November 6, 2025

6.S894 Accelerated Computing

Lecture 10: Beyond

Jonathan Ragan-Kelley

Conventional processor

Why yes,
I am still
basically a PDP-11

Throughput processor

You will **rewrite all your code** in CUDA . . .
and pray I don't alter the deal any further!

A **throughput-oriented processor** exploits parallelism & static control for efficiency

1	**Explicit parallelism** over implicit ILP	Multicore Wide issue / exec (SIMD)
2	Amortize **control overhead**	SIMD VLIW
3	**Explicit concurrency** over speculation	Multithreading Overlap mem & compute

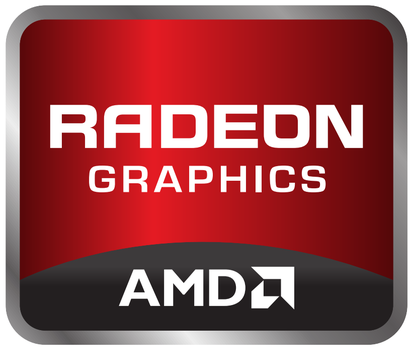
A **throughput-oriented processor** exploits arithmetic intensity & complex memory model

4	**Complex instructions** amortize operand fetch	Matrix Multiply Accumulate (Tensor Core)
5	**Wide memory** interfaces	GDDR, HBM Highly-banked SRAM
6	**Software-managed** memory hierarchy	Scratchpad vs. cache Explicit DMA

So far: NVIDIA GPUs
(+ modern x86)

Today: Other common
accelerators

GPUs beyond NVIDIA



AMD
RDNA

AMD
CDNA

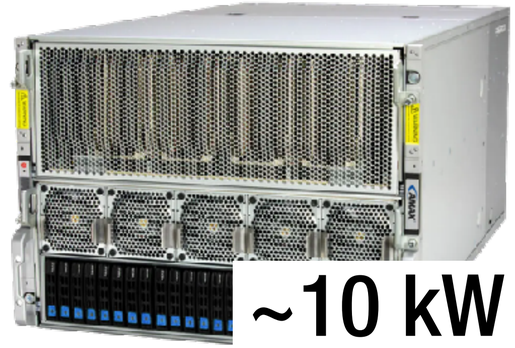
Intel
Gen/Arc

Intel
DC GPU

Qualcomm
Adreno

Arm Mali

GPUs span many **scales**



~10 kW

100s of mW

10s of W

100s of W

How do you **program** non-NVIDIA GPUs?

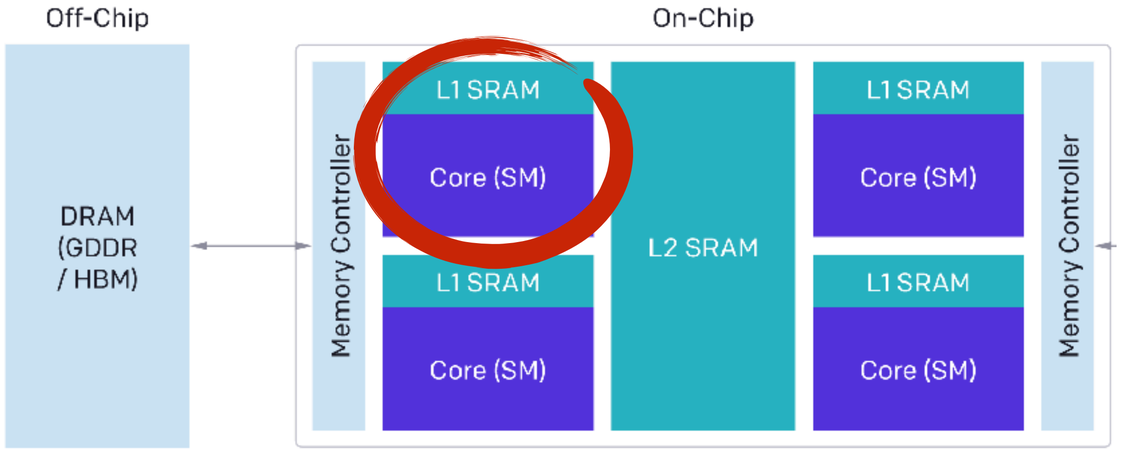
Compute APIs

Graphics APIs

↳ include **“compute shaders”**

Higher-level kernel languages

Triton



Explicit decomposition into grids & blocks

Implicit, automatic mapping & scheduling within blocks

Higher-level kernel languages

Triton

Embedded in Python

NumPy/PyTorch-like *within* blocks

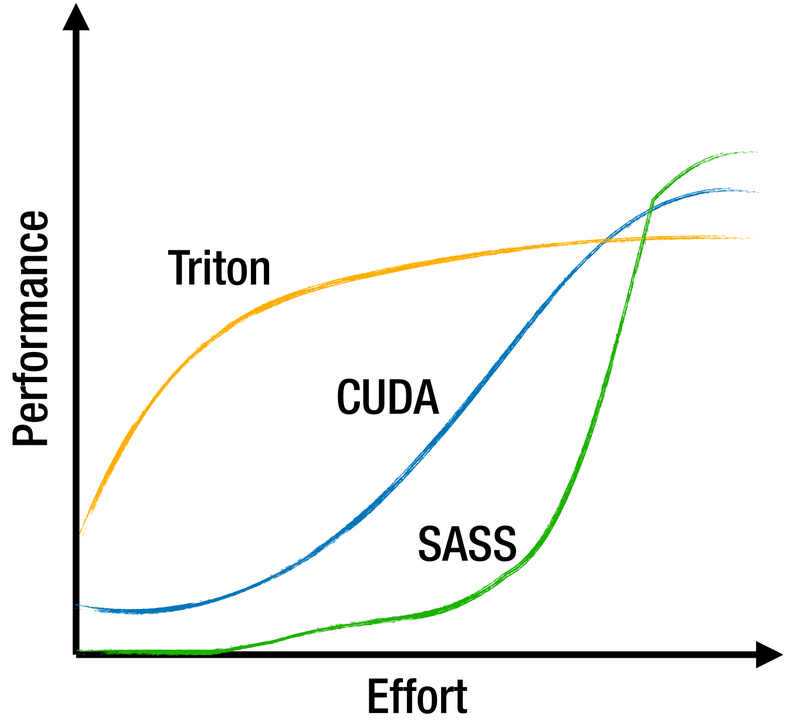
SPMD (CUDA-like) *across* blocks

```
@triton.jit
def matmul(A, B, C, M, N, K,
           stride_am, stride_ak,
           stride_bk, stride_bn,
           stride_cm, stride_cn,
           BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr,
           BLOCK_K: tl.constexpr):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)
    m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    k = tl.arange(0, BLOCK_K)
    # TILE_M x TILE_K pointers to A
    As = A + m[:,None]*stride_am + k[None,:]*stride_ak
    # TILE_K x TILE_N pointers to B
    Bs = B + k[:,None]*stride_bk + n[None,:]*stride_bn
    # Compute output tile
    c = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
    for k in range(0, K, BLOCK_K):
        a = tl.load(As)
        b = tl.load(Bs)
        C += tl.dot(a, b)
        As += BLOCK_K*stride_ak
        Bs += BLOCK_K*stride_bk
    # Write-back output tile
    Cs = C + m[:,None]*stride_cm + n[None,:]*stride_cn
    tl.store(Cs, c)
```

Higher-level kernel languages

Triton

cuTile,
CuTe DSL

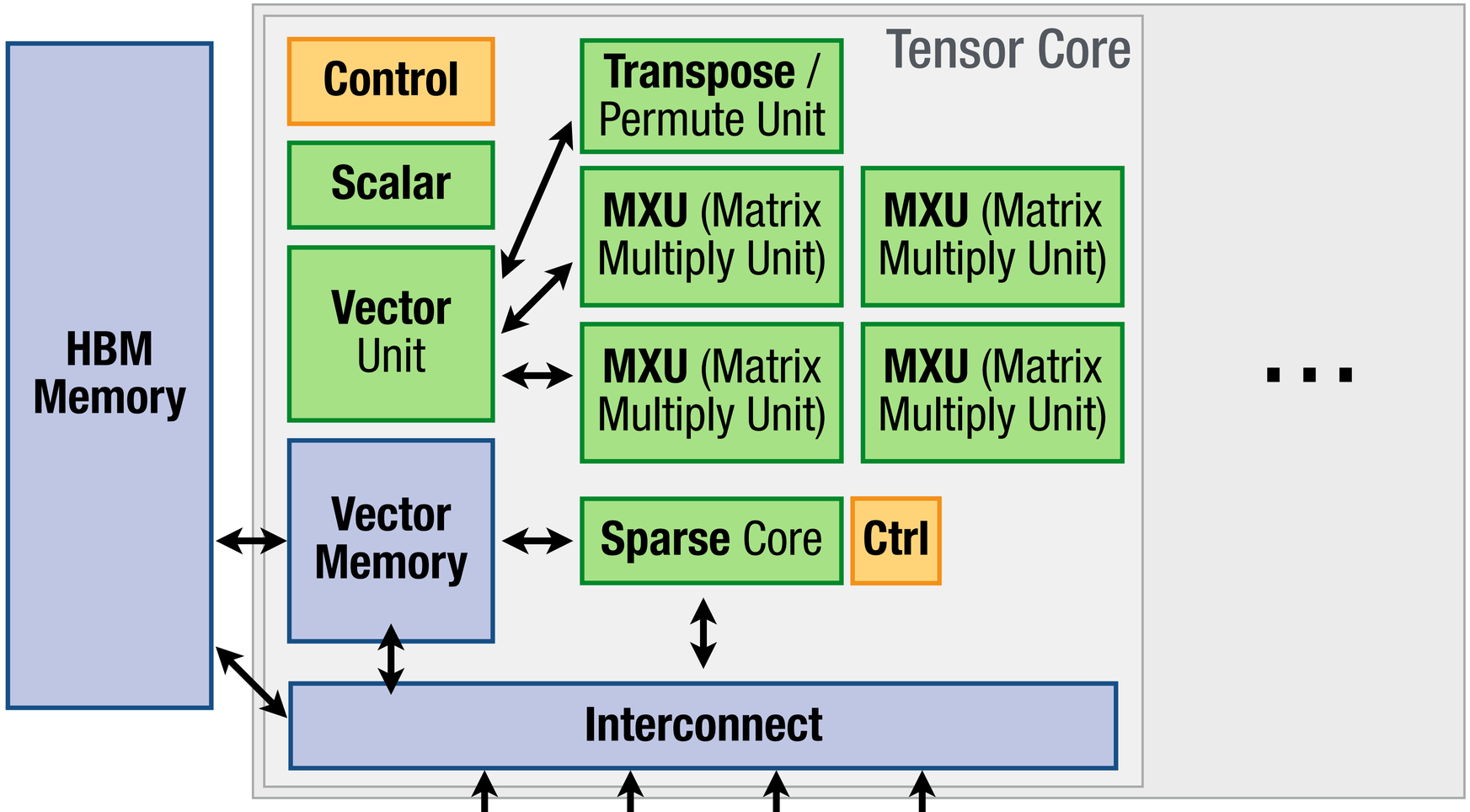


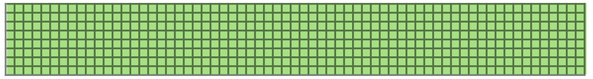
Google TPU

"Tensor Processing Unit"

Google TPU

“Tensor Processing Unit”





Vector Unit: 128x8

Memory: DMA only

MXU: 128x128 (systolic array)

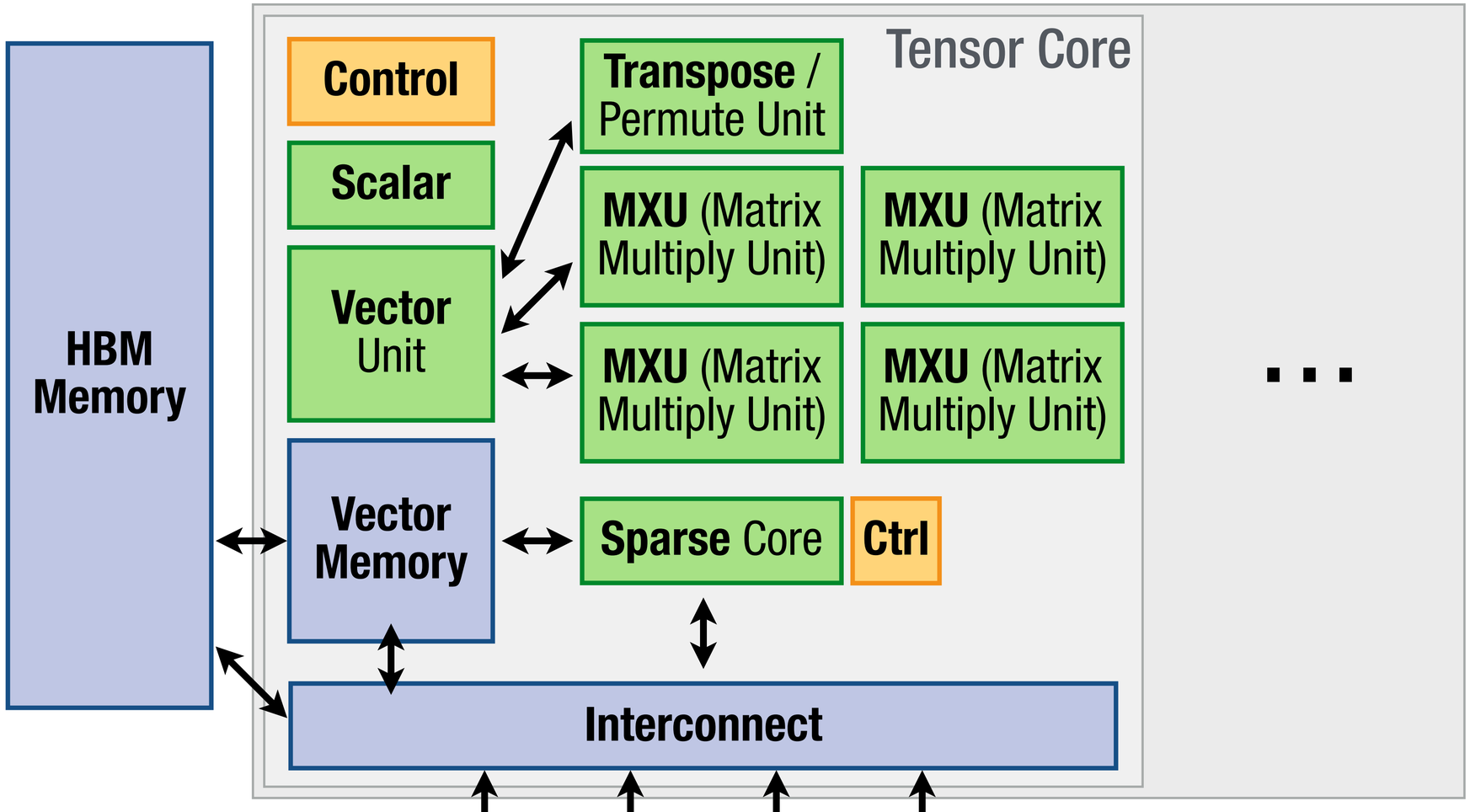
VLIW

Offload engines

High-BW Inter-chip interconnect

Google TPU

“Tensor Processing Unit”



vs. GPU:

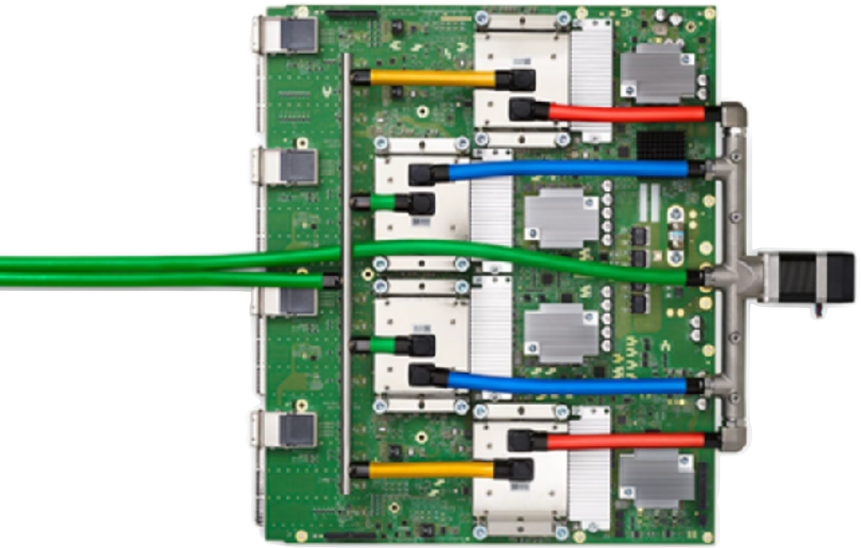
Few big cores

vs. many small cores

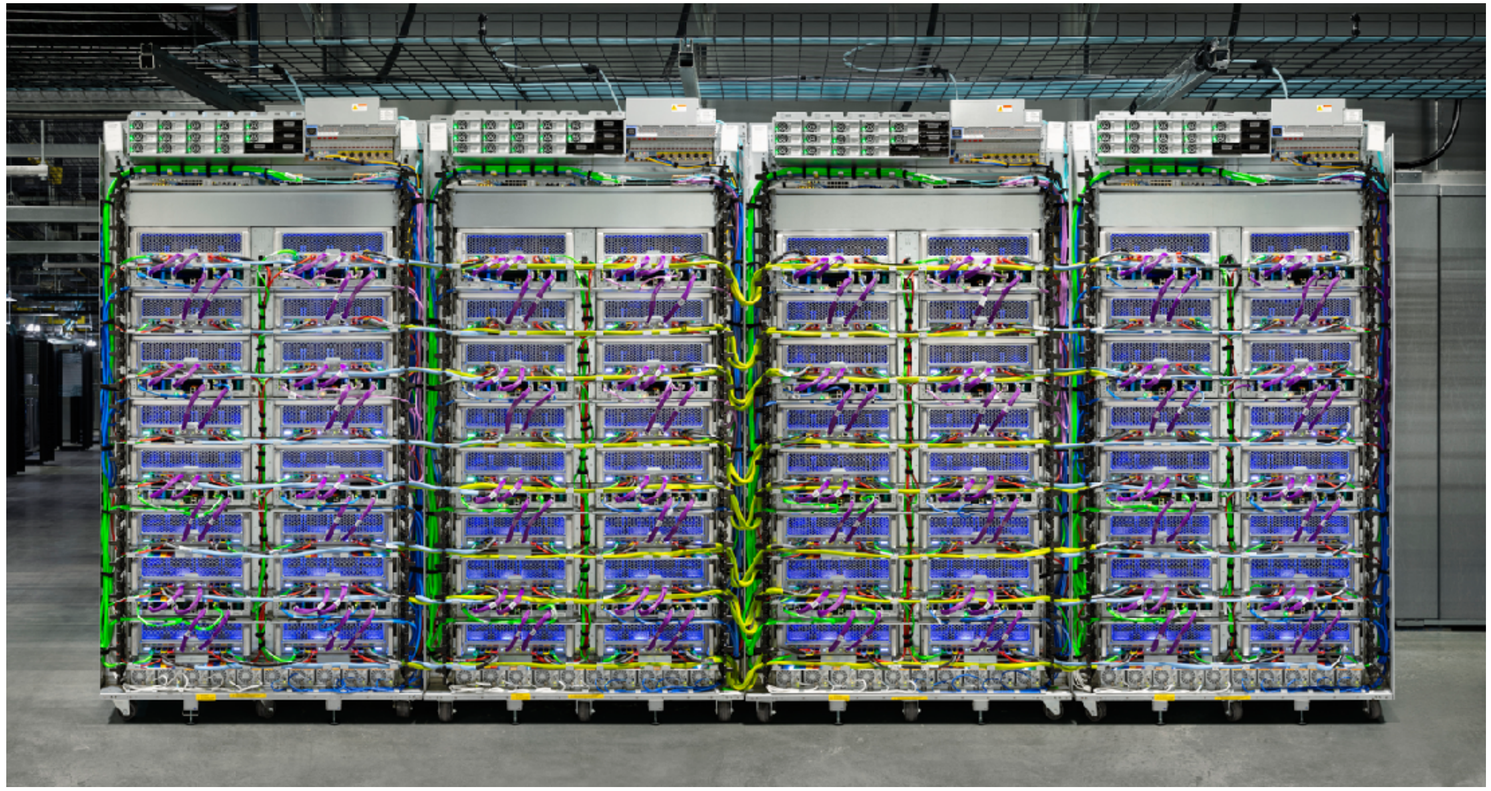
Static scheduling, explicit data movement

vs. dynamic scheduling & latency hiding

Designed to scale



4 chips / board



9,216 chips / “pod”

How do you program TPUs?

Built to run **ML models**, not kernels

How do you program TPUs?

Recently, also:
Pallas,
a Triton-like DSL

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

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

def matmul(
    x: jax.Array,
    y: jax.Array,
    *,
    bm: int = 128,
    bk: int = 128,
    bn: int = 128,
):
    m, k = x.shape
    _, n = y.shape
    return pl.pallas_call(
            matmul_kernel,
            out_shape=jax.ShapeDtypeStruct((m, n), x.dtype),
            in_specs=[pl.BlockSpec((bm, bk), lambda i, j, k: (i, k)),
                                pl.BlockSpec((bk, bn), lambda i, j, k: (k, j))],
            out_specs=pl.BlockSpec((bm, bn), lambda i, j, k: (i, j)),
            grid=(m // bm, n // bn, k // bk),
            compiler_params=pltpu.CompilerParams(
                    dimension_semantics=("parallel", "parallel", "arbitrary")),
    )(x, y)
```

TPU-like architectures are proliferating

Amazon
Trainium

Intel
Gaudi

NPU: Small, embedded “TPU”

- **Neural Engine**
  - 38 TOPS / sec
    16 cores
  - 32x32 = 1k MACs / core?
  - Fixed-point only (no training)
  - **Only programmable via high-level API** (CoreML)

Even more where that came from…

DSPs

IPUs

FPGAs

CGRAs / “RDAs”

SmartNICs / “DPUs”

…

Common theme:
change programming model & specialize
to unlock architectural efficiency

Questions?