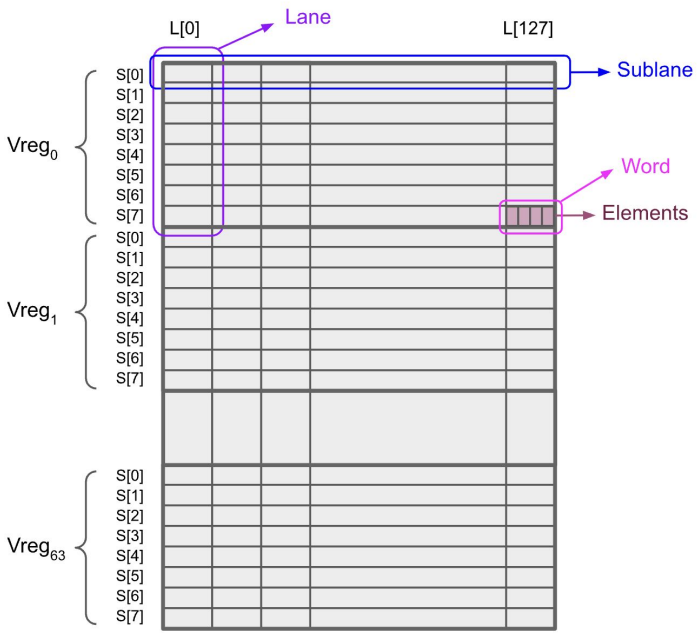
Pipelining in TPU

- **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.



TensorCore - Vector Register File (VRF)

- **Grid Structure**: Organized as **64 Vregs × 8 sublanes × 128 lanes of 32-bit words.**
- **The Vreg (Chunk)**: A 8x128 array that acts as the atomic unit for all data processing.
- **Word & Element**: Each 32-bit Word contains one or more Elements of the same data type.
- **Lanes & Sublanes**: **128 vertical Lanes and 8 horizontal Sublanes** per Vreg mirror the physical VPU layout.
- **Logical Rows**: While there are 8 physical sublanes, packing multiple elements into a word can increase the effective row count.



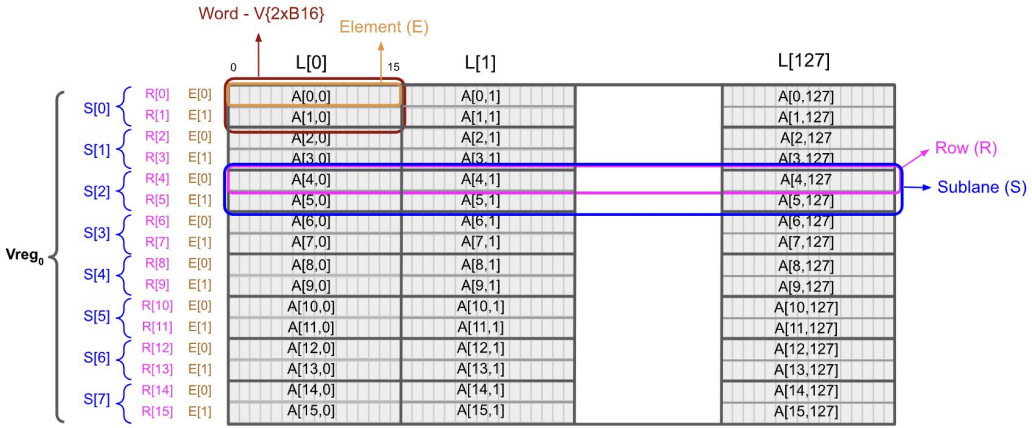
Compressed Packing and Row Scaling

Compressed Packing

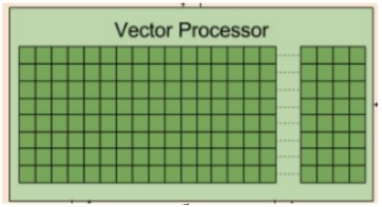
- Data elements from adjacent sublanes are squeezed sequentially into a single 32-bit Word to increase data density.

Dynamic Row Scaling

- This format causes instructions to act as if there are more rows per Vreg, calculating the total as 8 times the number of elements per Word.



The VPU: 2D Vector Architecture



- **2D SIMD Grid Execution**
  - The VPU operates as a 8x128 grid of 1024 sites that execute the same program in lock-step, ensuring massive parallel throughput.
- **The 4-ALU Multiplier**
  - Each site contains four independent floating-point ALUs, allowing the TPU to execute four distinct operations (vadd/vsub etc) simultaneously for a total of 4096 f32 operations per cycle.
- **Atomic Data Processing**
  - All instructions consume and produce **VREGs (Chunks)**, which are 8x128 arrays of 32-bit words that serve as the fundamental unit of data flow.
- **Communication Constraints**
  - While sublanes within a single lane can communicate via rotates and permutations, moving data across different lanes requires utilizing the **XLU**.

```
{ # In one clock cycle, the VPU can execute a block of instructions like this
    v2 = vadd.8x128.f32 v0, v1  → // ALU 0: Adds VREG 0 and 1, stores in VREG 2
    v5 = vsub.8x128.f32 v3, v4  →  // ALU 1: Subtracts VREG 4 from 3, stores in VREG 5
    v8 = vmul.8x128.f32 v6, v7  → //ALU 2: Multiplies VREG 6 and 7, stores in VREG 8
    v11 = vmax.8x128.f32 v9, v10 → // ALU 3: Finds max of VREG 9 and 10, stores in VREG 11
}
```

VPU Peak FLOPs → 1024 sites x 4 ALUs x 2 FLOPs/cycle x 2.2 GHz = **18.0 TFlops**

Vector Register Reductions – xlane vs. slane

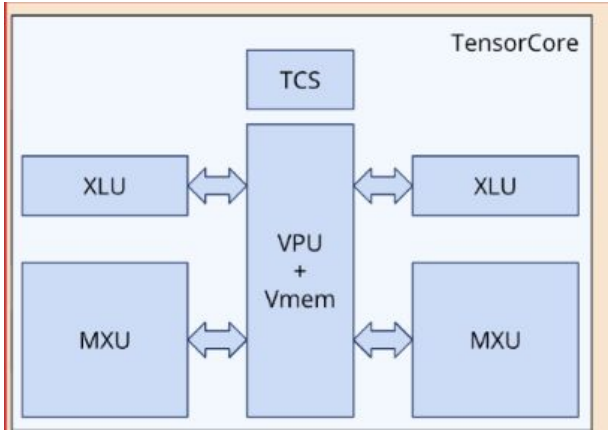
Sublane Reduction (slane): Aggregates values vertically within a single lane across all 8 sublanes, placing the result (e.g., 8.0f) in the 0th sublane while others remain "garbage".

Cross-Lane Reduction (xlane): Aggregates values horizontally within a single sublane across all 128 lanes, resulting in a consistent sum (e.g., 128.0f) across the SIMD width.

Softmax Utility: These primitives are combined in the LLO pipeline to perform row-wise operations, such as finding the maximum value or calculating the sum of exponentials across the vector registers.

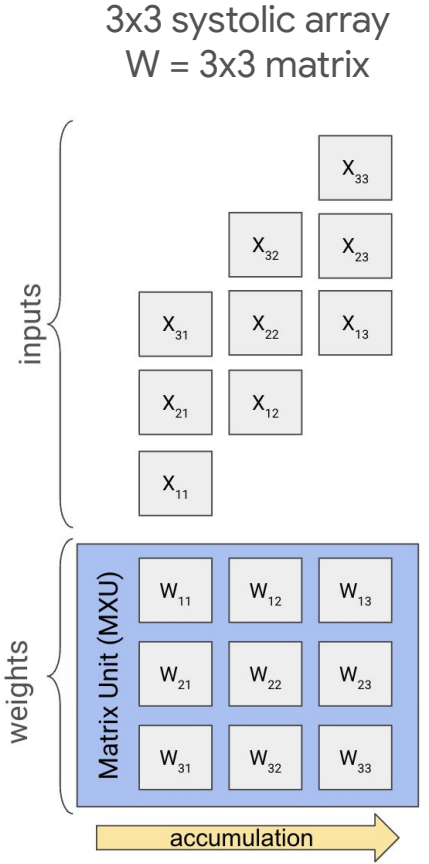
XLU (Cross-Lane Unit)

- **Lane Communication:** It is the dedicated hardware unit for moving data across the TPU's 128 parallel SIMD lanes, enabling communication that the standard Vector Processing Unit (VPU) cannot reach directly.
- **Horizontal Reductions**: Uses a Reducer Unit (RU) to find a single Max, Min, or Sum across lanes by "folding" data through a binary tree.
- **Data Reshaping:** Handles Transposes (flipping NxM ⟶ MxN) and Permutations (shuffling lanes) without using the MXU.
- **High-Speed Broadcasts:** Uses a 128x128 crossbar (a massive hardware switch) to "spray" a single value from one lane to all others instantly.
- **TPU Ironwood Dual Units:** Each tensor core has two units, identified as **XLU 0** and **XLU 1** in instruction traces.

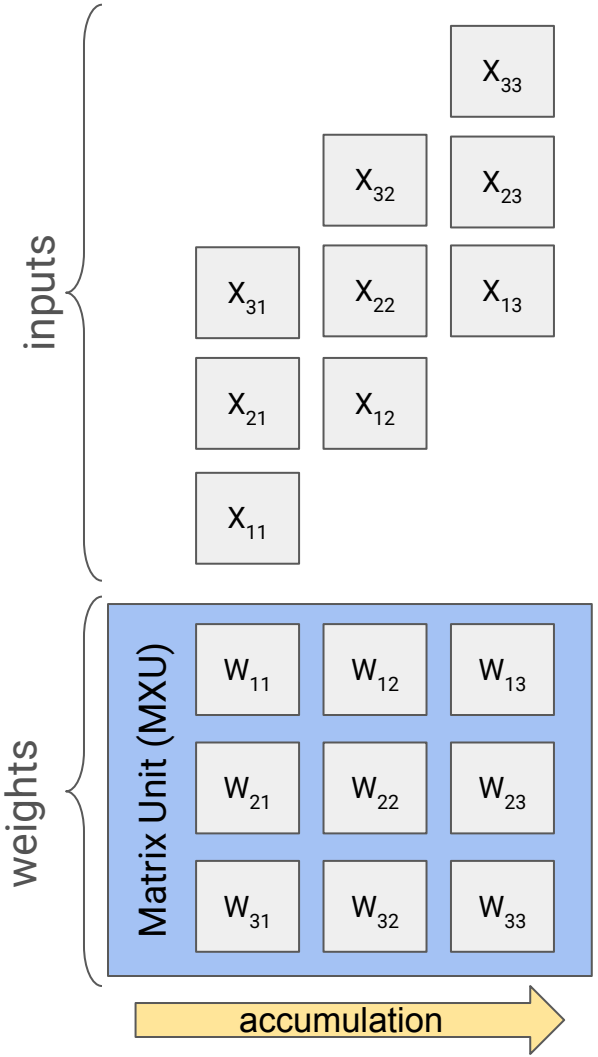


MXU (Matrix Multiply Unit)

- **Grid Scale**: Composed of a 2D grid of **256 x 256** processing elements (PEs), totaling **65,536 Multiply-Accumulate (MAC) units**.
- **Precision:** Inputs are processed in **bfloat16**, while all accumulations are performed in **FP32** to maintain high numerical stability.
- **Physical Connectivity:** Intermediate results pass directly between adjacent MAC units via short wires, avoiding the energy cost of round-tripping to memory.
- **Weight-Stationary Flow:** Weights from matrix B are preloaded into the grid. Activations (A) **flow vertically from top to bottom**, entering the grid in a staggered diagonal pattern.
- **Horizontal Accumulation:** Partial sums **flow horizontally across the rows**, accumulating results as they move through the PEs from one side to the other.
- **Massive Data Reuse:** Each activation is reused 256 times horizontally, and each partial sum accumulates 256 multiplications vertically, maximizing efficiency.



Matrix Unit Systolic Array



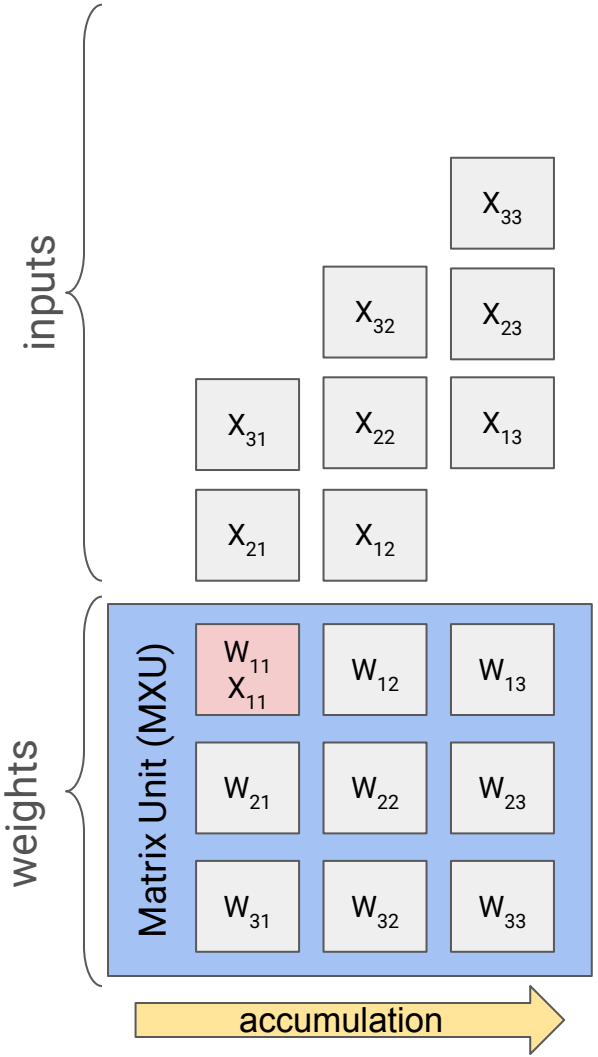
Computing y = Wx

3x3 systolic array
W = 3x3 matrix
Batch-size(x) = 3

Matrix Unit Systolic Array

Computing y = Wx

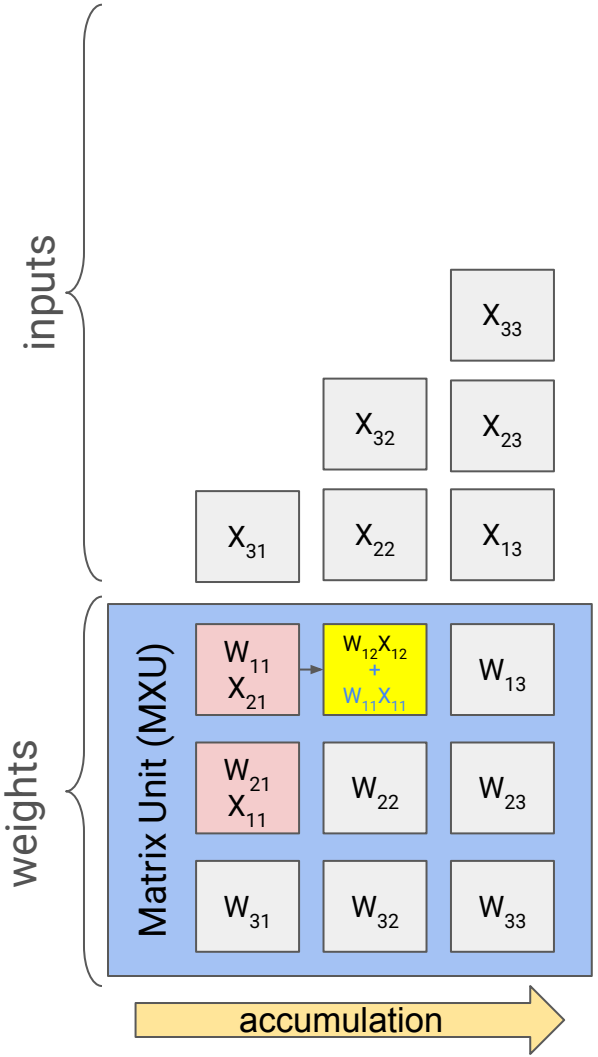
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

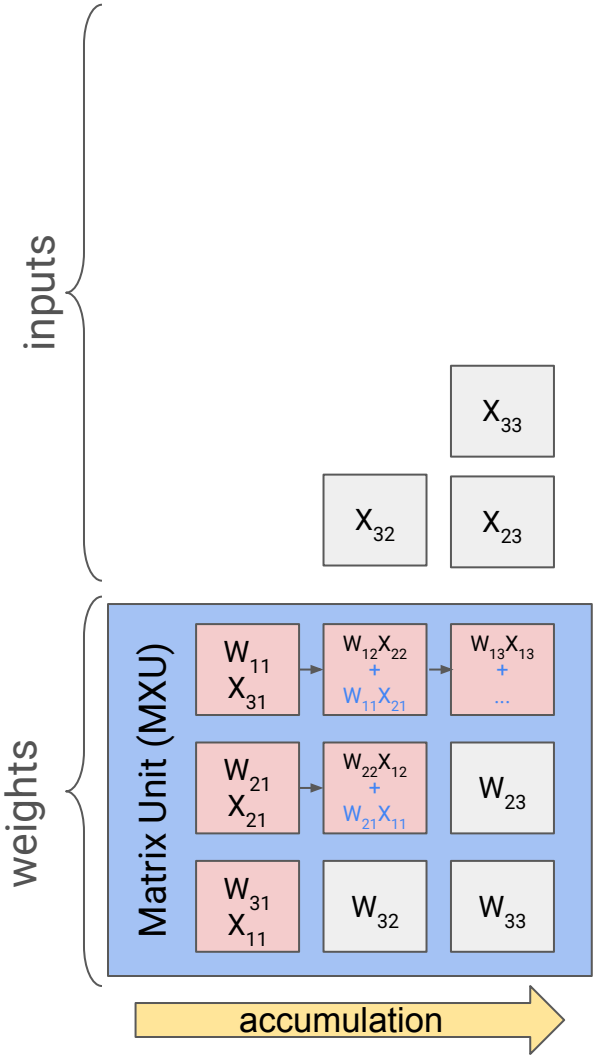
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

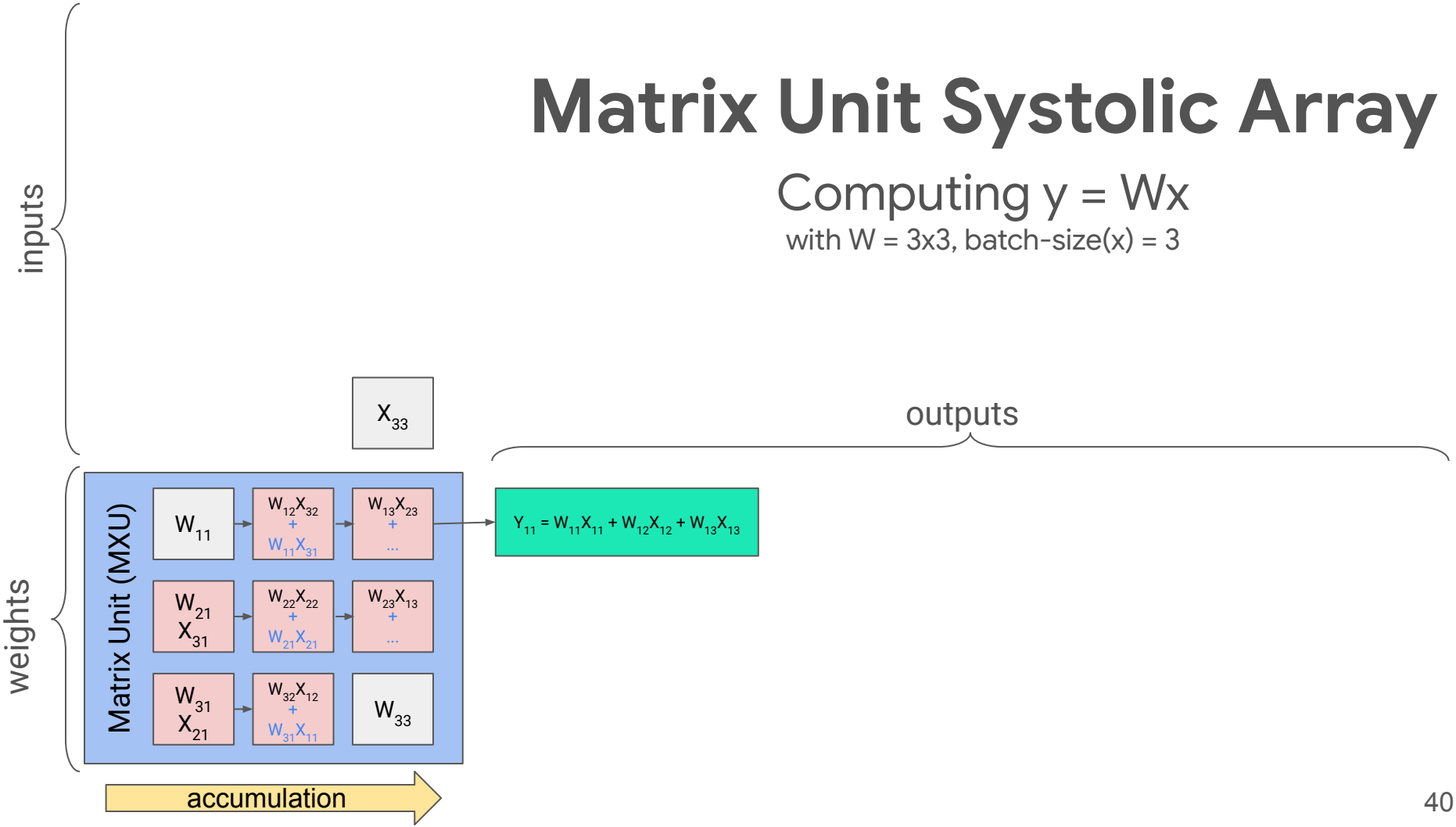
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

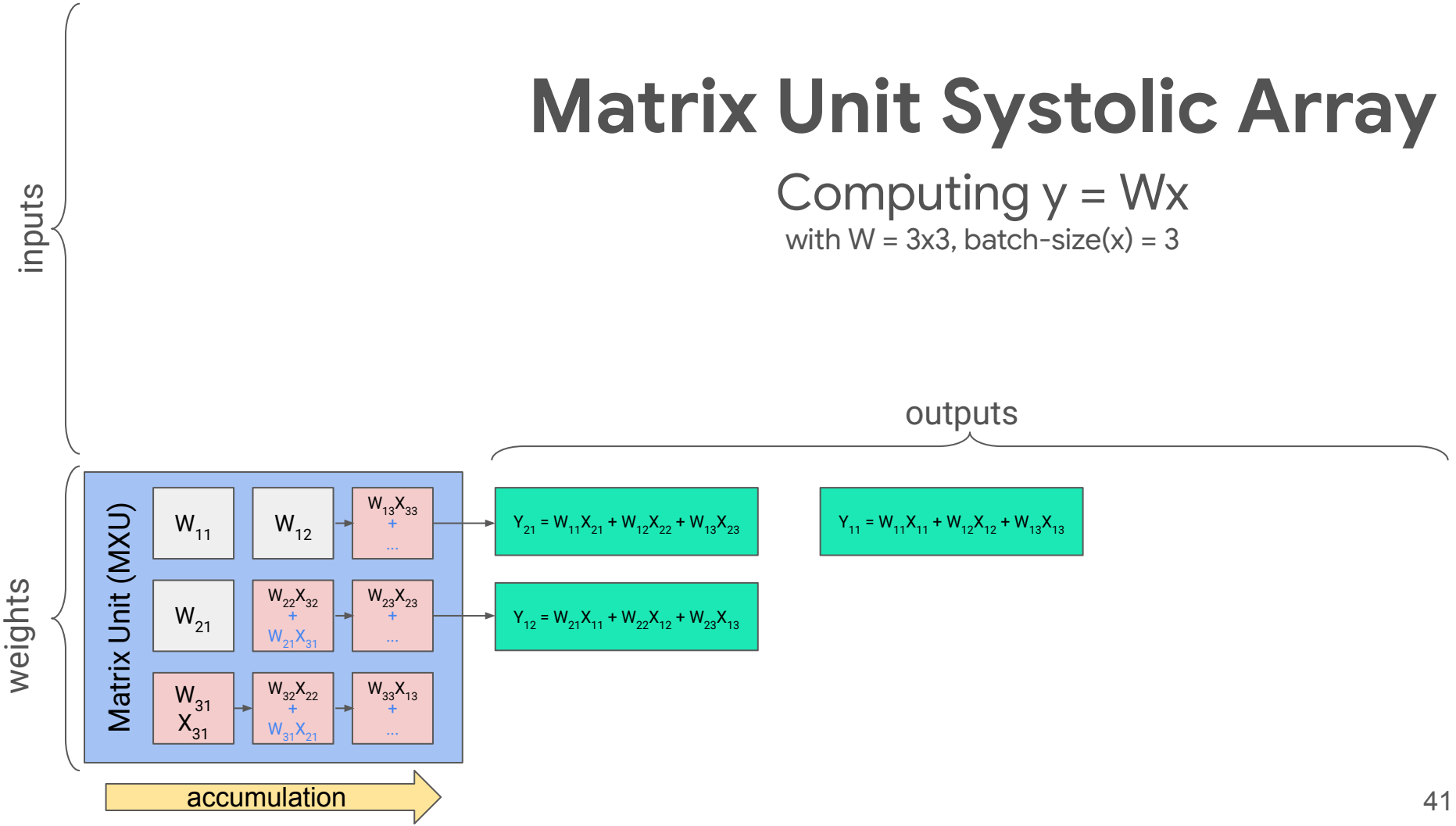
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

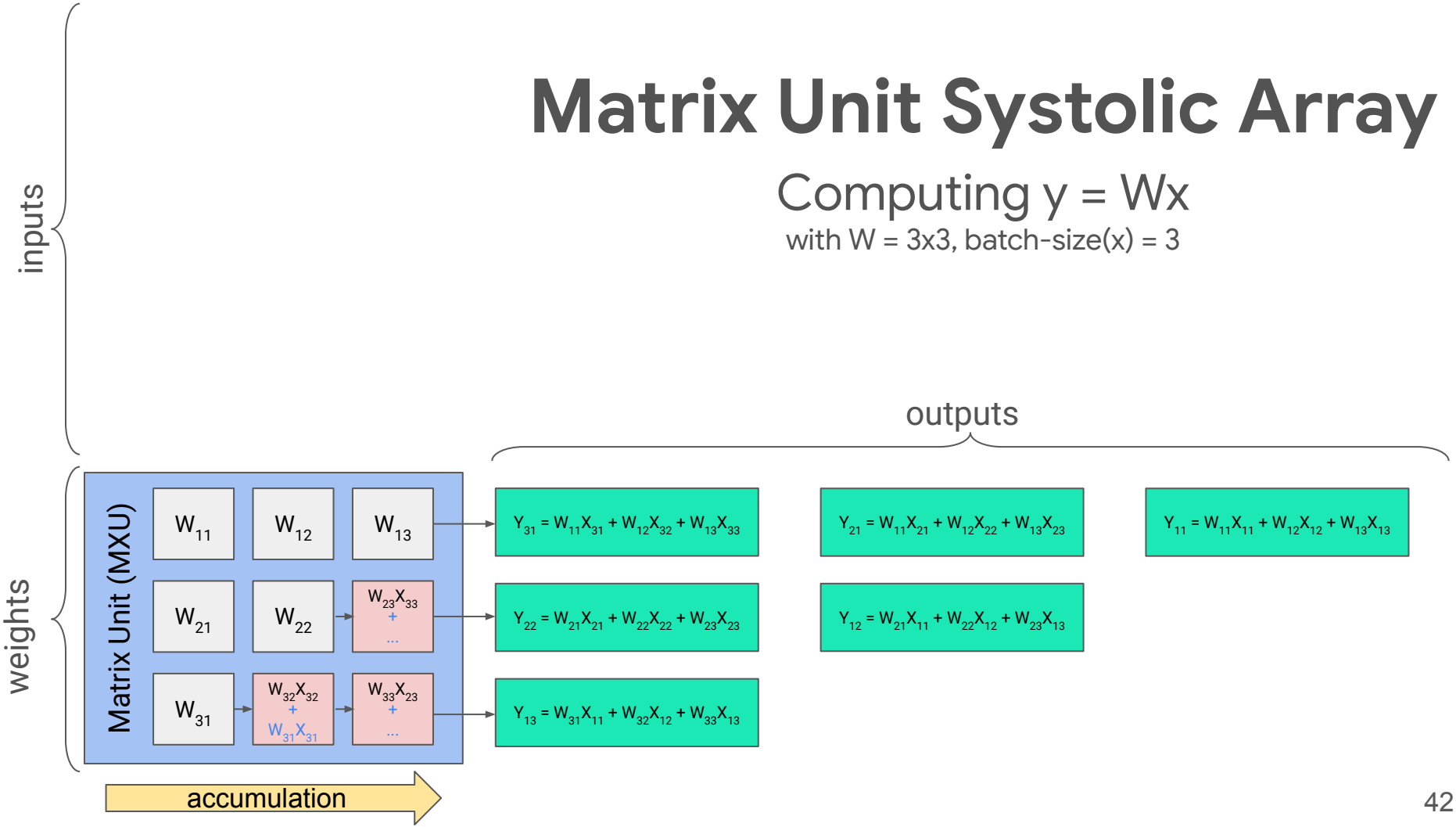
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

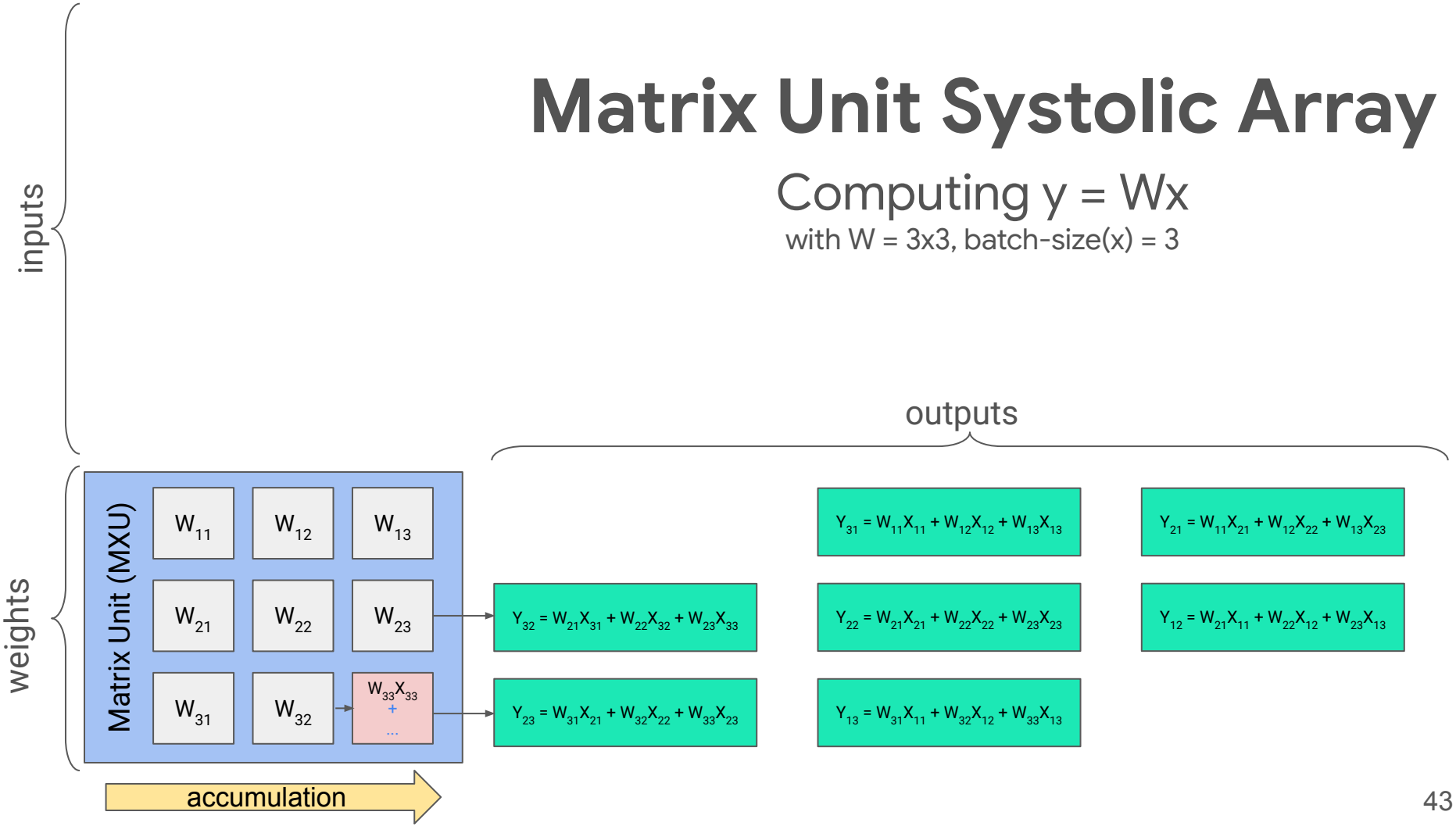
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

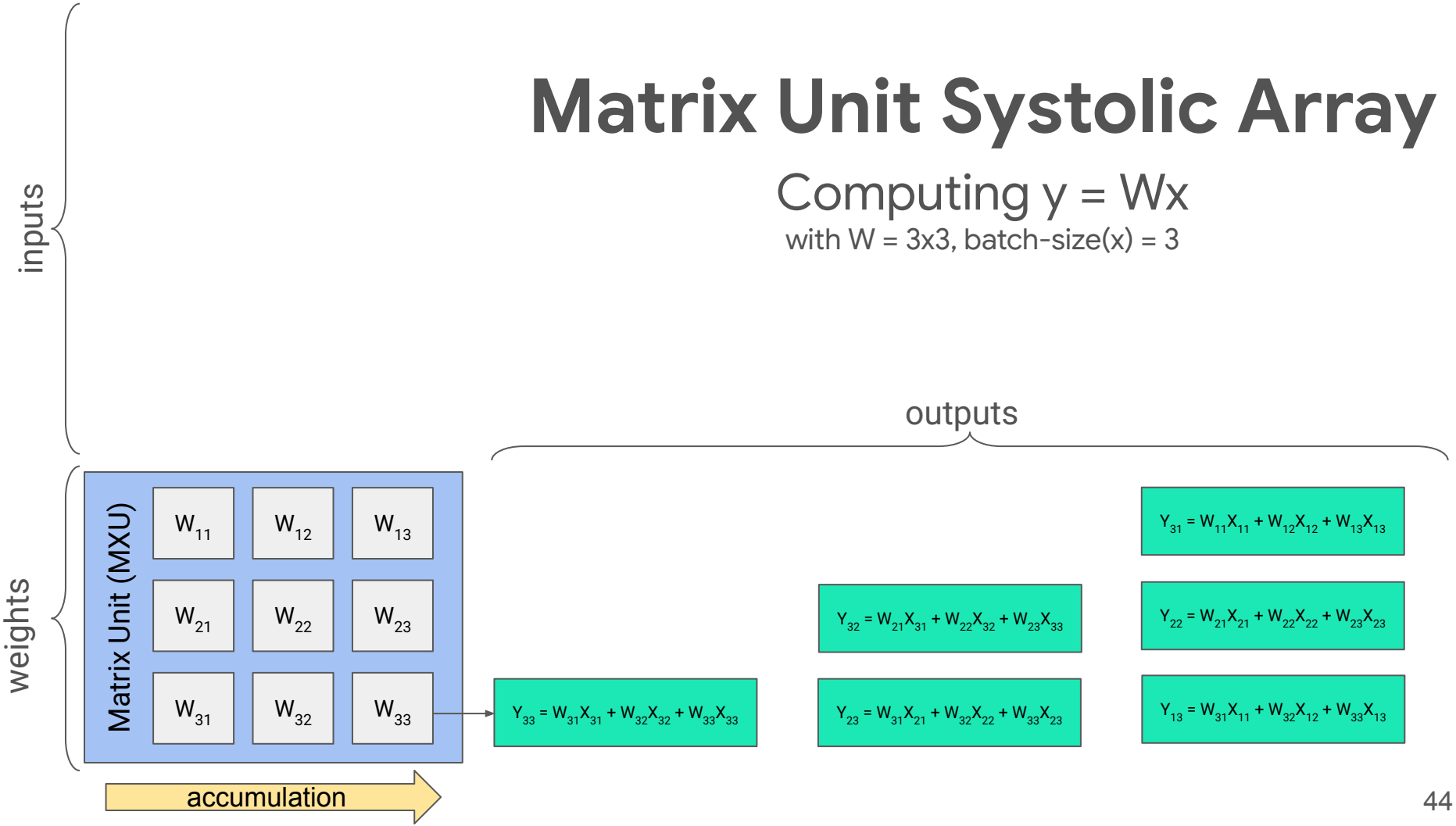
with W = 3x3, batch-size(x) = 3



Matrix Unit Systolic Array

Computing y = Wx

with W = 3x3, batch-size(x) = 3

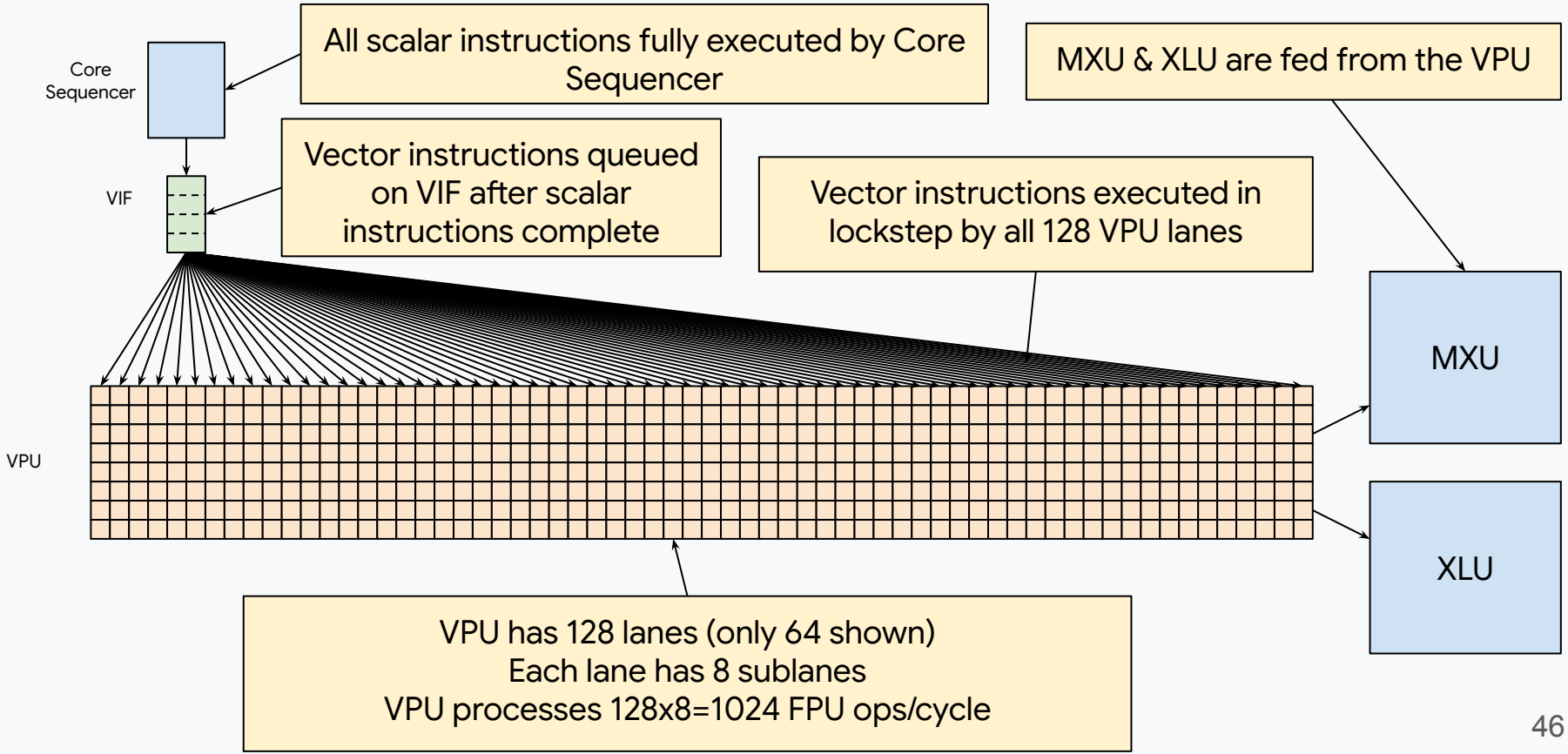


MXU Latency

For an NxN array, the time to complete one dense matmul is exactly **3N - 1 cycles**.

- **The "Fill" Phase:**
  It takes N cycles for the first activation entering at the top-left corner to reach the bottom-right corner of the grid. This is the time required to fully "prime" the systolic pipeline.
- **The "Drain" Phase:**
  Once the last activation enters the top row, it still needs to travel down the remaining rows to finish the final accumulations. In an optimized TPU MXU, the results are streamed out as they are finished.

TensorCore Instruction Flow

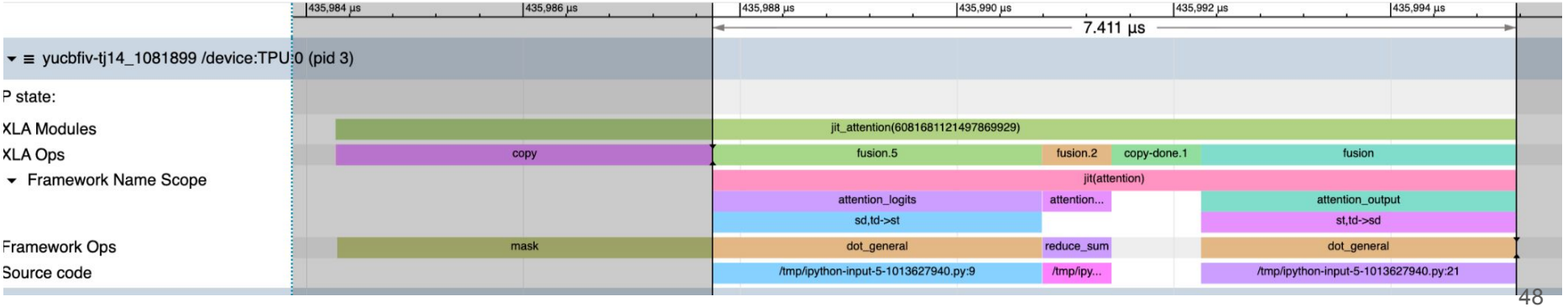


XLA Compilation

Basic Attention

```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)
```

```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
```



JAX Execution Lifecycle

```
apply_attn = jax.jit(attention) // returns PjitFunction
output = apply_attn(mask, q, k, v)
```

Compilation (Trace & Lowering)

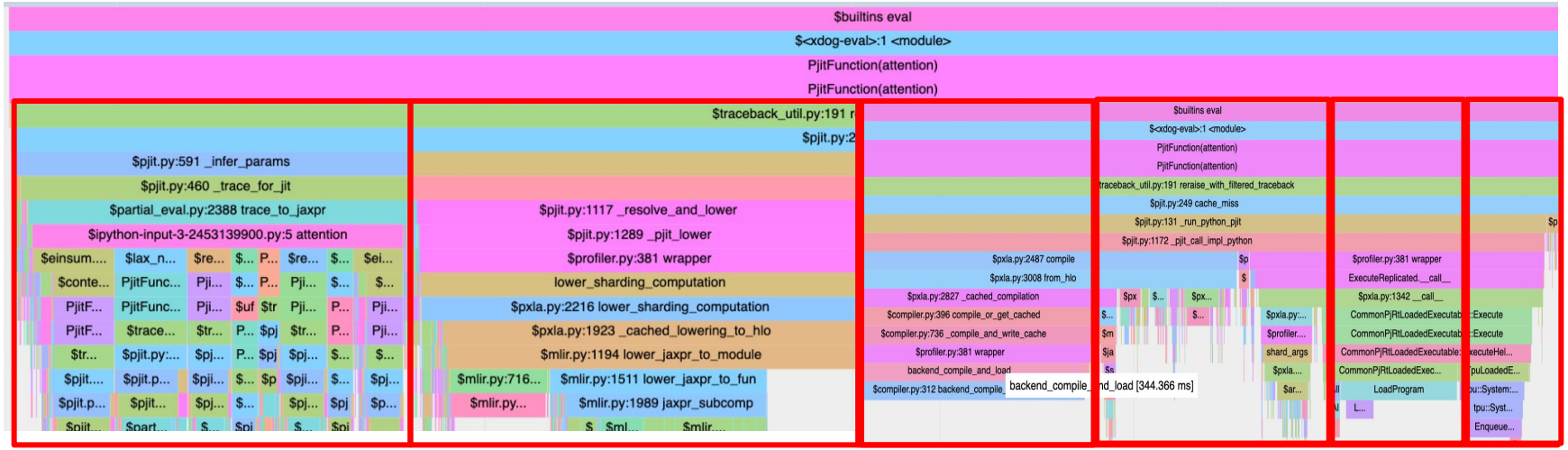
- **Dispatch & Cache:** PjitFunction hashes q, k, v signatures (shapes/dtypes/sharding). It retrieves the compiled executable from the **Compilation Cache** on a hit or triggers a new trace on a miss.
- **XLA Transformation:** Python logic is flattened into a **Jaxpr**, then lowered to **StableHLO**. XLA performs **Operator Fusion** (e.g., merging Mask + Softmax) and algebraic simplification to minimize memory overhead.
- **TPU Backend Specialization:** The HLO is translated into **LLO IR**, followed by **VLIW Scheduling** to map dot-products to the **MXU** (Matrix Unit). This produces the final hardware-specific **Binary Executable**.

Execution (Runtime)

- **Argument Handling:** Tensors are moved from Host (CPU) to TPU HBM. **Pjit** handles physical sharding across the **ICI** (Inter-Core Interconnect) if scaling across a TPU Pod.
- **Async Execution:** The Python thread launches the binary on the TPU and immediately returns a **Future** (jax.Array). The hardware executes the VLIW microcode asynchronously from the CPU.
- **Buffer Management:** The output is a pointer to TPU memory. JAX manages these buffers lazily; no data is copied back to the CPU until an explicit pull (e.g., .block_until_ready()).

Lifecycle of a JIT Call: Tracing, Compiling, and Executing

%%xprof
output = **apply_attn**(mask, q, k, v) // pjitfunction



- Cache miss
- Triggers Compilation
- Trace to Jaxpr

- Lowering
- Stable HLO

- Compile

StableHLO -> HLO
HLO→LLO → VLIW Bundles - Binary

- Add to Cache

- Inputs
- Sharding
- Host→HBM

Load Binary

Async Execution

JAX - TPU Compilation Pipeline

Compilation (The Pipeline)

- **Tracing: Python/JAX (Tracing) → Jaxpr**
  - # Captures high-level functional trace of the attention logic
  - attn_jaxpr = jax.**make_jaxpr(**attention)(mask, q, k, v)
- **Lowering: Jaxpr → StableHLO → HLO**
  - # Lowers JAX expressions to hardware-independent HLO (High-Level Optimizer) IR
  - attn_lowered = jax.**jit**(attention).**lower**(mask, q, k, v)
  - **stable_hlo** = attn_lowered.as_text()
  - **hlo** = attn_lowered.compiler_ir('hlo')
- **Compile: HLO → Optimized HLO → LLO IR → Scheduling → VLIW Bundles → Executable**
  - XLA performs Operator Fusion (e.g., Mask+Softmax) and algebraic simplification
  - Lowers to TPU Specific LLO IR and translates into scheduled VLIW bundles for binary creation
  - attn_compiled = attn_lowered.**compile**()
- **The Executable Object:** attn_compiled.**runtime_executable**() returns a **LoadedExecutable**.

Compiled Cost Analysis

- attn_compiled = attn_lowered.**compile**()
- attn_compiled.**cost_analysis**()

Metric	Value (Approx.)	Note / Significance
flops	2.15 GFLOPs	Total floating-point operations performed by the MXU.
bytes accessed	40.9 MB	Total HBM bandwidth consumed (Inputs + Outputs + Intermediate).
transcendentals	2.10M ops	Costly non-linear operations (specifically the exp in Softmax).
optimal_seconds	11.1 µs	Theoretical floor for execution time at 100% efficiency.
utilization0{}	10	Indicates the primary compute unit (MXU) is the lead worker.
Arithmetic Intensity	52.6 FLOP/Byte	Compare this TPU Ironwood Arithmetic intensity

```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)
```

JAX Tracing: Capturing the Computation Graph

Abstract Execution: No actual math or data processing occurs; JAX executes the Python code using **"abstract"** tracer objects instead of real arrays.

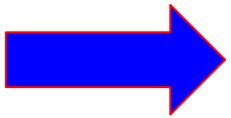
Recording the Flow: Every operation and its intermediate result is recorded into a sequence called a **JAXpr** (JAX expression).

Graph Reconstruction: This recorded sequence is used to reconstruct a complete **Operation Graph**, which the XLA compiler can then optimize and lower to hardware.

Side-Effect Isolation: Purely Python-side effects—like print() statements or list appends—are executed only once during tracing and are **excluded** from the final compiled graph.

Python/JAX (Tracing) → JAX Expression

```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
```



```
{ lambda ; a:bool[1024,1024] b:f32[1024,512] c:f32[1024,512] d:f32[1024,512]. let
    e:f32[1024,1024] = dot_general[
      dimension_numbers=(([1], [1]), ([], []))
      preferred_element_type=float32
    ] b c
    f:f32[1024,1024] = jit[
      name=_where
      jaxpr={ lambda ; a:bool[1024,1024] e:f32[1024,1024] g:f32[]. let
          h:f32[] = convert_element_type[new_dtype=float32 weak_type=False] g
          i:f32[1024,1024] = broadcast_in_dim[
            broadcast_dimensions=()
            shape=(1024, 1024)
            sharding=None
          ] h
          f:f32[1024,1024] = select_n a i e
        in (f,) }
    ] a e -1e+38:f32[]
    j:f32[1024] = reduce_max[axes=(1,)] f
    k:f32[1024,1] = broadcast_in_dim[
      broadcast_dimensions=(0,)
      shape=(1024, 1)
      sharding=None
    ] j
    l:f32[1024,1024] = sub f k
    m:f32[1024,1024] = exp l
    n:f32[1024] = reduce_sum[axes=(1,) out_sharding=None] m
    o:f32[1024,1] = broadcast_in_dim[
      broadcast_dimensions=(0,)
      shape=(1024, 1)
      sharding=None
    ] n
    p:f32[1024,1024] = div m o
    q:f32[1024,512] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] p d
  in (q,) }
```

attn_jaxpr = jax.**make_jaxpr**(attention)(mask, q, k, v)

JAX Lowering

The JAX Intermediate Representation (Jaxpr)

- JAX traces Python functions to generate a **Jaxpr** (JAX expression).
- The **Jaxpr** is JAX's internal IR, serving as a functional expression of JAX primitive operations (e.g., add_p).
- **Why a framework IR?** It provides a stable target for core JAX transformations like grad and vmap, which are implemented directly on the Jaxpr.

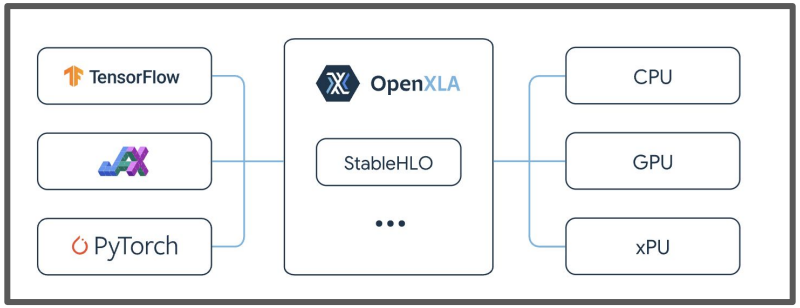
Transformation & Registration

- Each JAX primitive registers specific functions for:
  - **StableHLO Lowering:** Translating the primitive into the hardware-neutral StableHLO IR.
  - **Transformations:** Logic for how the primitive behaves under grad (differentiation) and vmap (vectorization).

https://github.com/jax-ml/jax/blob/main/jax/_src/lax/lax.py

```
# Defining the primitive as a functional JAX operation
add_p: Primitive = naryop(input_dtype, [_num, _num], 'add',
                          unreduced_rule=_add_unreduced_rule)

# Registering transformation rules (grad/vmap)
ad.primitive_jvps[add_p] = _add_jvp
ad.primitive_transposes[add_p] = _add_transpose

# Registering the hardware lowering to StableHLO
mlir.register_lowering(add_p, partial(_nary_lower_hlo, hlo.add))
```

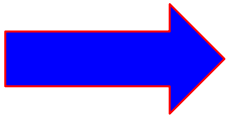
Why do we need StableHLO ?



- **Decouples machine learning frameworks from hardware**:
  - Solves the NxM problem by decoupling frameworks from direct hardware support.
- **Multiple ML frameworks lower to StableHLO**
  - JAX, TensorFlow, PyTorch
- **StableHLO is hardware agnostic**.
  - Has a spec and serialization format
  - Compiled by XLA into target hardware: CPU, TPU, GPU.
- **Backward Compatibility**
  - A permanent "contract" ensuring models exported today remain functional with future compiler updates.

JAX Expression → Stable HLO

```
{ lambda ; a:bool[1024,1024] b:f32[1024,512] c:f32[1024,512] d:f32[1024,512]. let
    e:f32[1024,1024] = dot_general[
      dimension_numbers=(([1], [1]), ([], []))
      preferred_element_type=float32
    ] b c
    f:f32[1024,1024] = jit[
      name=_where
      jaxpr={ lambda ; a:bool[1024,1024] e:f32[1024,1024] g:f32[]. let
          h:f32[] = convert_element_type[new_dtype=float32 weak_type=False] g
          i:f32[1024,1024] = broadcast_in_dim[
            broadcast_dimensions=()
            shape=(1024, 1024)
            sharding=None
          ] h
          f:f32[1024,1024] = select_n a i e
        in (f,) }
    ] a e -1e+38:f32[]
    j:f32[1024] = reduce_max[axes=(1,)] f
    k:f32[1024,1] = broadcast_in_dim[
      broadcast_dimensions=(0,)
      shape=(1024, 1)
      sharding=None
    ] j
    l:f32[1024,1024] = sub f k
    m:f32[1024,1024] = exp l
    n:f32[1024] = reduce_sum[axes=(1,) out_sharding=None] m
    o:f32[1024,1] = broadcast_in_dim[
      broadcast_dimensions=(0,)
      shape=(1024, 1)
      sharding=None
    ] n
    p:f32[1024,1024] = div m o
    q:f32[1024,512] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] p d
  in (q,) }
```



```
module @jit_attention attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas = 1 : i32} {
  func.func public @main(%arg0: tensor<1024x1024xi1>, %arg1: tensor<1024x512xf32>, %arg2:
tensor<1024x512xf32>, %arg3: tensor<1024x512xf32>) -> (tensor<1024x512xf32> {jax.result_info = "result"}) {
    %0 = stablehlo.dot_general %arg1, %arg2, contracting_dims = [1] x [1], precision = [DEFAULT, DEFAULT] :
(tensor<1024x512xf32>, tensor<1024x512xf32>) -> tensor<1024x1024xf32>
    %cst = stablehlo.constant dense<-9.99999968E+37> : tensor<f32>
    %1 = call @_where(%arg0, %0, %cst) : (tensor<1024x1024xi1>, tensor<1024x1024xf32>, tensor<f32>) ->
tensor<1024x1024xf32>
    %cst_0 = stablehlo.constant dense<0xFF800000> : tensor<f32>
    %2 = stablehlo.reduce(%1 init: %cst_0) applies stablehlo.maximum across dimensions = [1] :
(tensor<1024x1024xf32>, tensor<f32>) -> tensor<1024xf32>
    %3 = stablehlo.broadcast_in_dim %2, dims = [0] : (tensor<1024xf32>) -> tensor<1024x1xf32>
    %4 = stablehlo.broadcast_in_dim %3, dims = [0, 1] : (tensor<1024x1xf32>) -> tensor<1024x1024xf32>
    %5 = stablehlo.subtract %1, %4 : tensor<1024x1024xf32>
    %6 = stablehlo.exponential %5 : tensor<1024x1024xf32>
    %cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %7 = stablehlo.reduce(%6 init: %cst_1) applies stablehlo.add across dimensions = [1] :
(tensor<1024x1024xf32>, tensor<f32>) -> tensor<1024xf32>
    %8 = stablehlo.broadcast_in_dim %7, dims = [0] : (tensor<1024xf32>) -> tensor<1024x1xf32>
    %9 = stablehlo.broadcast_in_dim %8, dims = [0, 1] : (tensor<1024x1xf32>) -> tensor<1024x1024xf32>
    %10 = stablehlo.divide %6, %9 : tensor<1024x1024xf32>
    %11 = stablehlo.dot_general %10, %arg3, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] :
(tensor<1024x1024xf32>, tensor<1024x512xf32>) -> tensor<1024x512xf32>
    return %11 : tensor<1024x512xf32>
  }
  func.func private @_where(%arg0: tensor<1024x1024xi1>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<f32>) -> tensor<1024x1024xf32> {
    %0 = stablehlo.convert %arg2 : tensor<f32>
    %1 = stablehlo.broadcast_in_dim %0, dims = [] : (tensor<f32>) -> tensor<1024x1024xf32>
    %2 = stablehlo.select %arg0, %arg1, %1 : tensor<1024x1024xi1>, tensor<1024x1024xf32>
    return %2 : tensor<1024x1024xf32>
  }
}
```

HLO (High Level Optimizer)

- **Input language for XLA**
  - XLA: Accelerated Linear Algebra
- **Directed acyclic graph of computational operations.**
- **No ordering or recursion**: all computations are effectively linked.
  - Most instructions are purely functional without side effects.
  - Calculate A, Calculate B, Calculate C (using A and B)
    - A and B run in any order or simultaneously before C (dependency)
- **All array dimensions are known at compile time**
  - So memory usage can be fully determined at compile time.
  - Simplifies machine code generation.
- **Limited opcodes**
  - Elementwise: Add, Multiply, etc.
  - Array: Transpose, Dot, etc.
  - Collectives: AllGather, ReduceScatter, etc.

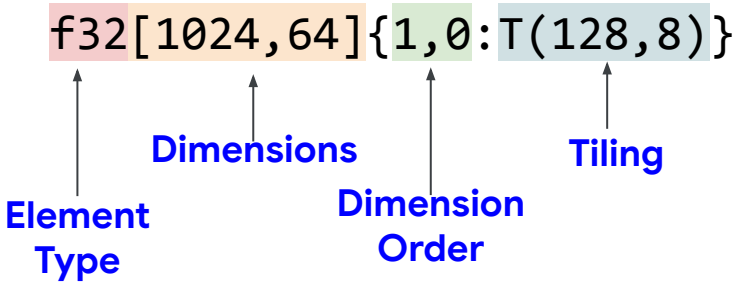
Stable HLO → HLO

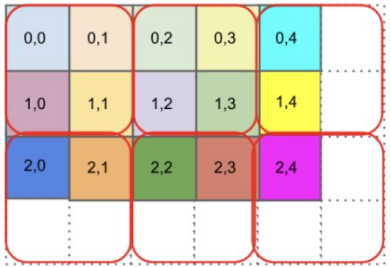
```
HloModule jit_attention,
entry_computation_layout={(pred[1024,1024]{1,0},
f32[1024,512]{1,0}, f32[1024,512]{1,0},
f32[1024,512]{1,0})->f32[1024,512]{1,0}}

_where.1 {
  Arg_0.1 = pred[1024,1024]{1,0} parameter(0)
  Arg_1.1 = f32[1024,1024]{1,0} parameter(1)
  Arg_2.1 = f32[] parameter(2)
  broadcast_in_dim.1 = f32[1024,1024]{1,0} broadcast(Arg_2.1),
dimensions={}
  ROOT select_n.1 = f32[1024,1024]{1,0} select(Arg_0.1, Arg_1.1,
broadcast_in_dim.1)
}

region_0.2 {
  reduce_max.3 = f32[] parameter(0)
  reduce_max.4 = f32[] parameter(1)
  ROOT reduce_max.5 = f32[] maximum(reduce_max.3,
reduce_max.4)
}

region_1.3 {
  reduce_sum.3 = f32[] parameter(0)
  reduce_sum.4 = f32[] parameter(1)
  ROOT reduce_sum.5 = f32[] add(reduce_sum.3, reduce_sum.4)
}
```

```
HloModule jit_attention, entry_computation_layout={(pred[1024,1024]{1,0},
ENTRY main.4 {
  mask.1 = pred[1024,1024]{1,0} parameter(0)
  q.1 = f32[1024,512]{1,0} parameter(1)
  k.1 = f32[1024,512]{1,0} parameter(2)
  dot_general.2 = f32[1024,1024]{1,0} dot(q.1, k.1), lhs_contracting_dims={1},
rhs_contracting_dims={1}
  constant.5 = f32[] constant(-1e+38)
  jit__where_.1 = f32[1024,1024]{1,0} call(mask.1, dot_general.2, constant.5),
to_apply=_where.1
  constant.4 = f32[] constant(-inf)
  reduce_max.7 = f32[1024]{0} reduce(jit__where_.1, constant.4), dimensions={1},
to_apply=region_0.2
  broadcast_in_dim.4 = f32[1024,1]{1,0} reshape(reduce_max.7)
  sub.4 = f32[1024,1]{1,0} broadcast(broadcast_in_dim.4), dimensions={0,1}
  sub.5 = f32[1024]{0} reshape(sub.4)
  sub.6 = f32[1024,1024]{1,0} broadcast(sub.5), dimensions={0}
  sub.7 = f32[1024,1024]{1,0} subtract(jit__where_.1, sub.6)
  exp.1 = f32[1024,1024]{1,0} exponential(sub.7)
  constant.3 = f32[] constant(0)
  reduce_sum.7 = f32[1024]{0} reduce(exp.1, constant.3), dimensions={1},
to_apply=region_1.3
  broadcast_in_dim.5 = f32[1024,1]{1,0} reshape(reduce_sum.7)
  div.4 = f32[1024,1]{1,0} broadcast(broadcast_in_dim.5), dimensions={0,1}
  div.5 = f32[1024]{0} reshape(div.4)
  div.6 = f32[1024,1024]{1,0} broadcast(div.5), dimensions={0}
  div.7 = f32[1024,1024]{1,0} divide(exp.1, div.6)
  v.1 = f32[1024,512]{1,0} parameter(3)
  ROOT dot_general.3 = f32[1024,512]{1,0} dot(div.7, v.1),
lhs_contracting_dims={1}, rhs_contracting_dims={0}
}
```

XLA Shapes and Memory Layout

- **Shape Basics**
  - Every array is defined by its data type and fixed dimensions, which are predetermined at compile time to ensure efficient memory allocation
- **Memory Layout (Minor-to-Major)**
  - Defines storage order from fastest-changing (minor) to slowest-changing (major) dimensions.
- **Tiling (Cache Efficiency)**
  - Arrays are split into smaller blocks (tiles) to fit into high-speed hardware memory.
  - XLA moves through the grid of tiles first, then through the elements within each tile.





Example
f32[3,5]{1,0:T(2,2)}

Sequential order in memory layout

XLA Compiler

The Frontend (HLO Transformer)

- **Graph Optimization:** Operates exclusively on the HLO graph, transforming the user's input into a optimized version ready for code generation.
- **Modular Passes:** Implemented as a series of device-independent and device-specific transformations (passes) that act on the graph structure.
- **Shared Codebase:** While each device has its own frontend, the majority of the optimization logic is shared across all hardware targets.

The Backend (Machine Code Generator)

- **Direct Translation:** Generates target-specific machine code from the final HLO graph without making further structural changes to the graph itself.
- **Hardware-Specific:** Backends are tailored for specific chips (TPU, GPU, CPU) and use "emitters" to append necessary machine instructions for each operation.
- **Fine-Grained Tuning:** Optimizations at this stage focus on individual machine instructions or SSA(Static Single Assignment)-level details to squeeze maximum performance out of the hardware.

HLO Optimization Passes

Logic & Math: Simplifies arithmetic (converting a constant division into multiplication by an inverse), folds constants at compile time, and removes dead code to minimize the graph.

Structural Refinement: The Reshape Mover pushes layout changes to the graph's edges, allowing math operations to sit together and fuse into high-speed loops.

Layout & Tiling: Organizes data in memory and partitions large tensors into **128 × 8 Chunks** to match the VREG architecture.

Kernel Fusion: Combines multiple operations (like vadd and vmul) into single kernels to minimize VMEM traffic.

Buffer & Copy Insertion: Assigns memory buffers and inserts copy instructions to manage data movement between memory tiers.

Scheduling & Latency Hiding: Reorders instructions and manages asynchronous operations to hide execution latency.

Memory Space Assignment: Finalizes the physical placement of tensors in VMEM or CMEM before generating the final code.

Streamlined HLO: Softmax Data Path

1. **dot**: Compute Logits.
2. **select**: Apply Mask.
3. **reduce**: Find Max (stability)
4. **Subtract:** Shift
5. **exp**: Exponentiate.
6. **reduce**: Compute Sum.
7. **divide**: Normalize.
8. **dot**: Final Weighted Sum.

```
/* 1. Logic & Math: Redundant reshapes collapsed into a direct broadcast */
%sub.6 = f32[1024,1024] broadcast(%reduce_max.7), dimensions={0}
%sub.7 = f32[1024,1024] subtract(%logits, %sub.6)

/* 2. Structural Refinement: Core math ops sit together, ready for fusion */
%exp.1 = f32[1024,1024] exponential(%sub.7)

/* 3. Logic & Math: 1D reduction result broadcasted directly back to 2D matrix */
%reduce_sum.7 = f32[1024] reduce(%exp.1, %c0), dimensions={1}, to_apply=%add_region
%div.6 = f32[1024,1024] broadcast(%reduce_sum.7), dimensions={0}

/* 4. Structural Refinement: Contiguous chain ready to be a single VPU kernel */
%div.7 = f32[1024,1024] divide(%exp.1, %div.6)
```

Physical Layouts & Tiling Transformations - 1

- **Mapping to the Hardware Grid:** The compiler translates logical tensor dimensions into the physical TPU core geometry. Using the **T(8,128)** tiling annotation, it partitions 1024x1024 data into the native hardware grid of **8 sublanes** and **128 lanes**.

```
/* MXU Operation: Row-Major LHS fed to systolic grid */
%q.1 = f32[1024,512]{1,0:T(8,128)} parameter(1)
%k.1 = f32[1024,512]{1,0:T(8,128)} parameter(2)

/* Result drains vertically out of MACs in Column-Major {0,1} */
%convolution = f32[1024,1024]{0,1:T(8,128)} convolution(%q.1,
%k.1)
```

- **Physical Layout Alignment:** The **copy** instruction performs a crucial layout transformation (e.g., Row-Major to Column-Major). This ensures that the **Mask** and **Logits** are physically aligned in memory, allowing the VPU to perform element-wise selection without data-shuffling stalls.

```
/* Logical Mask is usually Row-Major {1,0} */
%mask.1 = pred[1024,1024]{1,0:T(8,128)} parameter(0)

/* Copy flips it to Column-Major {0,1} to align with Logits */
%copy = pred[1024,1024]{0,1:T(8,128)} copy(%mask.1) /

* Now indices match perfectly for the 1024 VPU ALUs */
%select_n.0 = f32[1024,1024]{0,1:T(8,128)} select(%copy, %convolution,
%constant.5)
```

Physical Layouts & Tiling Transformations - 2

- **Softmax Data Path (VPU Chain):**
  The data stays in the VREGs in Column-Major order throughout the numerical stability and normalization passes.

```
/* Find Max per column -> Result is 1D Vector */
%reduce_max.7 = f32[1024]{0:T(1024)} reduce(%select_n.0, ...)

/* Broadcast 1D Max across the 2D matrix (still Column-Major) */
%sub.6 = f32[1024,1024]{0,1:T(8,128)} broadcast(%reduce_max.7),
dimensions={0}

/* Element-wise math chain: All {0,1} */
%sub.7 = f32[1024,1024]{0,1:T(8,128)} subtract(%select_n.0, %sub.6)
%exp.1 = f32[1024,1024]{0,1:T(8,128)} exponential(%sub.7)
%div.7 = f32[1024,1024]{0,1:T(8,128)} divide(%exp.1, %div.6)
```

- **Final Weighted Sum (MXU Intake):**
  The Softmax result (VPU output) now serves as the LHS input for the final projection.

```
/* Softmax weights come from VPU in Column-Major {0,1} */
/* Values (V) are fed in Row-Major {1,0} */
%v.1 = f32[1024,512]{1,0:T(8,128)} parameter(3)

/* Interpretation: MXU treats div.7 as a Column-Major LHS (bf_io) */
ROOT %convolution.1 = f32[1024,512]{1,0:T(8,128)} convolution(%div.7,
%v.1), dim_labels=bf_io->bf
```

Fusion - Kernel 1: Logit Generation & Stability Peak

- **Fuses** convolution (Q × K) ⟶ select (Mask) ⟶ reduce_max into one execution block.
- **Hardware Interaction:** The **MXU** systolic array streams raw logits directly into the **VPU** ALUs. These ALUs immediately "zip" the mask with the incoming stream and scan for the maximum value in real-time.

- **The Output Tuple:** Produces a dual-result output to minimize memory traffic:
  - **1D Vector (f32[1024]):** The **Maximums** for each row, stored and ready to stabilize the upcoming Exponential pass.
  - **2D Matrix (f32[1024,1024]):** The **Masked Logits**, already physically tiled (T(8,128)) and aligned in Column-Major order for the next kernel.
- **Optimization Result:** By consuming logits as they are produced, the compiler ensures numerical stability without ever writing the massive, raw intermediate matrix back to VMEM.

%fusion.5 = (**f32[1024]{0:T(1024)}, f32[1024,1024]{0,1:T(8,128)}**) fusion(%copy, %q.1, %k.1), kind=**kOutput**, calls=%fused_computation.7, metadata={...}

Fusion - Kernel 2: The Exponent & Summation Loop

- **Chained Ops:** subtract (Max) → exponential → reduce_sum.
- **Hardware Interaction:** This is a tight loop where the VPU pulls the **Masked Logits** and the **1D Max Vector** from VMEM. It performs the "Shift-and-Exp" math and immediately accumulates the result into a running sum.

- **The VREG Advantage:** The 1024x1024 results of the exponential operation are consumed by the reduce_sum while they are still sitting in the **Vector Registers (VREGs)**.
- **Optimization Result:** The huge intermediate matrix of exponentials is **never written to VMEM**. The compiler "streams" the data through the ALUs, producing only a tiny **1D Sum Vector** as the final output.

```
%get-tuple-element.1 = f32[1024,1024]{0,1:T(8,128)} get-tuple-element(%fusion.5), index=1    // Masked Logits
%get-tuple-element = f32[1024]{0:T(1024)} get-tuple-element(%fusion.5), index=0   // Maximums
%fusion.2 = f32[1024]{0:T(1024)} fusion(%get-tuple-element.1, %get-tuple-element), kind=kLoop, calls=%fused_computation.3
```

Performance Through Rematerialization (Recompute)

To achieve peak TFlops, the XLA compiler prioritizes **ALU math over memory movement**. Instead of storing the large Softmax matrix, it "rematerializes" (recomputes) it on the fly.

The "Memory vs. Math" Trade-off

- **The Memory Bottleneck:** Storing the 1024x1024 exponentiated matrix after Kernel 2 would require a **4MB write** to VMEM and a **4MB read** back for Kernel 3. This 8MB "bandwidth tax" would slow down the entire pipeline.
- **The ALU Efficiency:** The cycle cost of re-calculating exp(x - max) in the VPU is significantly lower than the latency of fetching 8MB from memory.

Math is cheap; bandwidth is expensive.

```
%get-tuple-element.1 = f32[1024,1024]{0,1:T(8,128)} get-tuple-element(%fusion.5), index=1    // Masked Logits
%get-tuple-element = f32[1024]{0:T(1024)} get-tuple-element(%fusion.5), index=0   // Maximums
%fusion.2 = f32[1024]{0:T(1024)} fusion(%get-tuple-element.1, %get-tuple-element) // Sum

ROOT %fusion = f32[1024,512]{1,0:T(8,128)} fusion(%v.1, %get-tuple-element.1, %fusion.2, %get-tuple-element), kind=kOutput
```

Fusion - Kernel 3: Attention Output Fusion

Recompute & Normalization Logic

- **Input Streams:** Kernel 3 pulls the **original Masked Logits** (from Kernel 1) and the **1D Max/Sum vectors** (from Kernels 1 & 2).
- **VREG Execution:** The VPU re-performs the subtract and exponential steps purely within the **Vector Registers**.
- **The Handshake:** As the VPU finishes the final divide (normalization) for a data tile, it streams those weights directly into the **MXU systolic array**.

Final Weighted Projection

- **Hardware Interaction:** The **MXU** consumes weights directly from the **VPU** registers to multiply them by the **Values (V)** matrix, ensuring the 1024x1024 Softmax matrix exists only in the chip's internal wiring and is never physically stored in memory.
- **Layout Restoration:** While weights are processed in **Column-Major ({0,1})** to stay aligned with the VPU, the MXU drains the final result in **Row-Major ({1,0})**, matching the format required by the rest of the model.

Core Architectural Decisions in Attention

- **Layout Selection:** The compiler forces a **Column-Major ({0,1})** layout with **8x128 hardware tiling**. This ensures the **VPU** can sweep across registers with zero striding overhead, enabling high-speed reduce_max and reduce_sum at peak hardware bandwidth.

- **Fusion 1 & 2 (The "Ghost Matrix"):**The kernels are fused to produce only **1D vectors** (Max and Sum) for VMEM. The massive 1024x1024 **Exponential matrix** exists only in the registers and is discarded immediately after summation, avoiding a 4MB memory write-back.

- **Rematerialization:** Kernel 3 **re-calculates** the Softmax math (e^{x-max} / sum) locally instead of reading pre-saved weights. By choosing high-speed ALU math over a slow **8MB memory fetch** (Write + Read), the compiler recovers significant bandwidth and maintains higher TFlops.

Physical Data Movement & Memory Tiering

Explicit Memory Tiering (S(n)): In the final HLO, tensors are annotated with specific memory spaces to define their physical location on the chip:

- **S(0) (HBM):** Off-chip High Bandwidth Memory. This is where parameters like weights start.
- **S(1) (VMEM):** On-chip VMEM. High-speed, low-latency storage located directly next to the ALUs.
- **S(2) The Signal (Sync Tokens):** Special hardware semaphore registers.

The copy-start instruction triggers an asynchronous DMA transfer of the **Values (V)** matrix from slow **S(0) (HBM)** to the fast **S(1) (VMEM)**, using a tiny **S(2) Sync Token** to signal the ALUs exactly when the data is ready for computation.

- %copy-start.1 = (**f32[1024,512]{1,0:T(8,128)S(1)**}, f32[1024,512]{1,0:T(8,128)}, u32[]{:S(2)}) copy-start(%v.1)

Async Scheduling

Async Scheduling & Latency Hiding: The compiler uses a **copy-start / copy-done** protocol to overlap slow memory transfers with high-speed computation.

- **Non-Blocking Prefetch: copy-start** initiates a DMA transfer of weights (e.g., V) from HBM to VMEM. Crucially, the ALU does not wait for this to finish.
- **Execution Overlap:** While the DMA controller is moving data for the next kernel, the ALUs are simultaneously executing the current kernel (e.g., while V is being copied, the VPU is finishing the Sum reduction).
- **Barrier Synchronization: copy-done** acts as a lightweight barrier, ensuring the ALUs only begin processing once the destination buffer in S(1) is fully populated.

```
%copy-start.1 = (f32[1024,512]{1,0:T(8,128)S(1)}, f32[1024,512]{1,0:T(8,128)}, u32[]{:S(2)}) copy-start(%v.1)
%fusion.2 = f32[1024]{0:T(1024)S(1)} fusion(%get-tuple-element.1, %get-tuple-element), kind=kLoop, calls=%fused_computation.3
%copy-done.1 = f32[1024,512]{1,0:T(8,128)S(1)} copy-done(%copy-start.1)
ROOT %fusion = f32[1024,512]{1,0:T(8,128)} fusion(%copy-done.1, %get-tuple-element.1, %fusion.2, %get-tuple-element),
kind=kOutput, calls=%fused_computation  // Attention Output Fusion
```

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×KT), 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 (ex−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)

LLO Lowering

The Final Bridge to Hardware: After HLO optimizations, XLA's TPU backend lowers each fusion into **Low Level Operators (LLO)**—a hardware-specific intermediate representation that maps directly to the physical units of the TensorCore.

Hardware Mapping & Resource Partitioning

- **Specialized Units**: LLO instructions are specifically addressed to the **MXU** (Matrix Multiply), **VPU** (Vector Unit), and **XLU** (Cross-Lane Unit).

VLIW: The "Very Long Instruction Word" Bundle

- The compiler's primary goal is **saturation**. It packs multiple independent LLO operations into a single **512-bit VLIW bundle** to be executed in lockstep.

The TPU Execution Core: VLIW

The TPU TensorCore maximizes efficiency by offloading scheduling to the **XLA compiler**, using **Very Long Instruction Word (VLIW)** bundles to drive parallel execution.

The VLIW Orchestration

- **Parallel Launch**: Packs multiple independent operations into a single **512-bit bundle**, launching them simultaneously across subsystems.
- **Compiler-Driven Scheduling**: Shifts complexity from hardware to **XLA**, which orchestrates multiple hardware units in parallel every cycle.
- **Deterministic Control**: XLA uses fixed hardware latencies to sequence bundles perfectly, ensuring constant utilization without the energy cost of dynamic scheduling.

LLO Bundle

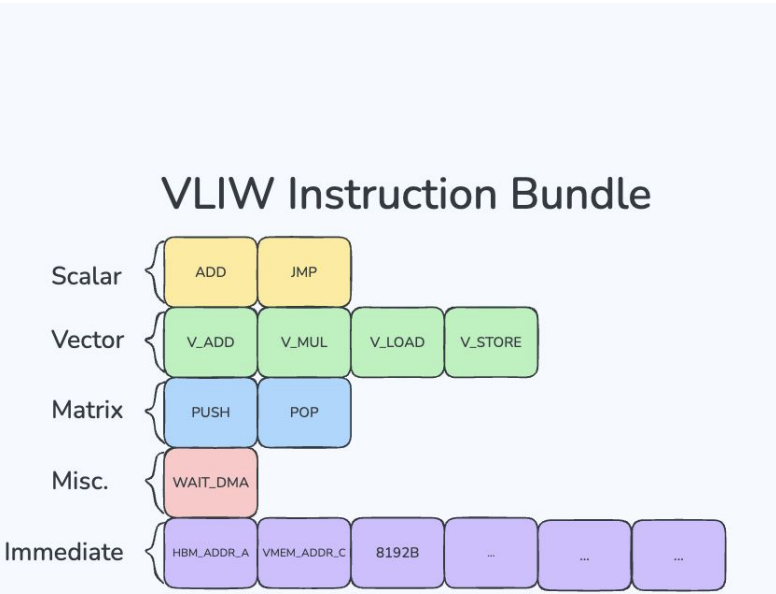
- **Final Execution Unit**: The ultimate, packed instruction format XLA sends to the TensorCore.
- **Latency Hiding**: Specifically designed to ensure **data movement (Loads)** never stalls **computation (Operations)**.
- **High-Density Packing**: Typically overlaps **2–3 memory loads** with a single matrix contraction to keep all 65,536 MAC units busy (256x256).

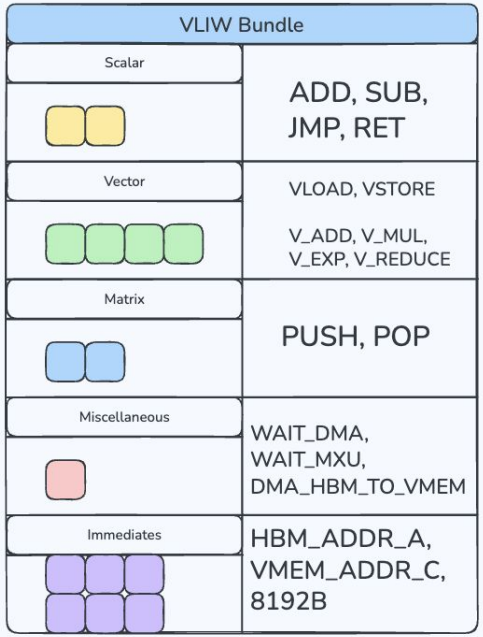
The Anatomy of a VLIW Bundle

A single bundle contains specialized slots that map directly to parallel functional units:

- **Logic & Control (Scalar & Predicate Slots):** Manages program flow through jumps and handles synchronization guards like **WAIT** instructions to prevent data hazards.
- **Vector Processing (4 Vector ALU Slots):** Powers the **Vector Processing Unit (VPU)** to perform element-wise arithmetic across 128-way SIMD lanes.
- **Matrix Multiplication (Matrix Push/Pop Slots):** Dedicated slots that "push" data into the **MXU** systolic array and "pop" finished results back into vector registers.
- **Memory Pipelining (3 Slots):** Features **2 Vector Loads** and **1 Vector Store**, enabling the core to move data between memory and registers without interrupting active computation.
- **Direct Data (6 Immediate Slots):** Directly supplies raw constants and memory addresses to functional units, avoiding extra cycles for literal loading.

Example - VLIW Instruction Bundle





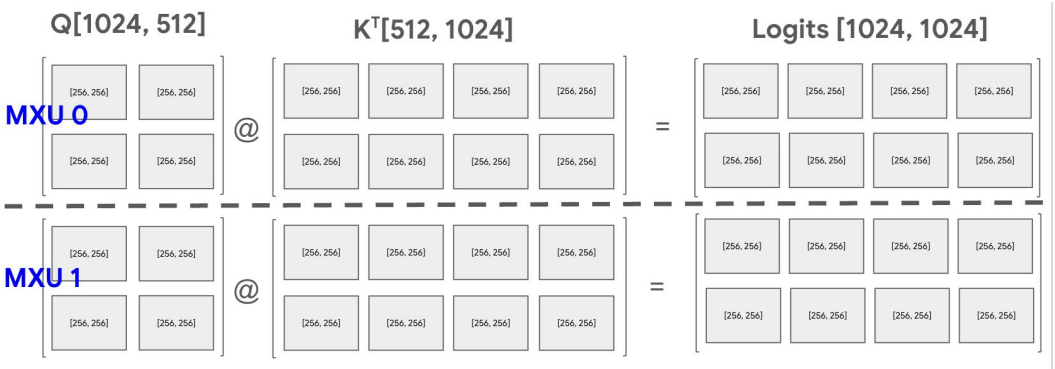
Logit & Max Fusion (%fused_computation.7)

Objective: High-level HLO functional definition of the kernel.

- **Fusion Purpose:** Executes the initial MatMul, applies masking, and calculates the max in a single fused pass to minimize VMEM traffic.
- **Inputs:** * **Query (Q):** 1024x512
  - **Key (K):** 1024x512 (Contracted as K^T of 512x1024)
  - **Mask:** 1024x1024
- **Outputs:** * **Masked Logits:** 1024x1024
- **Max Logits:** 1024 (One value per row for Softmax stabilization)
- **Fused Benefit:** By calculating the **Max** immediately after the **Pop**, the TPU avoids writing 1024x1024 intermediate values back to HBM just to read them again for Softmax.

Logical Decomposition & Hardware Mapping

- Map the 1024x1024 output to the 256x256 physical systolic array.
- The logical 1024x1024 result is divided into a **4x4 grid** of physical blocks.
- **Dual MXU Strategy:**
  - **MXU0:** Handles the top 512 rows (2x4 blocks).
  - **MXU1:** Handles the bottom 512 rows (2x4 blocks).
- **Contracting Summation:** Inner dimension K=512 requires **two sequential passes** (256x2) per physical block with internal accumulation.

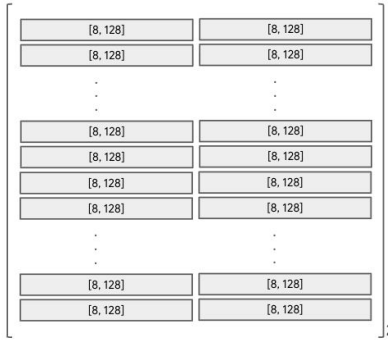


Filling the Grid (MXU Stationary K Phase)

Loading a single physical **[256, 256] Key (K) block** into the MXU stationary buffer via LLO.

- **Physical Block Context**: The compiler isolates a [256, 256] slice of the logical K matrix to "pin" into the systolic array's stationery registers.
- **Tiling & Packing**: This block is composed of 64 tiles (8x128). 32-bit f32 values from VMEM are down-cast and packed into bf16 using vpack to double the density of each 4KB vector register.
- **The 64-Push Sequence**: To fully saturate the 256-wide and 256-high physical grid, the compiler executes 32 vertical steps and 2 horizontal steps (32x2 = 64 pushes).

```
0x15 : {
 %v1579_v4 = vpack.c.bf16 %v34239_v2, %v34235_v1 ;; // Cast f32 -> bf16 and pack 2 per slot
 %v11218_v3 = vpack.c.bf16 %v34240_v3, %v34236_v12 ;; // Parallel pack for the next tile
}
0x17 : {
 %2091 = vmatprep.subr.bf16.mxu0 %v1579_v4 ;; // Set sublane pointers for MXU0 [0-255 range]
 %v34243_v8 = vld [vmem:[%s57053_s1 + $0x10]] ;; // Overlap: Load next Q-tile from VMEM
}
0x18 : {
 %2093 = vmatpush1.bf16.xpose.msra.mxu0 %v1577_v9 ;; // Push to MXU0 with Hardware Transpose
 %v34247_v14 = vld [vmem:[%s57053_s1 + $0x18]] ;; // Overlap: Keep the VPU pipeline full
}
```

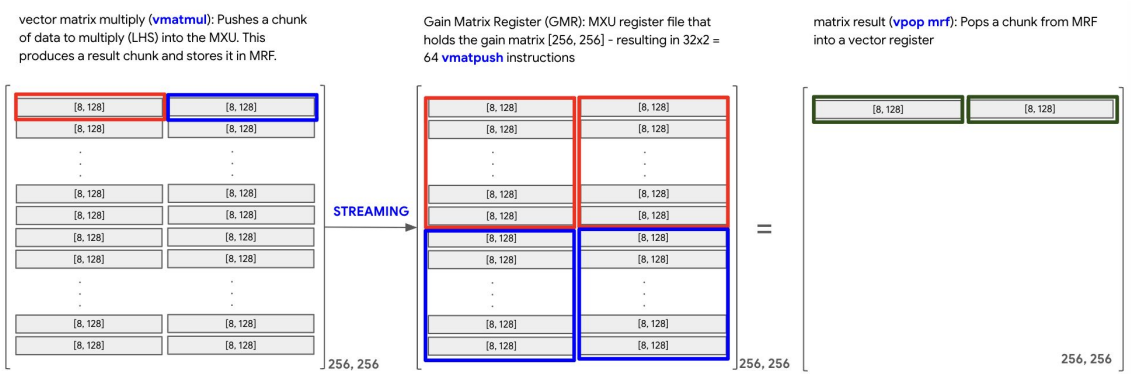


Streaming & Accumulation (Active Compute)

Objective: Streaming the **Query (Q)** tiles through the systolic array to accumulate the 512-wide dot product against stationary **Key (K)** blocks.

- **The Streaming Pass:** Query blocks are "pumped" through the stationary Key grid (256x256) at a rate of one 8x128 tile per cycle. To cover the full width and height of a physical 256x256block, the hardware streams 32 vertical x 2 horizontal tiles.
- **Two-Pass Accumulation (K=512)**:
  - **Pass 1 (Indices 0–255)**: The first physical block of K is stationary; the first block of Q streams through to generate the initial partial sum in the Matrix Result Folder (MRF).
  - **Pass 2 (Indices 256–511)**: The second physical block of K is loaded; the corresponding Q tiles stream through using the .mubr modifier to accumulate directly into the existing 32-bit results.

MXU Micro-Tiling and Compiler Instructions (256×256)



Instruction	Action & Flow	Count for 256×256 Block
vmatpush	Loads Y[8,128] chunks from VMem into the GSF/GMR. This 64 (32 x 2) instruction sequence must be completed before vmatmul starts	64
vmatmul	Pushes the X[8,128] chunks horizontally, driving the MAC array. The instruction is stretched/broadcast across the full 256-wide N-dimension of the MXU, accumulating partial sums for an entire 8×256 output strip against a K=128 slice.	128
vpopmrf	Retrieves the final Z[8,128] accumulated output chunk from the MRF. The result is immediately accumulated (vadd.f32) into the VMem array.	128

VPU Post Processing - MAX

Objective: Extracting logits and performing parallel row-max reduction.

- **Result Pop:** Finished 32-bit float logits are "popped" from the MXU Result Folder (MRF) into VPU registers.
- **Parallel Masking:** The VPU applies the attention mask (setting invalid entries to -1e^38) and finds the row max in the same bundle.
- **LLO IR Highlights:**
  - **vpop.f32.mrb**: Extracts the accumulated f32 tile from the Matrix Unit.
  - **vsel %vm, %v, -1e+38**: Applies the mask based on the pre-loaded vector mask.
  - **vmax.f32**: Updates the running row-max for the Softmax stabilization.
  - **vrot.slane**: Finalizes the max across the 8 sublanes to get the true row-wise maximum.

Softmax Fusion Pipeline

The TPU hardware is natively optimized for base-2 exponentiation (vpow2.f32). To compute the natural exponential (e^x) required for Softmax, XLA performs a change of base. The Mathematical Transformation: $e^x \longrightarrow 2^{x \log\_2(e)}$

- **$e^x = 2^{x\log\_2(e)}$**
- **log_2(e) = 1.442695**

The Softmax pipeline executes this in three distinct stages:

1. **Subtraction:** x = logit - max (
2. **Scaling:** y = x (1.442695)
3. **Exponentiation:** result = 2^y

Cycle Phase	Instruction	Purpose
Input	vld	Pull logit tile from VMEM.
Shift	vsub	logit−max (Numerical stability).
Scale	vmul	Multiply by 1.442695 (log2e).
Math	vpow2	Compute 2^y in hardware transcendental unit.
Collect	vpop.eup	Retrieve result from the Execution Unit Pipeline.
Reduce	vadd	Accumulate for the Softmax denominator.

The VLIW Pipeline in Motion

Cycle Phase	Instruction	Purpose
Input	vld	Pull logit tile from VMEM.
Shift	vsub	logit−max (Numerical stability).
Scale	vmul	Multiply by 1.442695 (log2e).
Math	vpow2	Compute 2^y in hardware transcendental unit.
Collect	vpop.eup	Retrieve result from the Execution Unit Pipeline.
Reduce	vadd	Accumulate for the Softmax denominator.

0x11 : { %v16570_v34 = vpop.eup %16569 ;; %v15538_v51 = vld [vmem:[%s24331_s0 + $0x3c0] sm:$0xff] ;; %16589 = vpow2.f32 %v161_v43 ;; %v191_v53 = vmul.f32 1.442695, %v187_v44 ;; %v217_v54 = vsub.f32 %v15537_v46, %v18657_v2 }

0x12 : { %v16572_v36 = vpop.eup %16571 ;; %v15539_v56 = vld [vmem:[%s24331_s0 + $0x400] sm:$0xff] ;; %16591 = vpow2.f32 %v176_v48 ;; %v206_v58 = vmul.f32 1.442695, %v202_v49 ;; %v232_v59 = vsub.f32 %v15538_v51, %v18657_v2 }

0x13 : { %v16574_v39 = **vpop.eup** %16573 ;; %v31_v40 = **vadd.f32** %v16572_v36, %v16570_v34 ;; %v15540_v61 = **vld** [vmem:[%s24331_s0 + $0x440] sm:$0xff] ;; %16593 = **vpow2.f32** %v191_v53 ;; %v221_v63 = **vmul.f32** 1.442695, %v217_v54 ;; %v247_v0 = **vsub.f32** %v15539_v56, %v18657_v2 }

0x14 : { %v16576_v42 = vpop.eup %16575 ;; %v15541_v3 = vld [vmem:[%s24331_s0 + $0x480] sm:$0xff] ;; %16595 = vpow2.f32 %v206_v58 ;; %v236_v5 = vmul.f32 1.442695, %v232_v59 ;; %v262_v6 = vsub.f32 %v15540_v61, %v18657_v2 }

0x15 : { %v46_v45 = vadd.f32 %v16574_v39, %v31_v40 ;; %v16578_v47 = vpop.eup %16577 ;; %v15542_v8 = vld [vmem:[%s24331_s0 + $0x4c0] sm:$0xff] ;; %16597 = vpow2.f32 %v221_v63 ;; %v251_v10 = vmul.f32 1.442695, %v247_v0 ;; %v277_v11 = vsub.f32 %v15541_v3, %v18657_v2 }

In a single cycle (**like bundle 0x13**), the TPU isn't just doing one task; it is simultaneously handling six different stages of the Softmax calculation for six different data tiles.

Pacchetto LLO Bundle Visualizer

What it is: Pacchetto is a visualization tool for analyzing **LLO (Low-Level Optimizer)** bundles produced by XLA for TPUs. It uses the Perfetto trace viewer to show a detailed timeline of a model's execution.

Data Flow: The tool visualizes how data flows through the TPU's memory and compute cores.

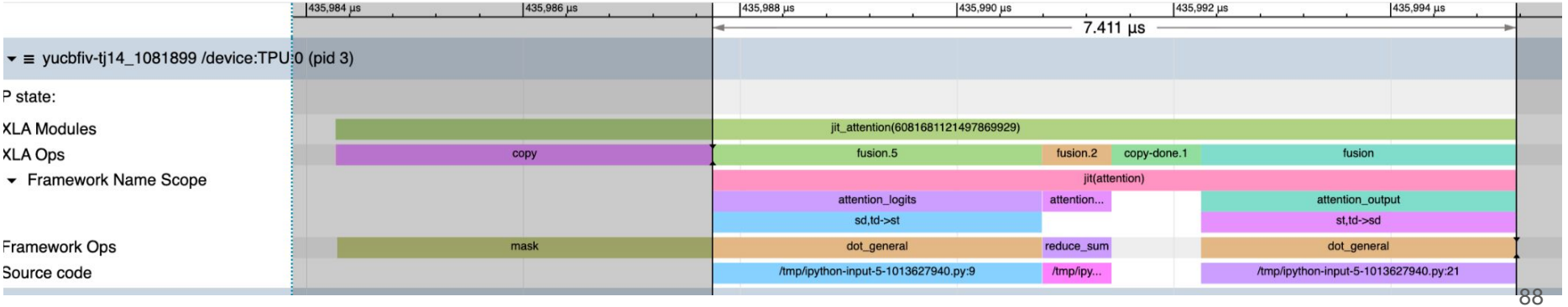
Resource Utilization: It helps analyze the utilization of key hardware resources, such as the **TensorCore**, including its **MXU** (Matrix-Vector Unit) and **ALU** (Arithmetic Logic Unit). It also provides insights into **register pressure**.

Key Use Case: Pacchetto is useful for identifying performance bottlenecks within complex, low-level kernels like those used for Flash Attention.

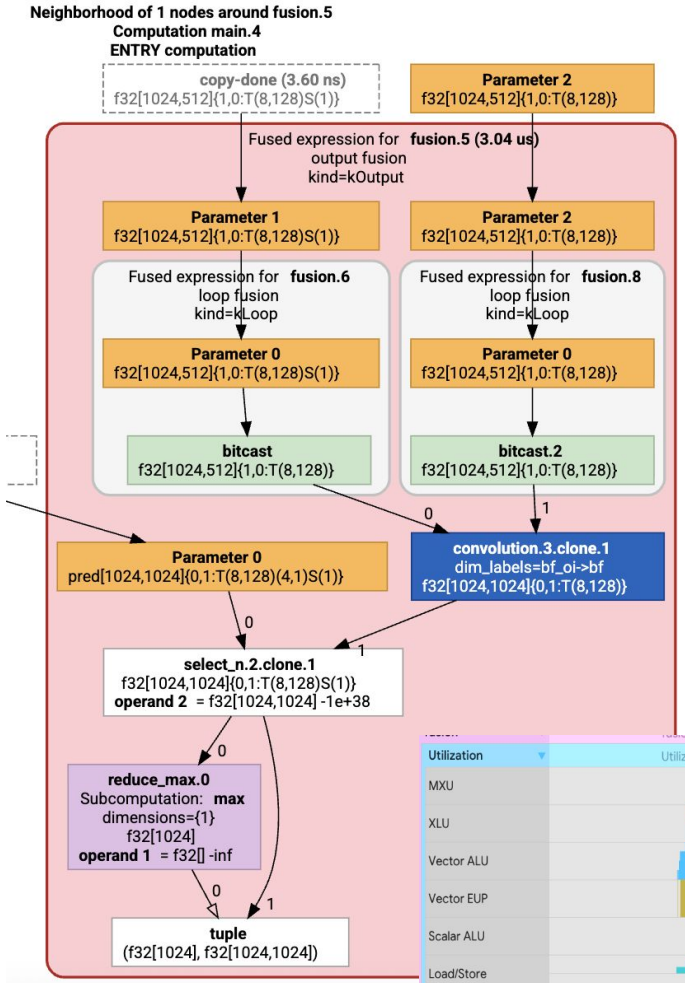
Basic Attention

```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)
```

```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
```



Attention Logits Fusion



Total Time Avg:
3.04 us

FLOPS utilization:
34.43%

HBM bandwidth utilization:
18.73%

On-Chip Read bandwidth utilization:
3.55%

On-Chip Write bandwidth utilization:
6.46%

FLOP rate (per core):
354.22 TFLOP/s

bf16 normalized FLOP rate (per core):
354.22 TFLOP/s

HBM bandwidth (per core):
690.49 GB/s

On-chip Read bandwidth (per core):
1.04 TB/s

On-chip Write bandwidth (per core):
1.38 TB/s

XLA expression:
%fusion.5 = (f32[1024]

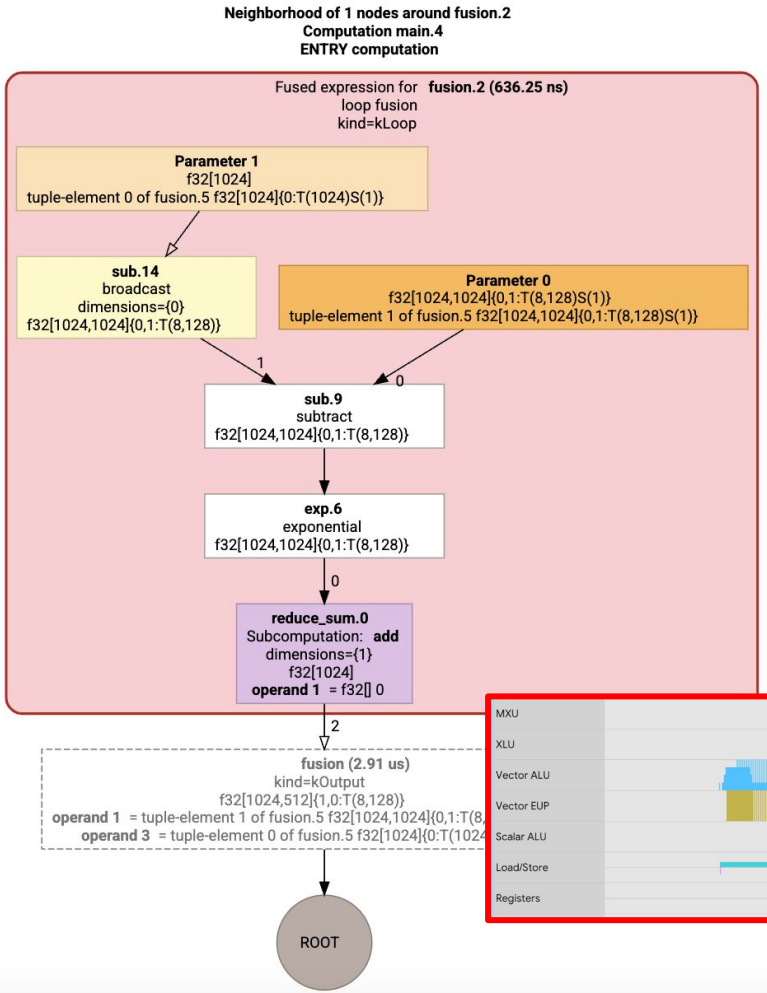
```
fused_computation.7 {
  param_0.20 = pred[1024,1024]{0,1:T(8,128)(4,1)S(1)} parameter(0)
  param_1.22 = f32[1024,512]{1,0:T(8,128)S(1)} parameter(1)
  fusion.6 = f32[1024,512]{1,0:T(8,128)} fusion(param_1.22), kind=kLoop, calls=bitcast_fusion
  param_2.14 = f32[1024,512]{1,0:T(8,128)} parameter(2)
  fusion.8 = f32[1024,512]{1,0:T(8,128)} fusion(param_2.14), kind=kLoop, calls=bitcast_fusion.2
  convolution.3.clone.1 = f32[1024,1024]{0,1:T(8,128)} convolution(fusion.6, fusion.8), dim_labels=bf_oi->bf
  constant.7.clone.1 = f32[]{:T(128)} constant(-1e+38)
  broadcast_in_dim.2.clone.1 = f32[1024,1024]{0,1:T(8,128)} broadcast(constant.7.clone.1), dimensions={}
  select_n.2.clone.1 = f32[1024,1024]{0,1:T(8,128)S(1)} select(param_0.20, convolution.3.clone.1, broadcast_in_dim.2.clone.1)
  constant.9 = f32[]{:T(128)} constant(-inf)
  reduce_max.0 = f32[1024]{0:T(1024)S(1)} reduce(select_n.2.clone.1, constant.9), dimensions={1}, to_apply=region_0.2
  ROOT tuple = (f32[1024]{0:T(1024)S(1)}, f32[1024,1024]{0,1:T(8,128)S(1)}) tuple(reduce_max.0, select_n.2.clone.1)
}
```

XLA expression:

%fusion.5 = (f32[1024]{0:T(1024)S(1)}, f32[1024,1024]{0,1:T(8,128)S(1)}) fusion(pred[1024,1024]{0,1:T(8,128)(4,1)S(1)} %copy, f32[1024,512]{1,0:T(8,128)S(1)} %copy-done, f32[1024,512]{1,0:T(8,128)} %k.1), kind=kOutput, calls=%fused_computation.7

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

Softmax Fusion



Total Time Avg:
636.25 ns

FLOPS utilization:
0.32%

On-Chip Read bandwidth utilization:
22.61%

On-Chip Write bandwidth utilization:
0.03%

FLOP rate (per core):
3.29 TFLOP/s

bf16 normalized FLOP rate (per core):
3.29 TFLOP/s

HBM bandwidth (per core):
0.00 B/s

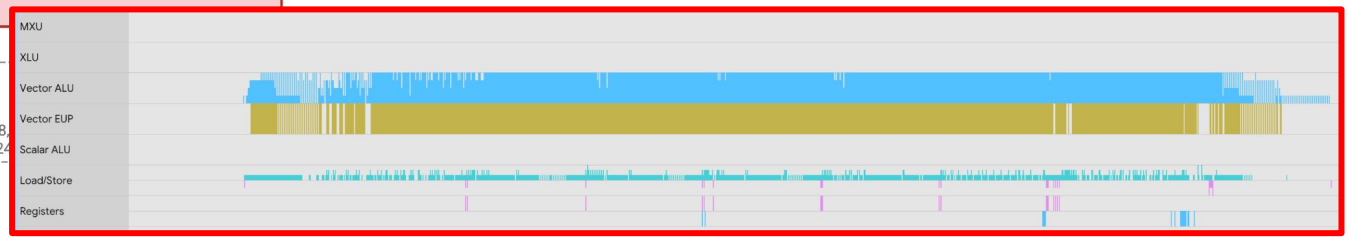
On-chip Read bandwidth (per core):
6.60 TB/s

On-chip Write bandwidth (per core):
6.44 GB/s

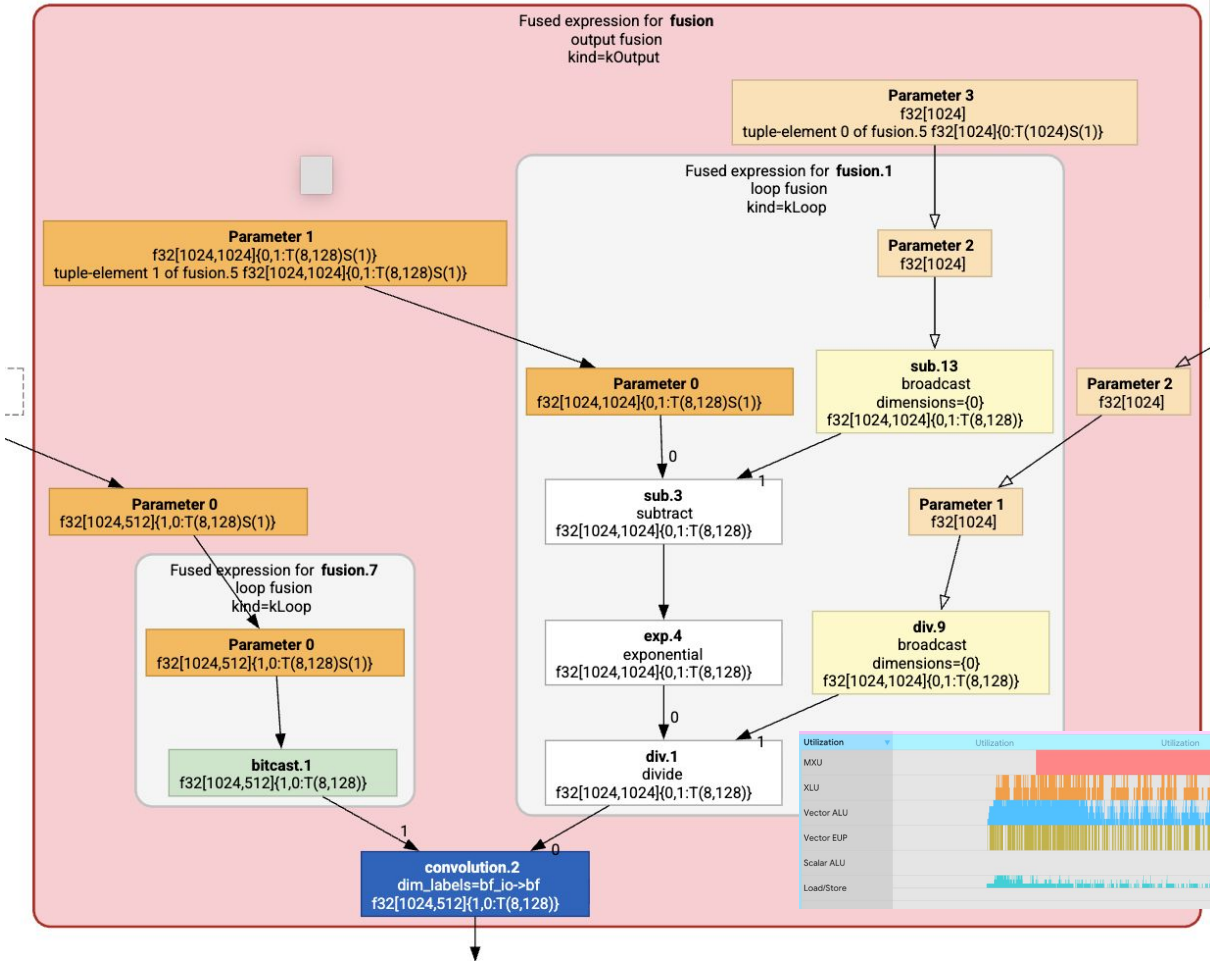
```
fused_computation.3 {
  param_0.17 = f32[1024,1024]{0,1:T(8,128)S(1)} parameter(0)
  param_1.19 = f32[1024]{0:T(1024)S(1)} parameter(1)
  sub.14 = f32[1024,1024]{0,1:T(8,128)} broadcast(param_1.19), dimensions={0}
  sub.9 = f32[1024,1024]{0,1:T(8,128)} subtract(param_0.17, sub.14)
  exp.6 = f32[1024,1024]{0,1:T(8,128)} exponential(sub.9)
  constant.8 = f32[]{:T(128)} constant(0)
  ROOT reduce_sum.0 = f32[1024]{0:T(1024)S(1)} reduce(exp.6, constant.8), dimension
}
```

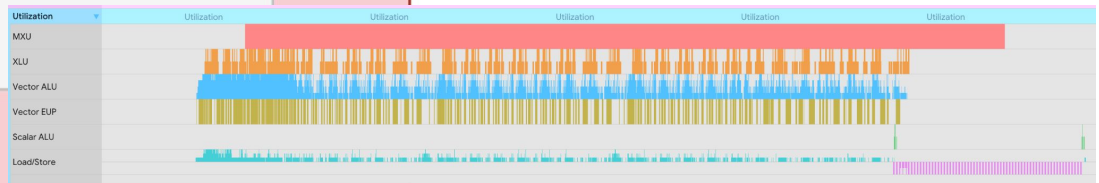
XLA expression:

%fusion.2 = f32[1024]{0:T(1024)S(1)} fusion(f32[1024,1024]{0,1:T(8,128)S(1)} %get-tuple-element.1, f32[1024]{0:T(1024)S(1)} %get-tuple-element), kind=kLoop, calls=%fused_computation.3



Attention Output





Total Time Avg:
2.91 us

FLOPS utilization:
35.98%

HBM bandwidth utilization:
19.58%

On-Chip Read bandwidth utilization:
7.43%

FLOP rate (per core):
370.17 TFLOP/s

bf16 normalized FLOP rate (per core):
370.17 TFLOP/s

HBM bandwidth (per core):
721.57 GB/s

On-chip Read bandwidth (per core):
2.17 TB/s

Simple Attention Fusions Summary

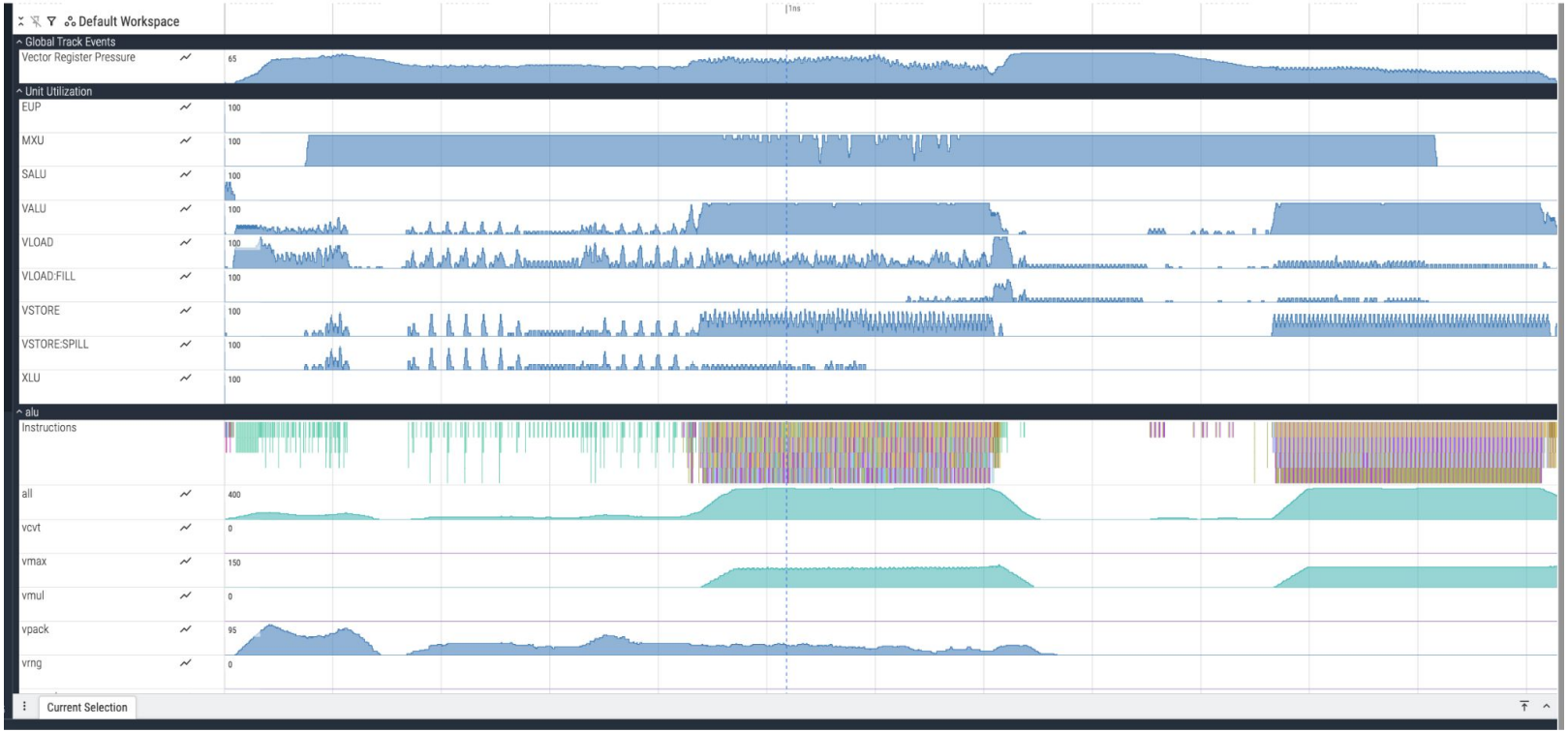
TPU Ironwood

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.

Attention - Pacchetto Trace

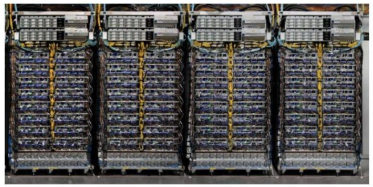


Sharding, JIT & shard_map

TPU Interconnects: ICI and Torus Topology

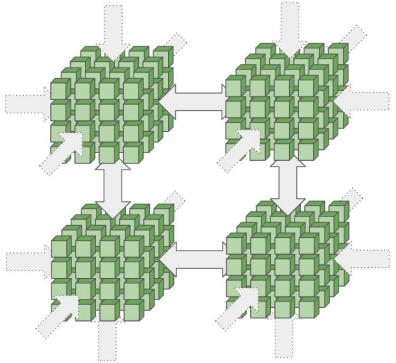
- **Inter-Chip Interconnect (ICI):** The TPU cluster uses dedicated, high-speed **ICI** links to connect individual chips, enabling massive scale-out of computation.

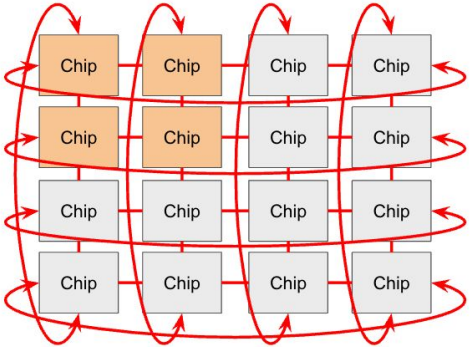
- **Torus Network Topology:** The networking structure forms a 2D or 3D **torus**. This geometry ensures low-latency communication by connecting every TPU chip to its **nearest neighbors** in a grid-like structure, with connections wrapping around the edges.



Each rack corresponds to 4x4x4 TPUv4s

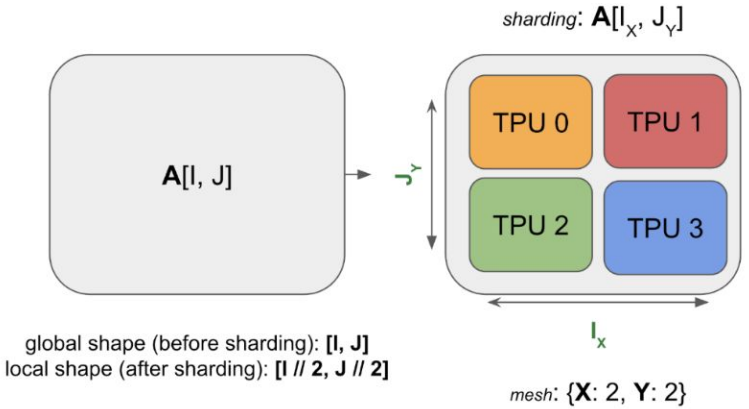
These can be configured into arbitrary 4ix4jx4k toruses via optical interconnects





Sharding Arrays: Distributing Tensors Across Devices

- **Device Mesh Partitioning:** The array A is split across a predefined hardware topology, or **Mesh**, often a 2D or 3D grid of devices.
- **Global vs. Local Shape:** The original **Global Shape** [I, J] is evenly divided by the size of the mesh axes to determine the **Local Shape** [I // 2, J // 2] stored on each device.
- **Sharding Annotation:** Notation like A[I_{X}, J_{Y}] specifies that the **I** dimension is sharded over the **X** axis of the mesh, and **J** is sharded over the **Y** axis.
- **Parallel Compute:** Each device performs computations only on its local tensor slice, significantly reducing per-device memory consumption and latency.



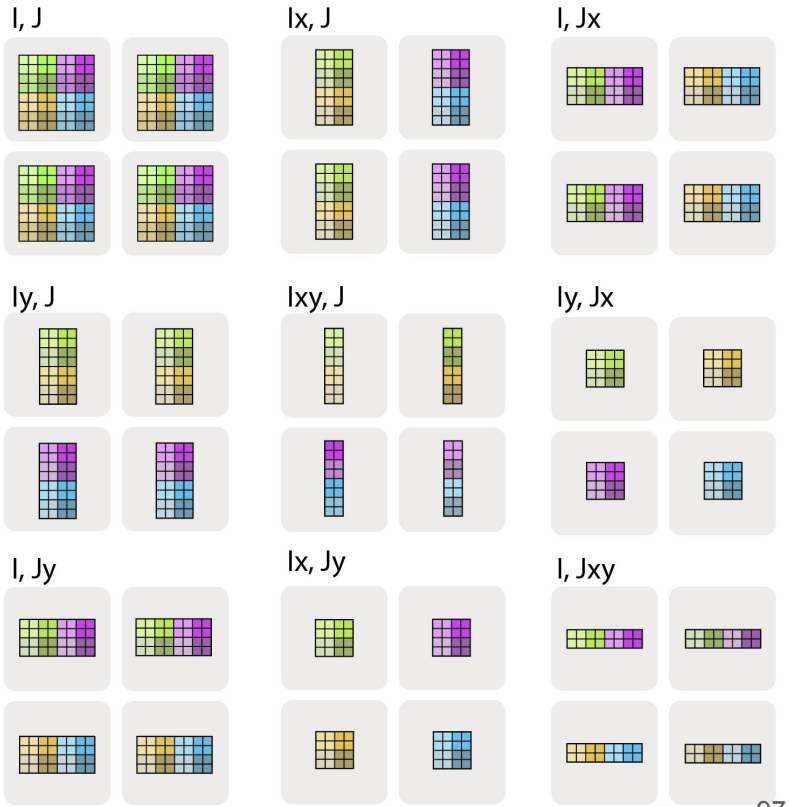
https://jax-ml.github.io/scaling-book/sharding/

Defining Distributed Layout: Mesh and Spec

Device Mesh: Defines the physical hardware topology, assigning names (e.g., X, Y) to the axes of the device grid (e.g., 2x4 TPU cluster).

Sharding Specification (Spec): Maps these physical mesh names to the dimensions of a specific tensor, indicating which dimension is sharded and along which mesh axis (e.g., 'X', 'Y'), or None for replication).

Abridged Notation: The concise format **A**[I_X, J_Y, K] denotes that array dimension Il is sharded across mesh axis X, dimension J is sharded across Y, and dimension K is fully replicated.



https://jax-ml.github.io/scaling-book/sharding/

Automatic Parallelism with GSPMD

- **JIT Orchestration:** JAX's Just-In-Time (JIT) compilation can orchestrate automatic parallelism via the **GSPMD** (General Single-Program, Multiple-Data) backend.
- **User Annotation Required:** GSPMD relies entirely on the user specifying the desired input and output sharding (layout) annotations for the tensors.
- **Compiler-Driven Parallelism:** Given these sharding specifications, the XLA compiler automatically partitions the entire computation graph and inserts the required cross-device communication primitives (like All-Reduce or All-Gather).
- **High-Level Abstraction:** This provides a compiler-driven approach to parallelism, contrasting with the more explicit, manual control offered by lower-level primitives like shard_map.

Explicit Parallelism with shard_map

Manual Control: shard_map provides explicit, manual control over **SPMD** (Single-Program, Multiple-Data) parallelism, giving the user fine-grained control over computation and communication.

Topology Definition: It relies on a jax.sharding.Mesh to define the logical topology of physical devices (CPUs/GPUs/TPUs) using named axes (e.g., 'data', 'model').

Sharding Specification: Uses jax.sharding.PartitionSpec to describe precisely how tensor dimensions should be **sharded (split)** or **replicated** across these Mesh axes.

Communication Requirement: Unlike GSPMD, shard_map requires **explicit specification** of cross-device communication primitives (e.g., using jax.lax.psum for summing partial results or jax.lax.all_gather for data collection).

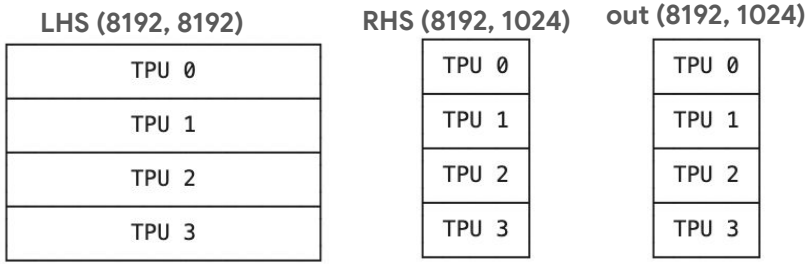
https://docs.jax.dev/en/latest/notebooks/shard_map.html

Shard_map Example (matmul)

Partitioning: Both LHS and RHS are **row-sharded** - (**P('i', None)**).

Communication: **jax.lax.all_gather** is applied to the RHS block. Every device receive the **complete RHS matrix**, enabling the local matrix multiplication.

Computation: Each device performs the dense multiplication (hs_block @ full_rhs), calculating its corresponding **row-partition** of the final output, which is also sharded **P('i', None)**

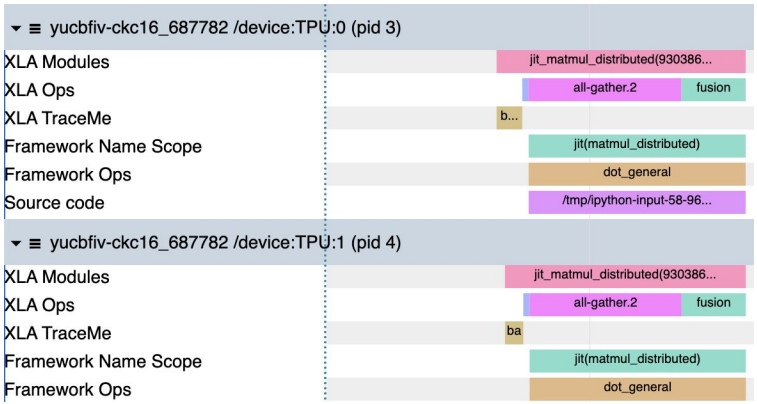



```python
import jax
import jax.numpy as jnp
from jax.sharding import Mesh, PartitionSpec as P
from jax.sharding import NamedSharding

mesh = Mesh(jax.devices()[:4], ('i',))
jax.set_mesh(mesh)

def device_put(x, pspec):
  return jax.device_put(x, NamedSharding(mesh, pspec))

lhs_spec = P('i', None)
lhs = device_put(jax.random.normal(jax.random.key(0), (8192, 8192)), lhs_spec)

rhs_spec = P('i', None)
rhs = device_put(jax.random.normal(jax.random.key(1), (8192, 1024)), rhs_spec)

@jax.jit
@jax.shard_map(mesh=mesh, in_specs=(lhs_spec, rhs_spec),
               out_specs=rhs_spec)
def matmul_allgather(lhs_block, rhs_block):
  rhs = jax.lax.all_gather(rhs_block, 'i', tiled=True)
  return lhs_block @ rhs

out = matmul_allgather(lhs, rhs)
```

Example: JIT vs Shard_map

- The **XLA compiler** injects an **All-Gather** on the RHS because the input and output shardings require that the **full** RHS matrix be present on every device to compute the final, sharded result correctly.

(Only two devices are shown here)




```python
import jax
import jax.numpy as jnp
from jax.sharding import Mesh, PartitionSpec as P
from jax.sharding import NamedSharding

mesh = Mesh(jax.devices()[:4], ('i',))
jax.set_mesh(mesh)

def device_put(x, pspec):
  return jax.device_put(x, NamedSharding(mesh, pspec))

lhs_spec = P('i', None)
lhs = device_put(jax.random.normal(jax.random.key(0), (8192, 8192)), lhs_spec)

rhs_spec = P('i', None)
rhs = device_put(jax.random.normal(jax.random.key(1), (8192, 1024)), rhs_spec)

@jax.jit
@jax.shard_map(mesh=mesh, in_specs=(lhs_spec, rhs_spec),
               out_specs=rhs_spec)
def matmul_allgather(lhs_block, rhs_block):
  rhs = jax.lax.all_gather(rhs_block, 'i', tiled=True)
  return lhs_block @ rhs

out = matmul_allgather(lhs, rhs)

@jax.jit(in_shardings=(NamedSharding(mesh, lhs_spec),
                       NamedSharding(mesh, rhs_spec)),
         out_shardings=NamedSharding(mesh, rhs_spec))
def matmul_distributed(lhs, rhs):
  return lhs @ rhs

out = matmul_distributed(lhs, rhs)
```

StableHLO: Preserving Sharding Logic

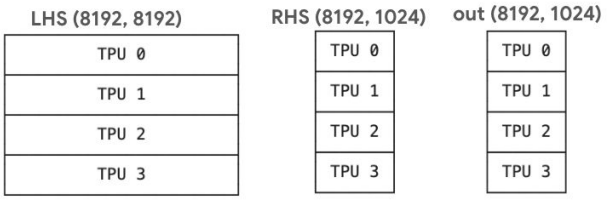
Metadata Preservation: JAX lowers high-level **PartitionSpec('i', None)** into explicit **#sdy.sharding** attributes on StableHLO input arguments, ensuring the logical mesh topology is baked into the function signature.

Automated Partitioning: The **num_partitions = 4** attribute forces the compiler to respect physical device boundaries and auto-inject collectives like **all-gather** to satisfy sharding constraints.

```python
# Create mesh
mesh = Mesh(jax.devices()[:4], ('i',))
jax.set_mesh(mesh)

def device_put(x, pspec):
    return jax.device_put(x, NamedSharding(mesh, pspec))

# Define Sharding Specifications
lhs_spec = P('i', None)
lhs = device_put(jax.random.normal(jax.random.key(0),
                 (8192, 8192)), lhs_spec)

rhs_spec = P('i', None)
rhs = device_put(jax.random.normal(jax.random.key(1),
                 (8192, 1024)), rhs_spec)

# Compiler automatically injects all-gather
@jax.jit(in_shardings=(NamedSharding(mesh, lhs_spec),
                       NamedSharding(mesh, rhs_spec)),
         out_shardings=NamedSharding(mesh, rhs_spec))
def matmul_distributed(lhs, rhs):
    return lhs @ rhs

# Check arg shardings in stable hlo
matmul_auto_lower = jax.jit(matmul_distributed).lower(lhs, rhs)
print(matmul_auto_lower.as_text())
```

```
module @jit_matmul_distributed attributes {mhlo.num_partitions = 4 : i32, mhlo.num_replicas = 1 : i32} {
  sdy.mesh @mesh = <["i"=4]>
  func.func public @main(%arg0: tensor<8192x8192xf32> {sdy.sharding = #sdy.sharding<@mesh, [{"i"}, {}]>}, %arg1: tensor<8192x1024xf32> {sdy.sharding
= #sdy.sharding<@mesh, [{"i"}, {}]>}) -> (tensor<8192x1024xf32> {jax.result_info = "result"}) {
    %0 = call @matmul_distributed(%arg0, %arg1) : (tensor<8192x8192xf32>, tensor<8192x1024xf32>) -> tensor<8192x1024xf32>
    return %0 : tensor<8192x1024xf32>
  }
  func.func private @matmul_distributed(%arg0: tensor<8192x8192xf32>, %arg1: tensor<8192x1024xf32>) -> tensor<8192x1024xf32> {
    %0 = sdy.sharding_constraint %arg0 <@mesh, [{"i"}, {}]> : tensor<8192x8192xf32>
    %1 = sdy.sharding_constraint %arg1 <@mesh, [{"i"}, {}]> : tensor<8192x1024xf32>
    %2 = stablehlo.dot_general %0, %1, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<8192x8192xf32>, tensor<8192x1024xf32>) ->
tensor<8192x1024xf32>
    %3 = sdy.sharding_constraint %2 <@mesh, [{"i"}, {}]> : tensor<8192x1024xf32>
    return %3 : tensor<8192x1024xf32>
  }
}
```

Optimized HLO: Collective Injection




```
%bitcast_fusion (bitcast_input: f32[2048,8192]) -> f32[2048,8192] {
  %bitcast_input = f32[2048,8192]{1,0:T(8,128)} parameter(0)
  ROOT %bitcast = f32[2048,8192]{1,0:T(8,128)} bitcast(%bitcast_input)
}

%copy_fusion (input: bf16[8192,1024]) -> bf16[8192,1024] {
  %input = bf16[8192,1024]{1,0:T(16,128)(2,1)S(1)} parameter(0)
  ROOT %copy = bf16[8192,1024]{1,0:T(8,128)(2,1)} copy(%input)
}

%bitcast_fusion.1 (bitcast_input.1: bf16[8192,1024]) -> bf16[8192,1024] {
  %bitcast_input.1 = bf16[8192,1024]{1,0:T(16,128)(2,1)S(1)} parameter(0)
  %fusion.3 = bf16[8192,1024]{1,0:T(8,128)(2,1)} fusion(%bitcast_input.1), kind=kLoop,
              output_to_operand_aliasing={{}: (0, {})}, calls=%copy_fusion
  ROOT %bitcast.1 = bf16[8192,1024]{1,0:T(8,128)(2,1)} bitcast(%fusion.3)
}

%fused_computation (param_0: f32[2048,8192], param_1: bf16[8192,1024]) -> f32[2048,1024] {
  %param_0 = f32[2048,8192]{1,0:T(8,128)} parameter(0)
  %fusion.1 = f32[2048,8192]{1,0:T(8,128)} fusion(%param_0), kind=kLoop, calls=%bitcast_fusion
  %param_1 = bf16[8192,1024]{1,0:T(16,128)(2,1)S(1)} parameter(1)
  %fusion.2 = bf16[8192,1024]{1,0:T(8,128)(2,1)} fusion(%param_1), kind=kLoop, calls=%bitcast_fusion.1
  ROOT %convolution.1 = f32[2048,1024]{1,0:T(8,128)} convolution(%fusion.1, %fusion.2), dim_labels=bf_io->bf, ...
}

ENTRY %main.1_spmd (param: f32[2048,8192], param.1: f32[2048,1024]) -> f32[2048,1024] {
  %param.1 = f32[2048,1024]{1,0:T(8,128)} parameter(1), sharding={devices=[4,1]<=[4]}, ...
  %param = f32[2048,8192]{1,0:T(8,128)} parameter(0), sharding={devices=[4,1]<=[4]}, ...
  %convert = bf16[2048,1024]{1,0:T(16,128)(2,1)S(1)} convert(%param.1), ...
  %all-gather.2 = bf16[8192,1024]{1,0:T(16,128)(2,1)S(1)} all-gather(%convert), channel_id=2, replica_groups=[1,4]<=[4],
    dimensions={0}, use_global_device_ids=true, "collective_algorithm_config":{"emitter":"1DAllGatherNonMajorDimHierarchical", ..}
  ROOT %fusion = f32[2048,1024]{1,0:T(8,128)} fusion(%param, %all-gather.2), kind=kOutput, calls=%fused_computation, ...
}
```

Sharded Parameters: The **%main.1_spmd** entry point reflects local shard shapes (2048x8192) and (2048x1024) rather than global dimensions, preserving the sharding intent from the Python mesh.

Bandwidth-Optimized All-Gather: The compiler invokes an **all-gather** on the RHS, temporarily shifting the tiling from **T(8, 128) to T(16, 128)** to maximize communication throughput across the TPU Inter-Connect.

MXU Layout Alignment: Before the MatMul, a bitcast restores the RHS to T(8, 128) to align with the LHS layout and the physical 8-row sublane height of the (MXU).

Manual & Automated Sharding

- **Logic & Shapes:** Manual (shard_map) operates inside **sdy.manual_computation** on local shapes (2048), forcing the user to think at the device level. Automated (jit) uses **sdy.sharding_constraint** on global shapes (8192), allowing the compiler to manage the distributed logic.

- **Collective Control:** In the manual version, the user must explicitly write the jax.lax.all_gather to ensure data availability. In the automated version, the compiler automatically injects necessary collectives during lowering to satisfy the sharding requirements.

```
// Compiler – Auto injection allgather
func.func private @matmul_distributed(%arg0: tensor<8192x8192xf32>, %arg1:
tensor<8192x1024xf32>) -> tensor<8192x1024xf32> {
    %0 = sdy.sharding_constraint %arg0 <@mesh, [{"i"}, {}]> :
    tensor<8192x8192xf32>
    %1 = sdy.sharding_constraint %arg1 <@mesh, [{"i"}, {}]> :
    tensor<8192x1024xf32>
    %2 = stablehlo.dot_general %0, %1, contracting_dims = [1] x [0], precision
    = [DEFAULT, DEFAULT] : (tensor<8192x8192xf32>, tensor<8192x1024xf32>) ->
    tensor<8192x1024xf32>
    %3 = sdy.sharding_constraint %2 <@mesh, [{"i"}, {}]> : tensor<8192x1024xf32>
    return %3 : tensor<8192x1024xf32>
  }

// Shard_map – manual explicit all-gather
func.func private @matmul_allgather(%arg0: tensor<8192x8192xf32>, %arg1:
tensor<8192x1024xf32>) -> tensor<8192x1024xf32> {
    %0 = sdy.manual_computation(%arg0, %arg1) in_shardings=[<@mesh, [{"i"}, {}]
    >, <@mesh, [{"i"}, {}]>] out_shardings=[<@mesh, [{"i"}, {}]>] manual_axes=
    {"i"} (%arg2: tensor<2048x8192xf32>, %arg3: tensor<2048x1024xf32>) {
      %1 = "stablehlo.all_gather"(%arg3) <{all_gather_dim = 0 : i64,
      channel_handle = #stablehlo.channel_handle<handle = 1, type = 1>,
      replica_groups = dense<[[0, 1, 2, 3]]> : tensor<1x4xi64>,
      use_global_device_ids}> : (tensor<2048x1024xf32>) -> tensor<8192x1024xf32>
      %2 = stablehlo.dot_general %arg2, %1, contracting_dims = [1] x [0],
      precision = [DEFAULT, DEFAULT] : (tensor<2048x8192xf32>,
      tensor<8192x1024xf32>) -> tensor<2048x1024xf32>
      sdy.return %2 : tensor<2048x1024xf32>
    } : (tensor<8192x8192xf32>, tensor<8192x1024xf32>) -> tensor<8192x1024xf32>
    return %0 : tensor<8192x1024xf32>
  }
```

SPMD - Single Program, Multiple Data

Unified Execution Model: When you invoke **@jax.jit,** JAX compiles a **single global program** that is broadcast and launched across all devices in the mesh simultaneously.

Logical vs. Physical View: While the user writes a single logical MatMul **(lhs @ rhs)**, the compiler lowers it into a physical SPMD (Single Program, Multiple Data) version where each device executes the same instructions on its local data shard.

Injected Coordination: The "Single Program" includes both the local computation (the MatMul) and the necessary coordination (collectives like **all-gather**) injected by the compiler to ensure data is moved between devices at the right moment.

Hardware Synchronization: On the TPU, this program is loaded as a single binary that runs asynchronously across all cores, using the **#sdy.sharding and num_partitions = 4** attributes to maintain synchronization and data consistency.

Exercises

1. **GPT Tensor Parallel Scaling:** Transition from a (8, 1) mesh to a (4, 2) configuration to observe how large weight matrices are physically split across two devices to reduce memory footprint.
2. **Collective Analysis:** Use XLA instruction traces to examine the automatic injection of All-Gather and Reduce-Scatter operations required for hybrid data/model parallelism.
3. **Roofline Profiling:** Utilize XProf or Pacchetto to identify if the Multi-Head Attention block is Compute-Bound (limited by MXU TFLOPs) or Memory-Bound (limited by HBM bandwidth).
4. **HLO Optimization Audit:** Inspect the Optimized HLO graph to see how the compiler collapses redundant reshapes and fuses operations into high-speed fused kernels.
5. **VLIW Bundle Inspection:** Analyze the Low-Level Operations (LLO) to see how the compiler packs memory loads and matrix math into 512-bit bundles to keep all 65,536 MAC units saturated.

Questions ?