Lecture 11:

Programming Specialized Hardware for AI

Parallel Computing
Stanford CS149, Fall 2025

Today's Theme

Specialized HW for AI?

How do you program specialized hardware?

Google TPU

- Efficient dense matrix multiply ⇒systolic array

Nvidia H100 and B100

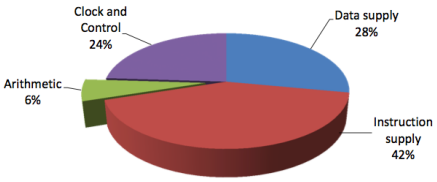
- Asynchronous compute and memory mechanisms ⇒ complex programing
- Simplify with Thunderkittens DSL

SambaNova SN40L

- Dataflow architecture
- Programing model: tiling and streaming with metapipelining

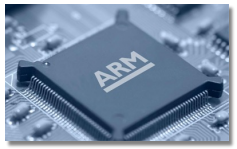
Recall: Energy Efficiency vs. Programmability

Programmability adds overhead ⇒ reduces efficiency

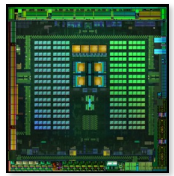


Efficient Embedded Computing [Dally et al. 08]

Energy-optimized CPU



Throughput-oriented processor (GPU)



Programmable DSP

HEXAGON™

Domain Specific Accelerator

Google TPU

FPGA/ reconfigurable logic

ASIC

Video encode/decode, Audio playback, Camera RAW processing, neural nets (future?)

~10X more efficient

~20X

~50X??? (jury still out)

~100-1000X more efficient

Easiest to program

Limited domain of programmability with DSLs (e.g. DNN)

Difficult to program (making it easier is active area of research)

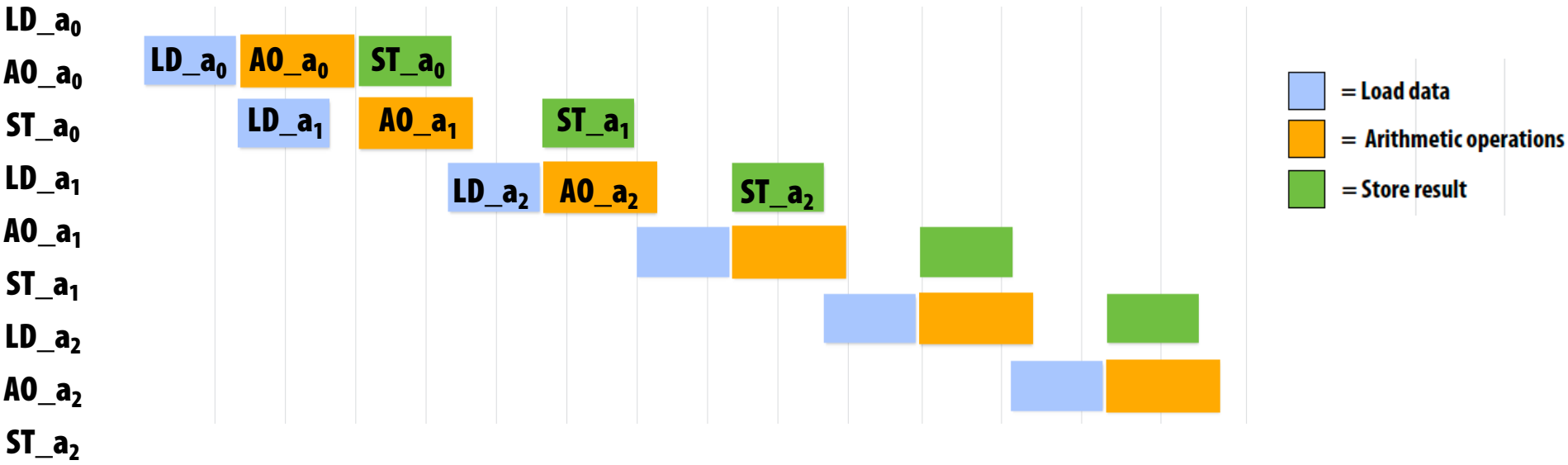
Not programmable + costs 10-100's millions of dollars to design / verify / create

Credit: Pat Hanrahan for this slide design

Synchronous (blocking) Execution

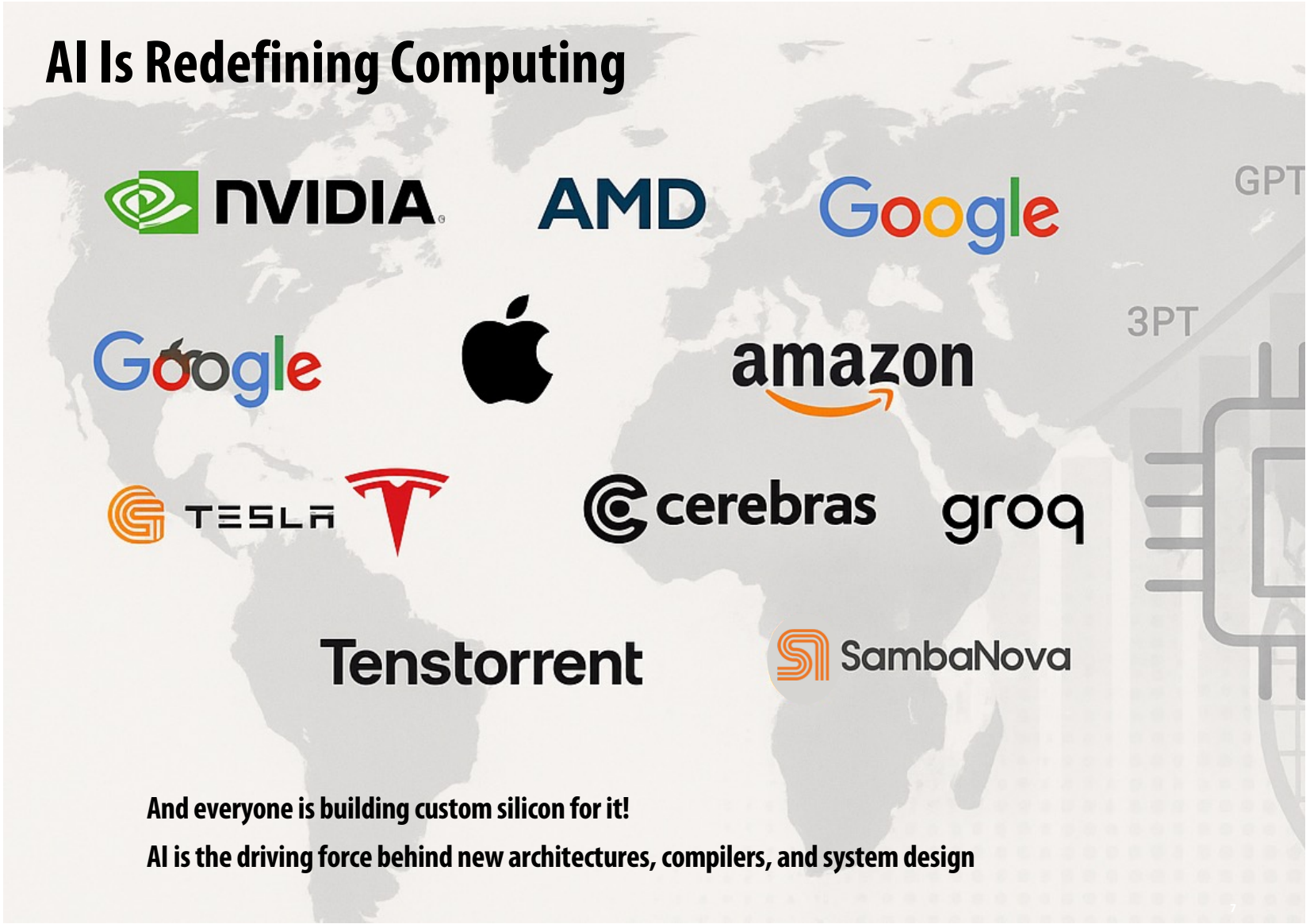
Start later operations after earlier operations are complete

Asynchronous (Nonblocking) Execution



Start later operations before earlier operations are complete
Software + Hardware: asynchronous instructions, synchronization
Hardware: out-of-order execution

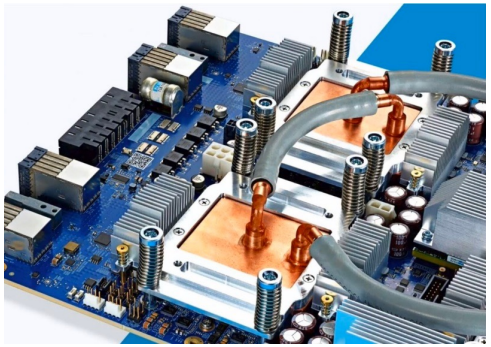
AI Is Redefining Computing



And everyone is building custom silicon for it!

AI is the driving force behind new architectures, compilers, and system design

Hardware acceleration of AI inference/training



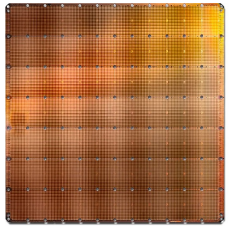
Google TPU3

AWS Trainium 2

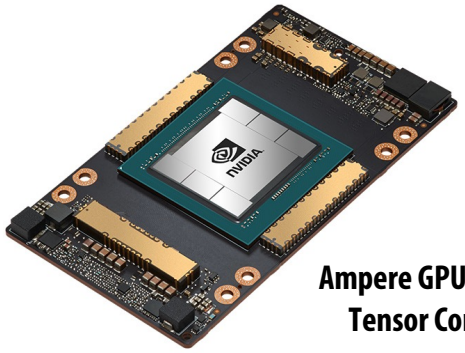
Apple Neural Engine

Intel Deep Learning
Inference Accelerator

SambaNova
Cardinal SN10



Cerebras Wafer Scale Engine



Ampere GPU with
Tensor Cores

Google's TPU (v1)

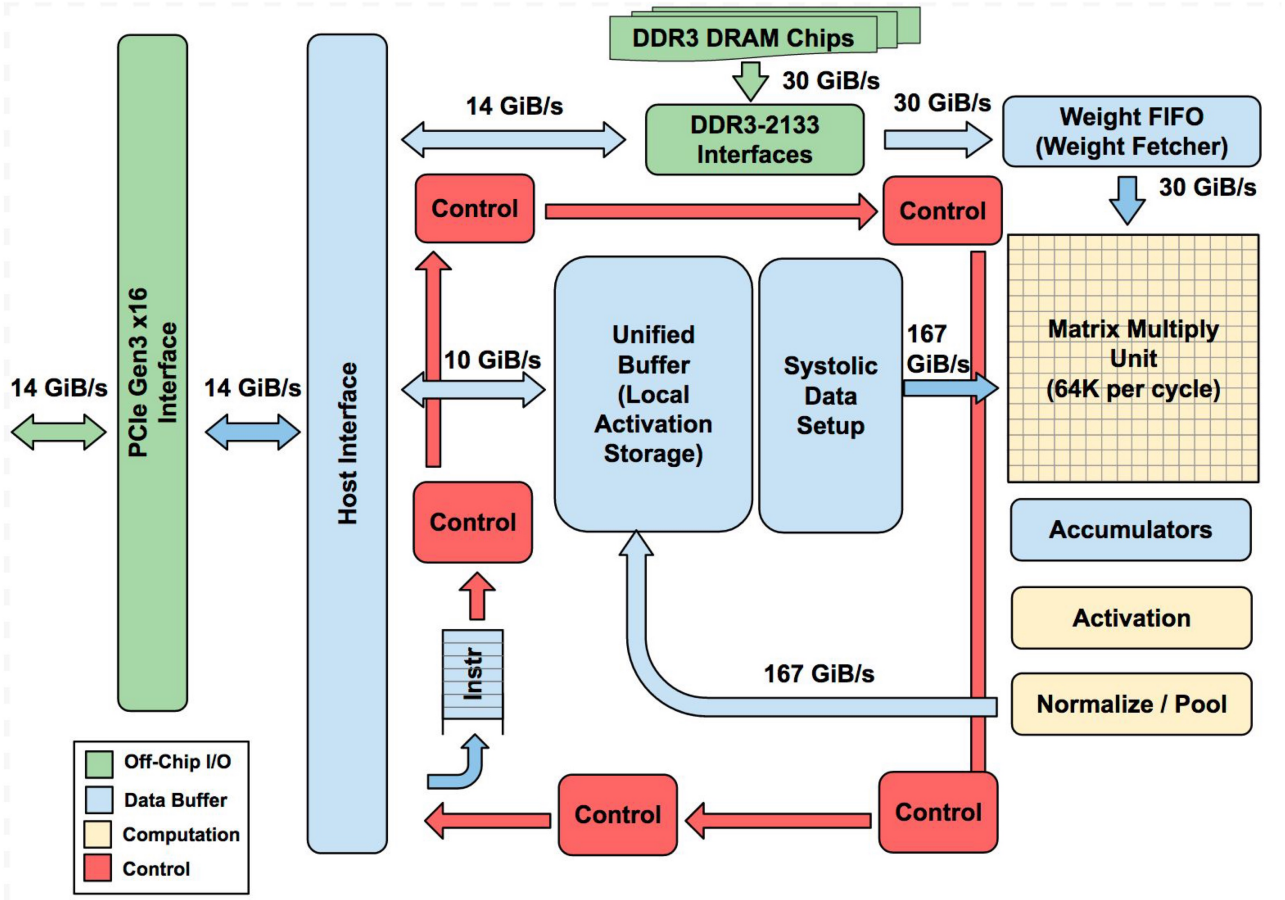
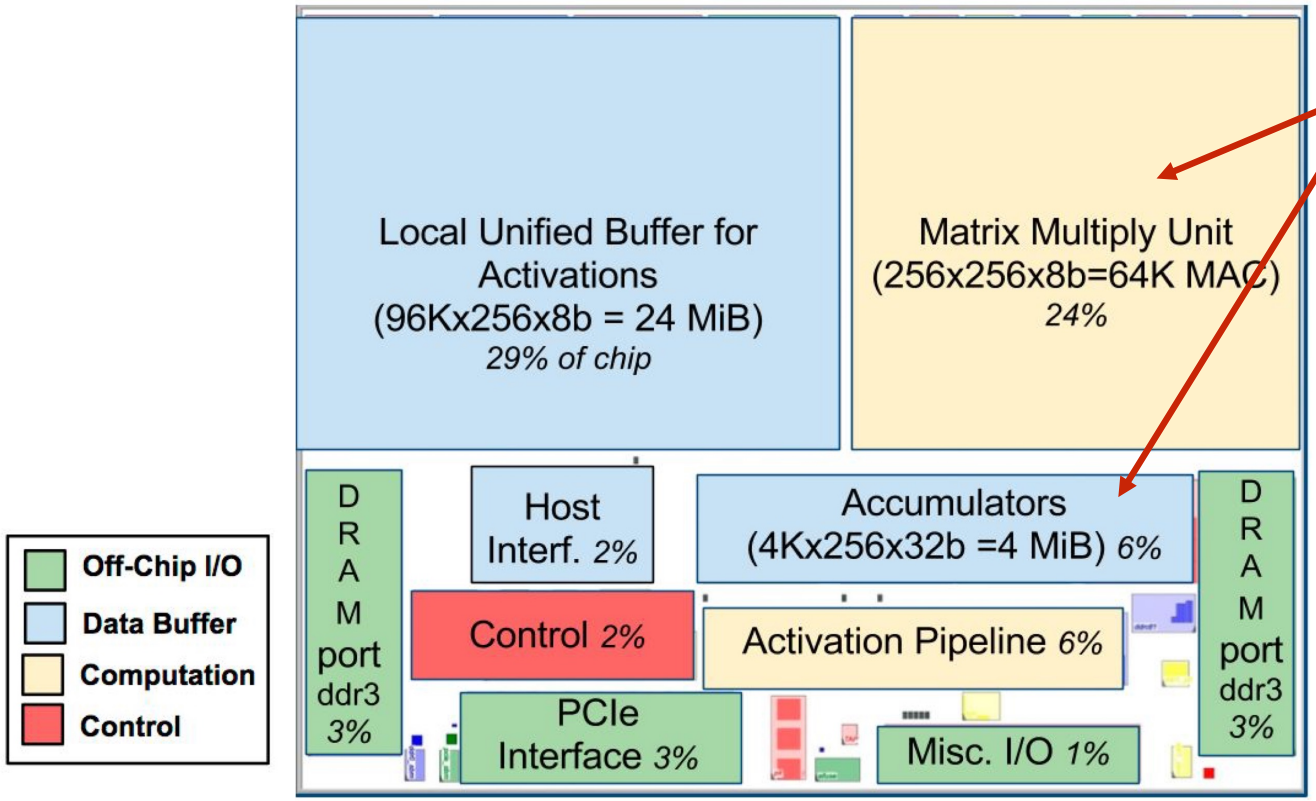


Figure credit: Jouppi et al. 2017

TPU area proportionality



Arithmetic units ~ 30% of chip

Note low area footprint of control

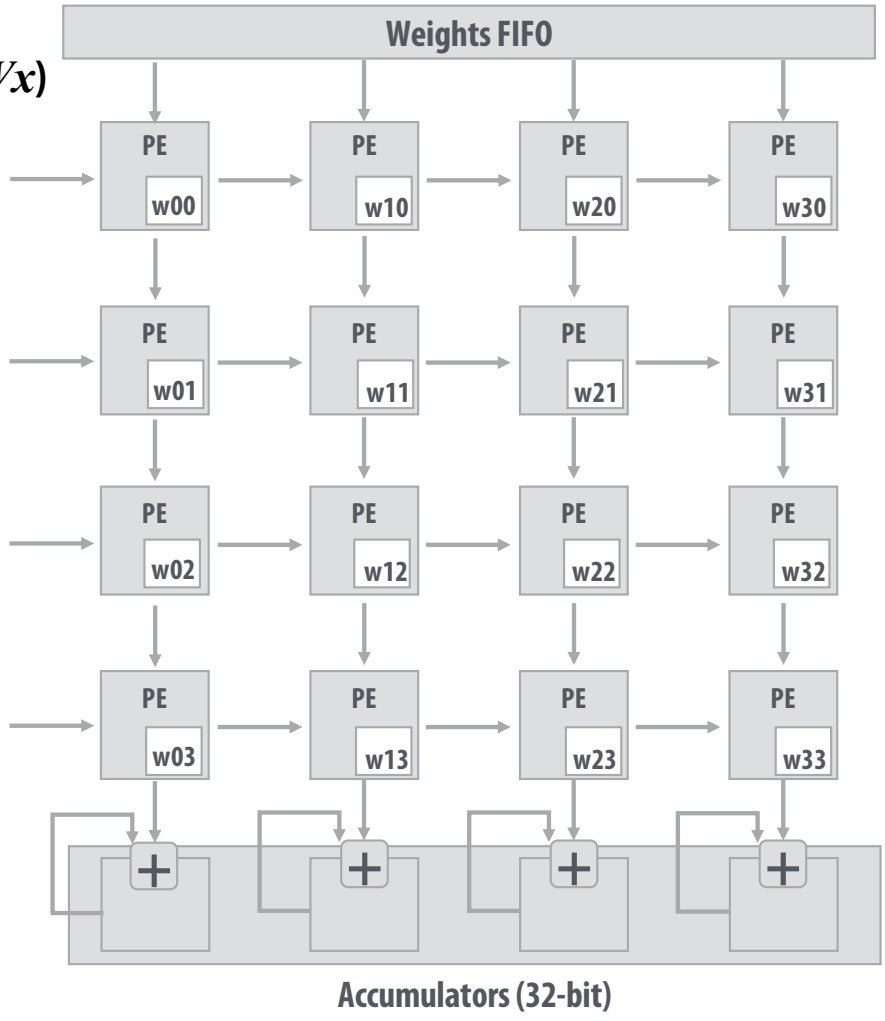
Key instructions:

- **read host memory**
- **write host memory**
- **read weights**
- **matrix_multiply / convolve**
- **activate**

Figure credit: Jouppi et al. 2017

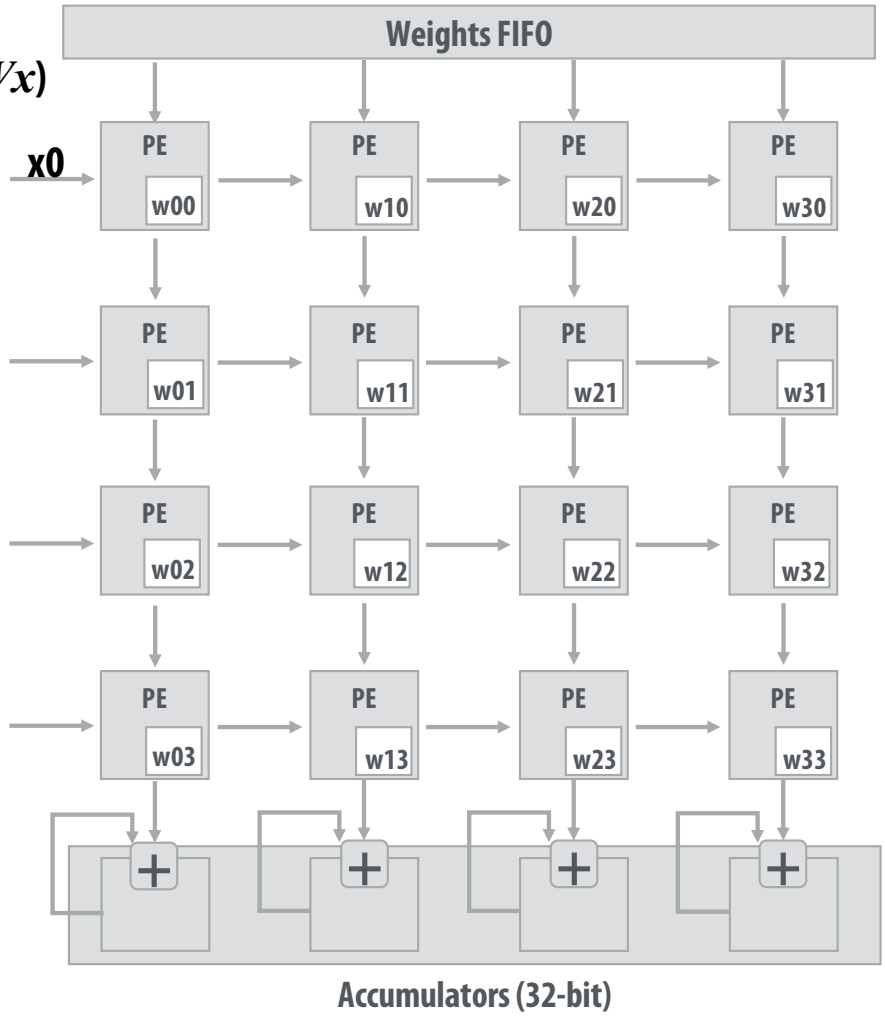
Systolic array

(matrix vector multiplication example: $y=Wx$)



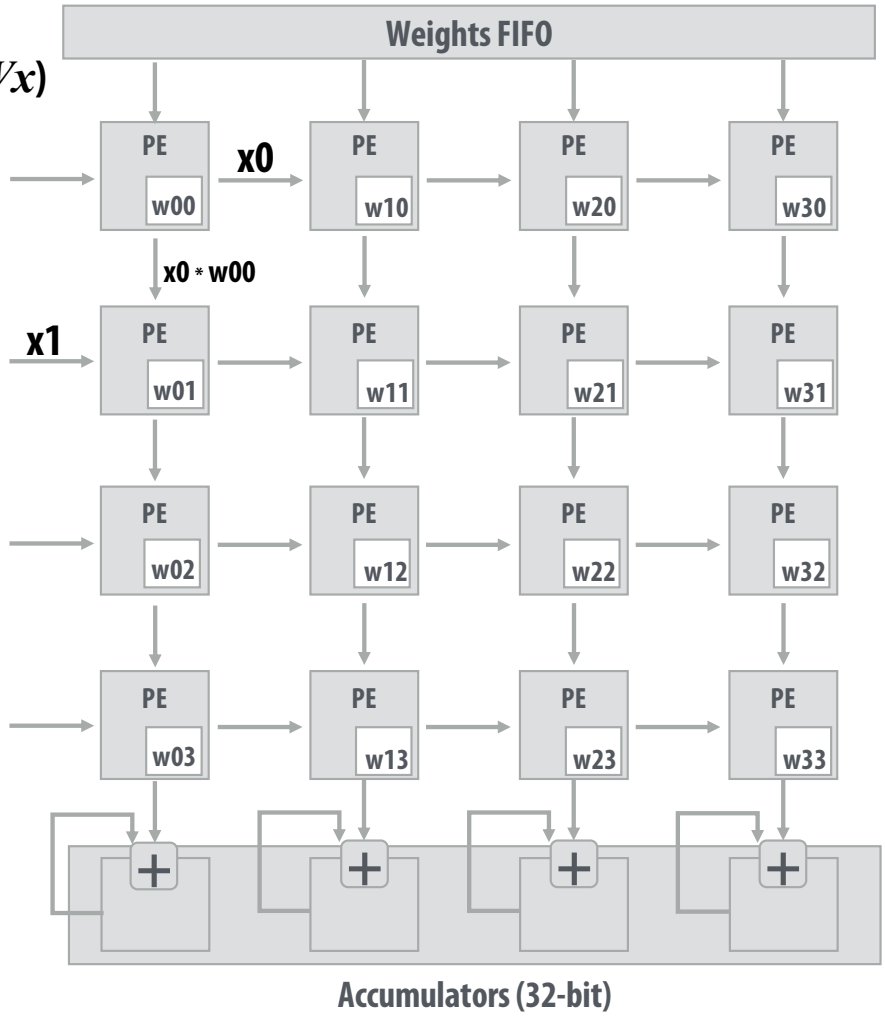
Systolic array

(matrix vector multiplication example: $y=Wx$)



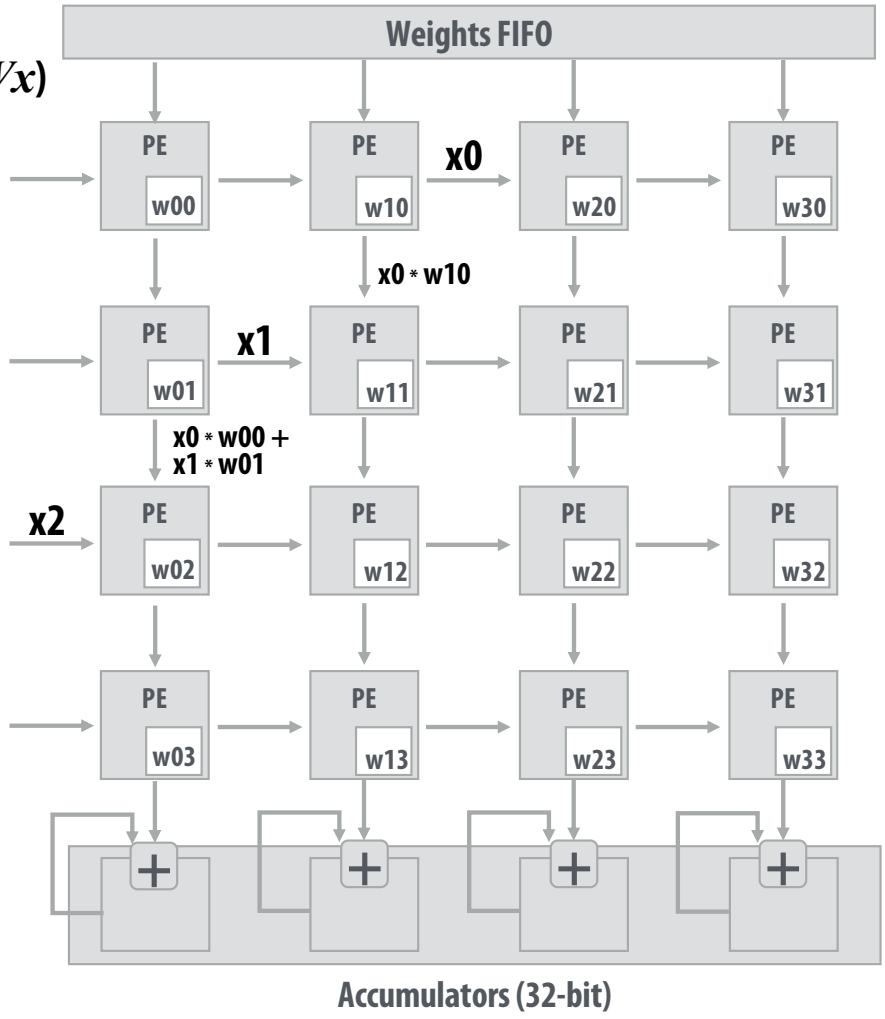
Systolic array

(matrix vector multiplication example: $y=Wx$)



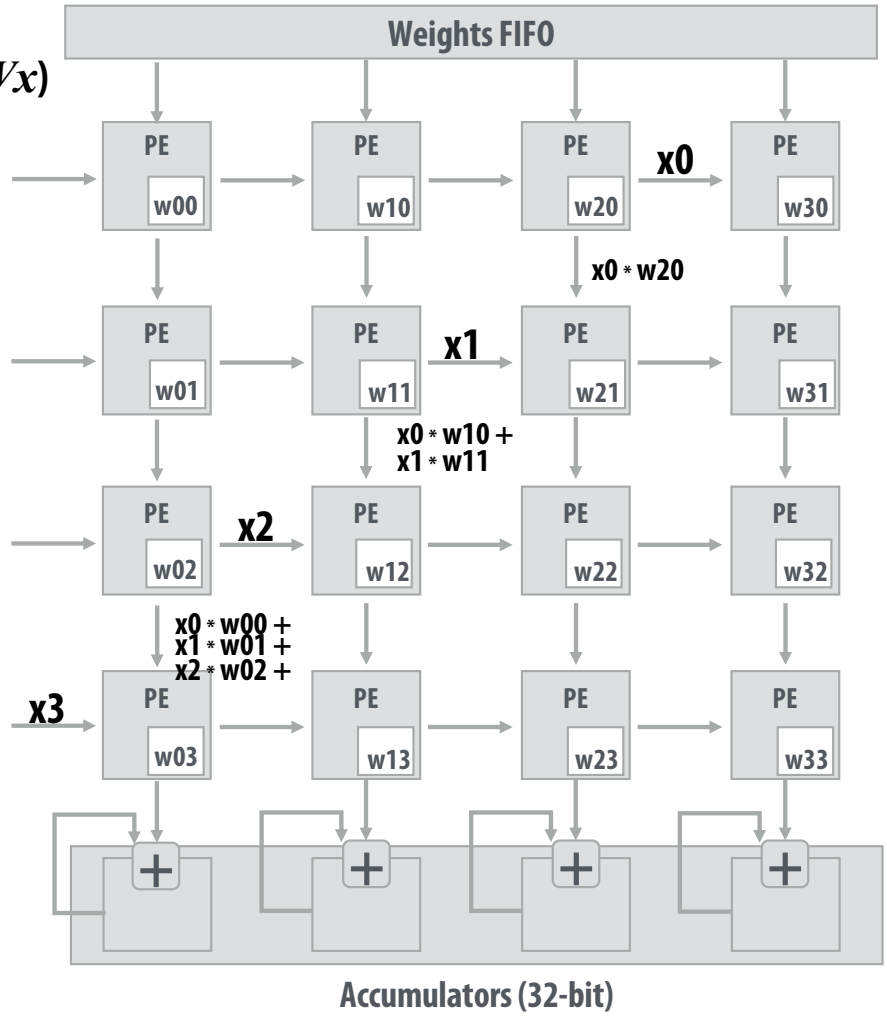
Systolic array

(matrix vector multiplication example: $y=Wx$)



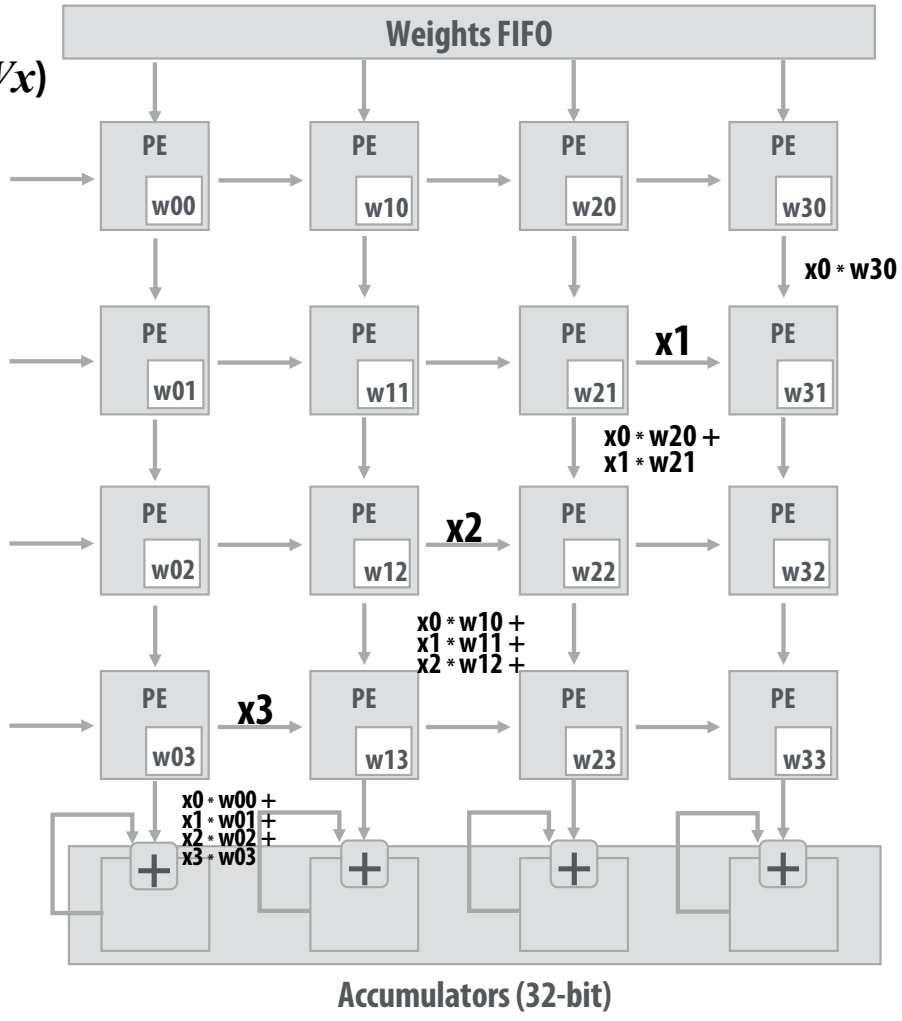
Systolic array

(matrix vector multiplication example: $y=Wx$)



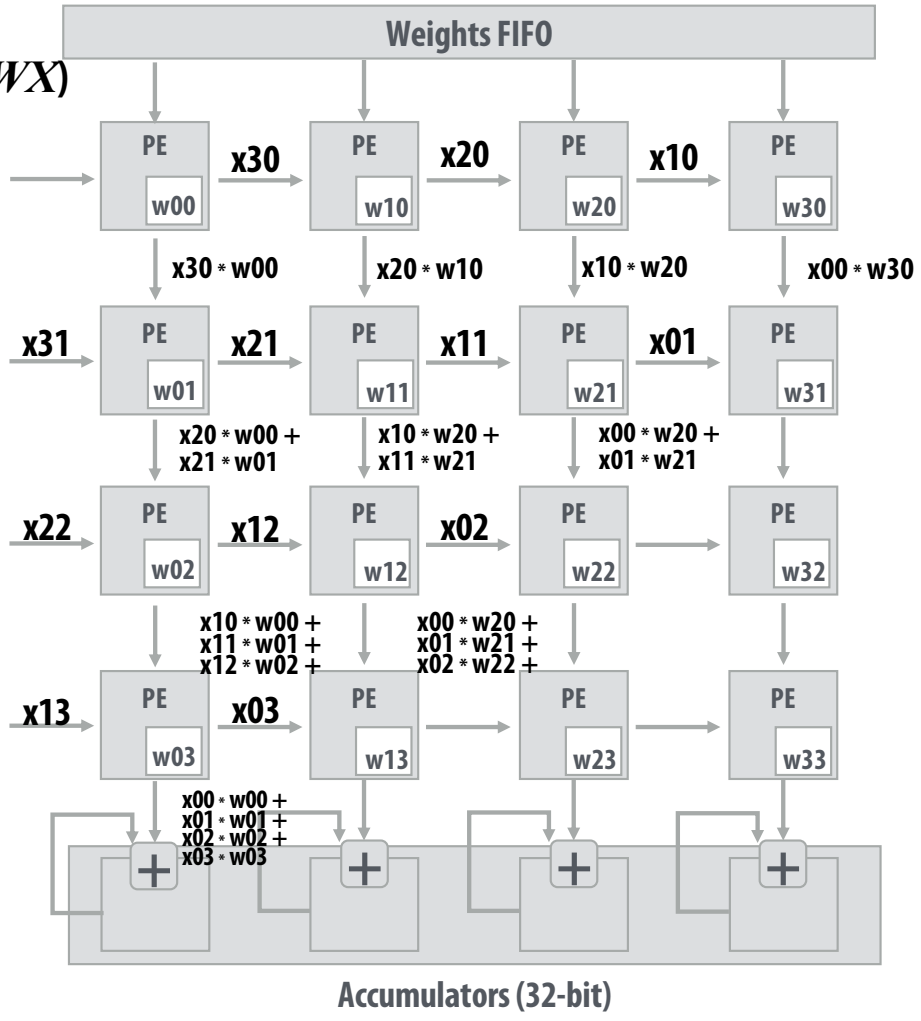
Systolic array

(matrix vector multiplication example: $y=Wx$)



Systolic array

(matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

Systolic Array Dataflow

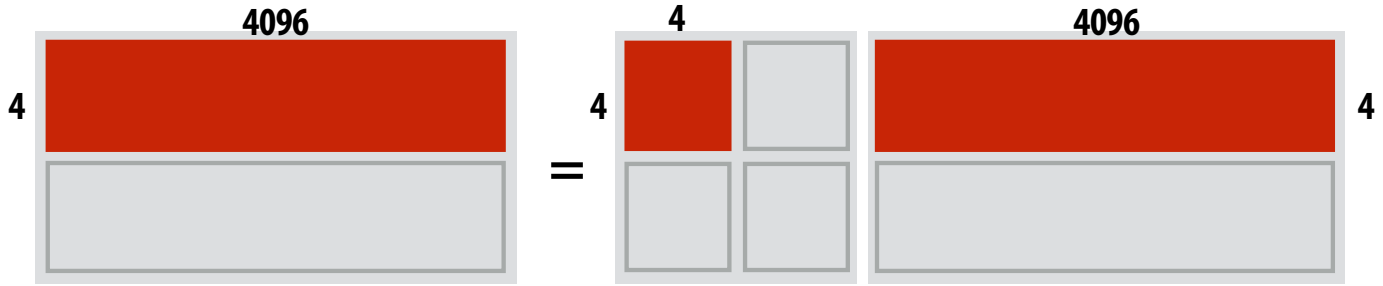
Dataflow Type	What stays in each PE	What streams through	Main goal
Weight-Stationary (WS)	Weight values	Inputs (activations) and partial sums	Minimize reloading of weights
Output-Stationary (OS)	Partial sums (outputs)	Inputs and weights	Minimize movement of accumulated results
Input-Stationary (IS)	Input activations	Weights and partial sums	Minimize reloading of inputs

SIMD vs Systolic Array

Feature	SIMD	Systolic Array
Dataflow	Control-driven (instructions)	Data-driven (wavefront)
Locality (data reuse)	Limited	Temporal and spatial
Communication	Global (register/memory)	Local (neighbor PEs)
Control	Centralized	Distributed
Efficiency (perf/mm^2, perf/Watt)	low	high

Building larger matrix-matrix multiplies

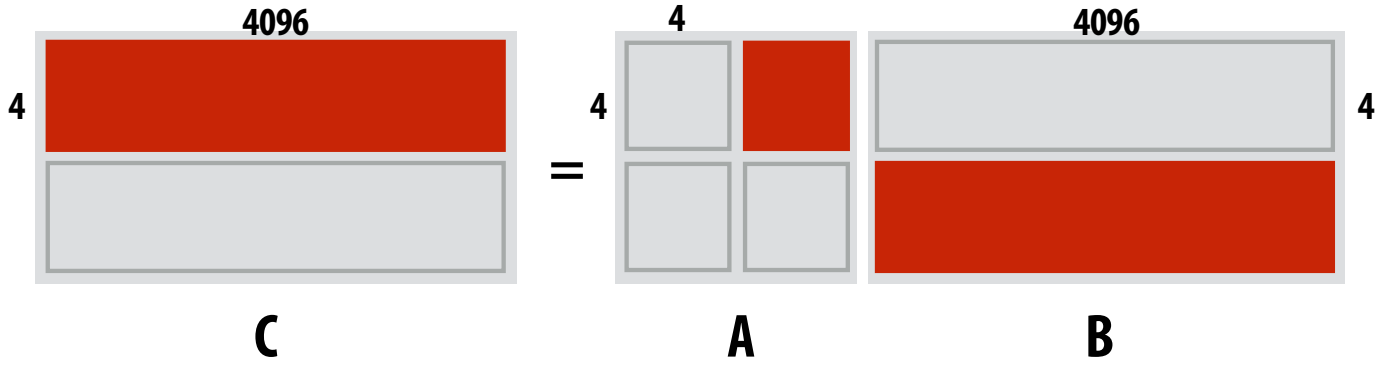
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

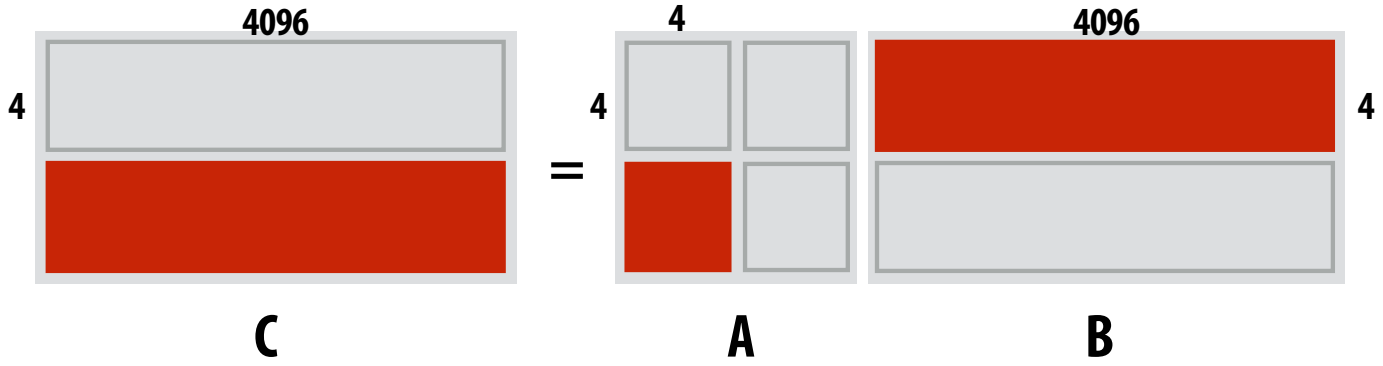
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

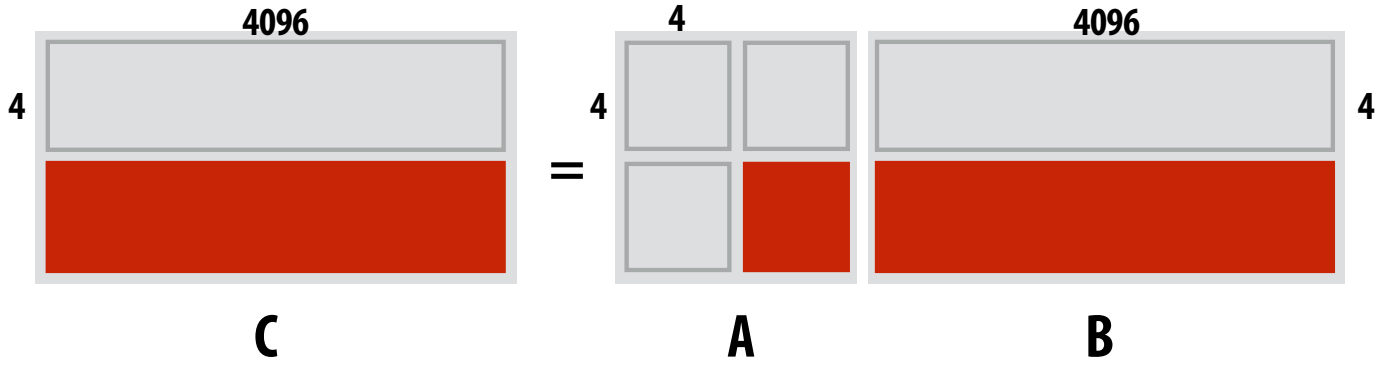
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

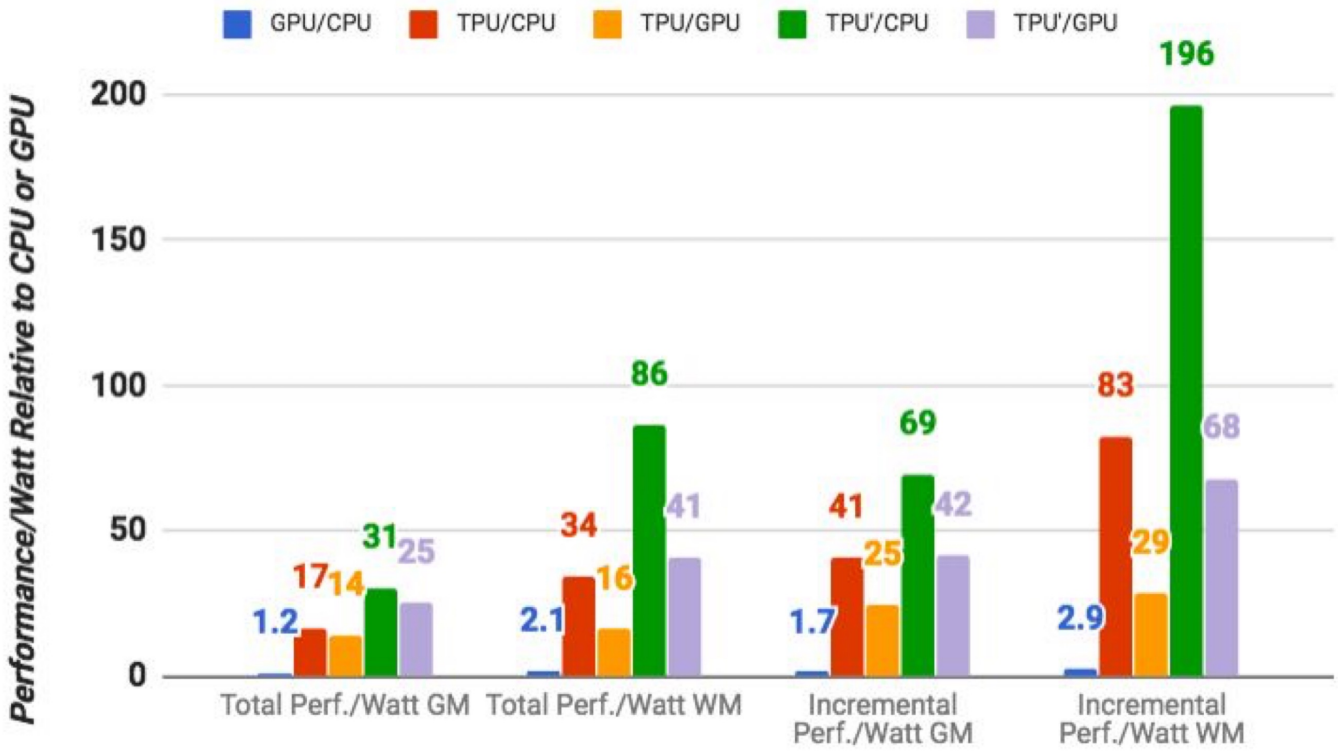
Building larger matrix-matrix multiplies

Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

TPU Performance/Watt



GM = geometric mean over all apps
WM = weighted mean over all apps

total = cost of host machine + CPU
incremental = only cost of TPU

Figure credit: Jouppi et al. 2017

Evolution of Google TPUs

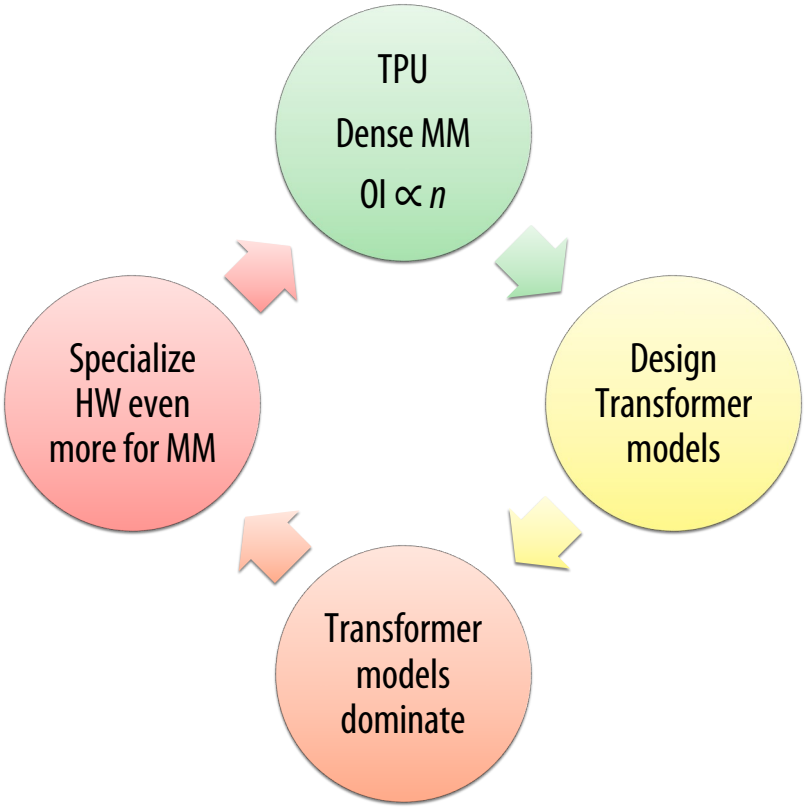
Google TPU Compute Engines								"Trillium"	"Ironwood"
	TPU v1	**TPU v2**	**TPU v3**	**TPU v4i**	**TPU v4**	**TPU v5p**	**TPU v5e**	**TPU v6e**	**TPU v7p**
First Deployed	Q2 2015	Q3 2017	Q4 2018	Q1 2020	Q4 2021	Q4 2023	Q3 2023	Q4 2024	*Q4 2025*
ML Inference	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ML Training	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Chip Process	28 nm	16 nm	16 nm	7 nm	7 nm	*5 nm*	*5 nm*	*4 nm*	*3 nm*
Transistors	3.0 B	9.0 B	10.0 B	16.0 B	*31.2 B*	*54.9 B*	*27.4 B*	*86.7 B*	*274.4 B*
Die Size	330 mm^2	625 mm^2	700 mm^2	400 mm^2	*780 mm^2*	*700 mm^2*	*350 mm^2*	*790 mm^2*	*2 * 445 mm^2*
Clock Speed	700 MHz	700 MHz	940 MHz	1,050 MHz	*1,050 MHz*	*2,040 MHz*	1,750 MHz	*2,060 MHz*	*1,633 MHz*
TensorCores Per Chip	1	2	2	1	2	2	1	1	*2*
SparseCores Per Chip	-	-	-	-	-	4	-	2	*4*
MXU Matrix Size/Core	1 * 256x256	1 * 128x128	2 * 128x128	4 * 128x128	4 * 128x128	4 * 128x128	4 * 128x128	4 * 256x256	*4 * 256x256*
Dataflow SparseCores	-	-	-	-	4	4	*2*	*4*	*4*
On Chip Cache Memory	28 MB	32 MB	32 MB	144 MB	32 MB	48 MB	112 MB	*???*	*???*
Off Chip HBM Memory	8 GB	16 GB	32 GB	8 GB	32 GB	95 GB	16 GB	32 GB	192 GB
HBM Memory Bandwidth	*300 Gb/sec*	700 GB/sec	900 GB/sec	*300 GB/sec*	1,228 GB/sec	2,765 GB/sec	819 GB/sec	1,640 GB/sec	7,372 GB/sec
Precision	INT8	BF16	BF16	BF16 INT8	BF16 INT8	BF16 INT8	BF16 INT8	BF16 INT8	BF16 **INT8** FP8
INT8 Peak Teraops	92	-	-	138	275	918	393	1,836	4,614
BF16 Peak Teraflops	-	46	123	69	137.5	459	196.5	918	2,307
FP8 Peak Teraflops	-	-	-	-	-	-	-	-	4,614
ICI Links * Speed Gb/sec	-	4 * 496	4 * 656	2 * 400	6 * 448	6 * 800	4 * 400	4 * 896	4 * 1,344
ICI Bandwidth	-	1,984 Gb/sec	2,624 Gb/sec	800 Gb/sec	2,668 Gb/sec	4,800 Gb/sec	1,600 Gb/sec	3,584 Gb/sec	5,378 Gb/sec
Interconnect Topology	-	2D Torus	2D Torus	-	3D Torus	3D Torus	2D Torus	2D Torus	3D Torus
Chip Idle Watts	28	53	84	55	170	*???*	*???*	*???*	*???*
Max Measured Watts	*???*	*???*	262	*???*	192	*???*	*???*	*???*	*???*
Chip TDP Watts	75	280	450	175	*300*	*537*	*225*	*383*	*959*
Chips Per CPU Host	4	4	4	8	4	*8*	*8*	*8*	*8*
Max Chips Per Pod	-	**256**	**1,024**	-	**4,096**	**8,960**	**256**	**256**	**9,216**
Peak Petaops/Petaflops Per Pod (INT8 OR FP8 ELSE BF16)	-	*12*	*126*	-	*1.126*	*8.225*	*101*	*470*	*42.523*
All-Reduce Bandwidth Per Pod	-	*120 TB/sec*	340 TB/sec	-	1,100 TB/sec	*4,325 TB/sec*	51.2 TB/sec	102.4 TB/sec	*4,981 TB/sec*
Bisection Bandwidth Per Pod	-	2 TB/sec	6.4 TB/sec	-	24 TB/sec	*94.5 TB/sec*	1.6 TB/sec	3.2 TB/sec	*108.9 TB/sec*

Source: The Next Platform

Hardware Lottery

When a research idea wins because it is suited to the available software and hardware and not because the idea is universally superior to alternative research directions.

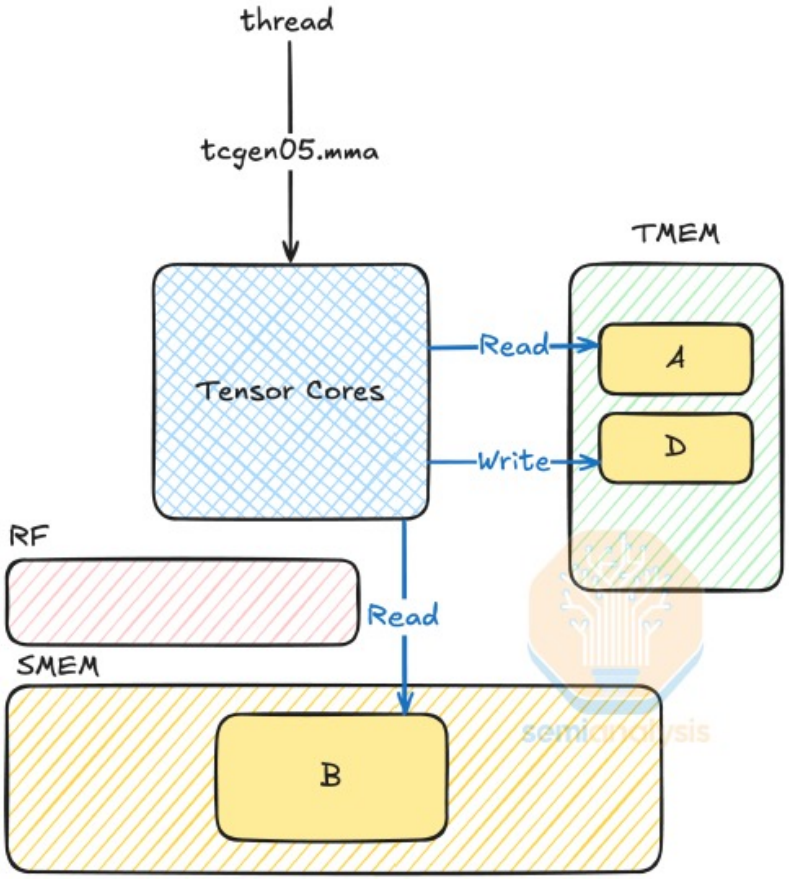
Sara Hooker



Summary: specialized hardware for AI model processing

- Specialized hardware for executing key DNN computations efficiently
- Feature many arithmetic units
- Customized/configurable datapaths to directly move intermediate data values between processing units (schedule computation by laying it out spatially on the chip) at multiple granularities
- Large amounts of on-chip storage for fast access to intermediates

Tensor Cores in B100



Register bandwidth limits for tensor cores in B100

Tensor data in SMEM and TMEM

Single threads execute MMA ⇒ No more warps!

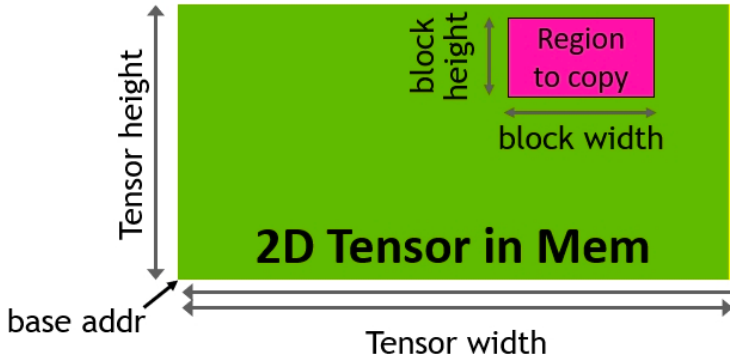
Programming Tensor Cores

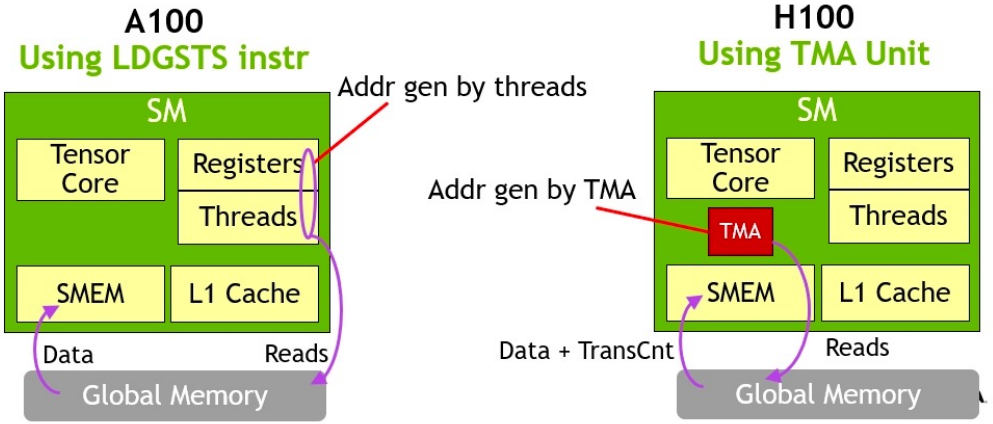
- **Allocate TMEM and descriptors**
  - tcgen05.alloc
- **Prefetch/stream tiles with TMA (async)**
  - cp.async.bulk.tensor, coordinate with mbarrier
- **Launch async MMAs**
  - tcgen05.mma batch with tcgen05.commit
- **Order & retire**
  - tcgen05.fence

Not your father's CUDA

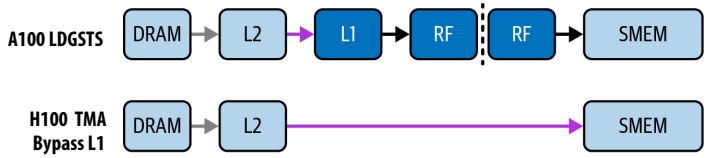
Tensor Memory Accelerator

Copy Descriptor





- Special purpose instructions for efficient data movement
- Asynchronously load/store a region of a tensor from global to shared memory
- Copy descriptor describes region
- Single thread issue TMA operation `cuda::memcpy_async`
- Signal barrier when copy is complete
- Hardware address generation and data movement



Specialization Improves Efficiency

Tensor Cores

- Specialized MMA compute

Tensor Core Size Increases			
Architecture	Tensor Core FLOP/Cycle/SM	Max MMA Shape	Max FLOPs per PTX Instruction (2 * m * n * k)
Volta	F16: 1024	m8n8k4	512
Ampere	F16: 2048	m16n8k16	4096
Hopper	F16: 4096 F8: 8192	Warpgroup Level F16: m64n256k16 F8: m64n256k64	Warpgroup Level F16: 524,288 F8: 2,097,152
Blackwell	F16: 8192 F8: 16384 F4: 32768	2 SM F16: m256n256k16 F8: m256n256k32 F4: m256n256k96	2 SM F16: 2,097,152 F8: 4,191,304 F4: 12,582,912

- Warpgroup: 128 consecutive threads
- PTX: Parallel Thread Execution
  NVIDIA's virtual instruction set architecture

TMA

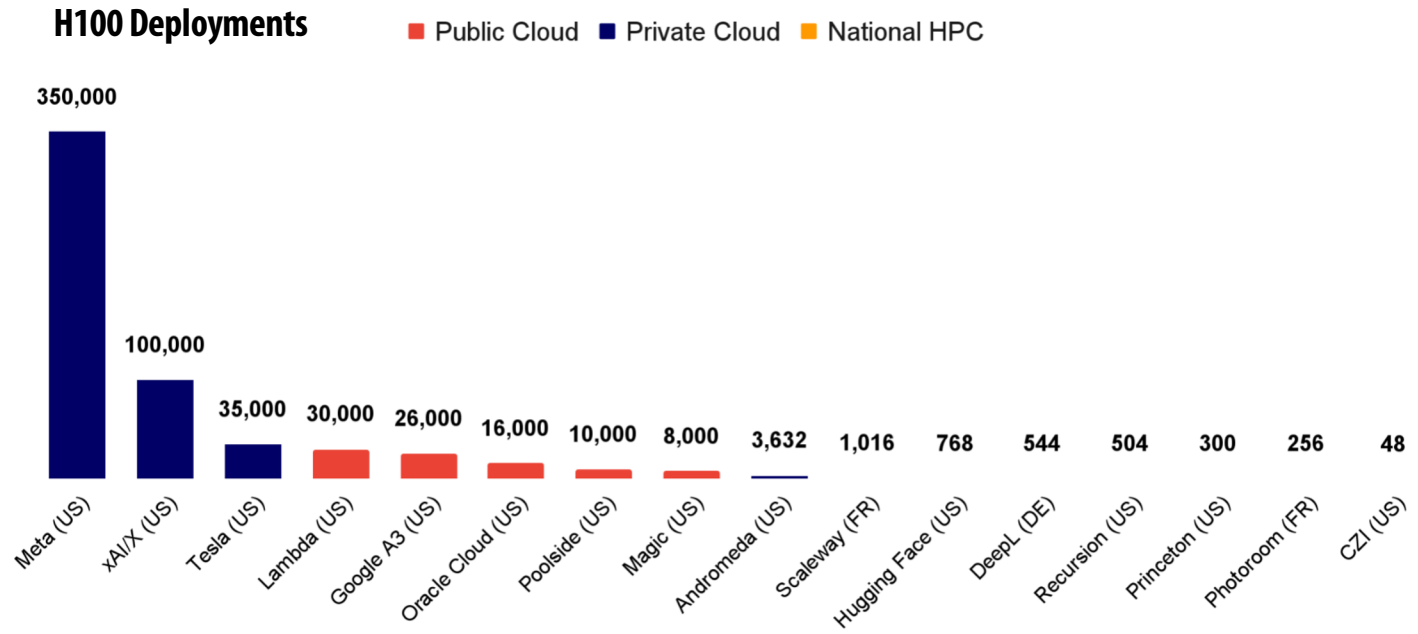
- Specialized block data movement unit
- Eliminates 1000's of instructions and memory addressing overhead
- Eliminates unnecessary data movement through L1 and registers

How Ideal are GPUs

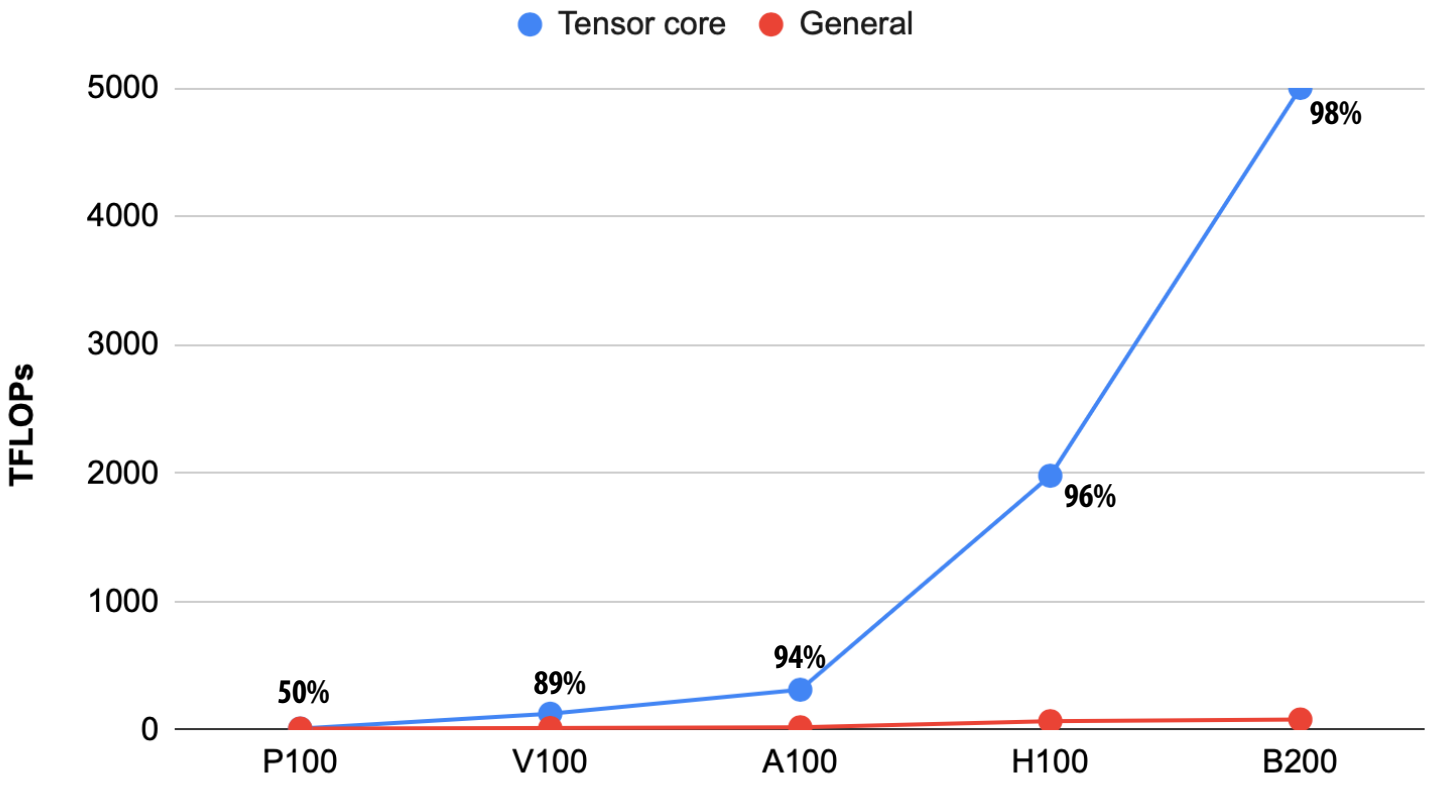
Feature	Why?	Nvidia GPU
Tiled tensors (e.g. 16 x 16, 32 x 32)	Max TFLOPS on GEMM Low instr. overhead	✅
Asynchronous compute	Overlap compute and memory access	✅ mma_async
Asynchronous memory access	Overlap compute and memory access	✅ TMA+TMEM
Asynchronous chip-to-chip communication	Overlap compute, memory and communication	
Compute unit to compute unit comm.	Fusion and pipelining Streaming Dataflow	? TB Cluster

GPU Kernels are Important

- 2025 GPU market is enormous ⇒ NVIDIA 2025 quarterly revenue of >$47B
- GPU AI kernels are often run on clusters of hundreds of millions of dollars of GPUs, for months on end. (e.g. large training runs, serving models at scale, etc.)
- FlashAttention-2 degraded from ~70% on A100s to ~35% on H100s. Took 2 years to come back up to ~65% with FlashAttention-3
- Poor kernels underutilize billions of dollars worth of compute

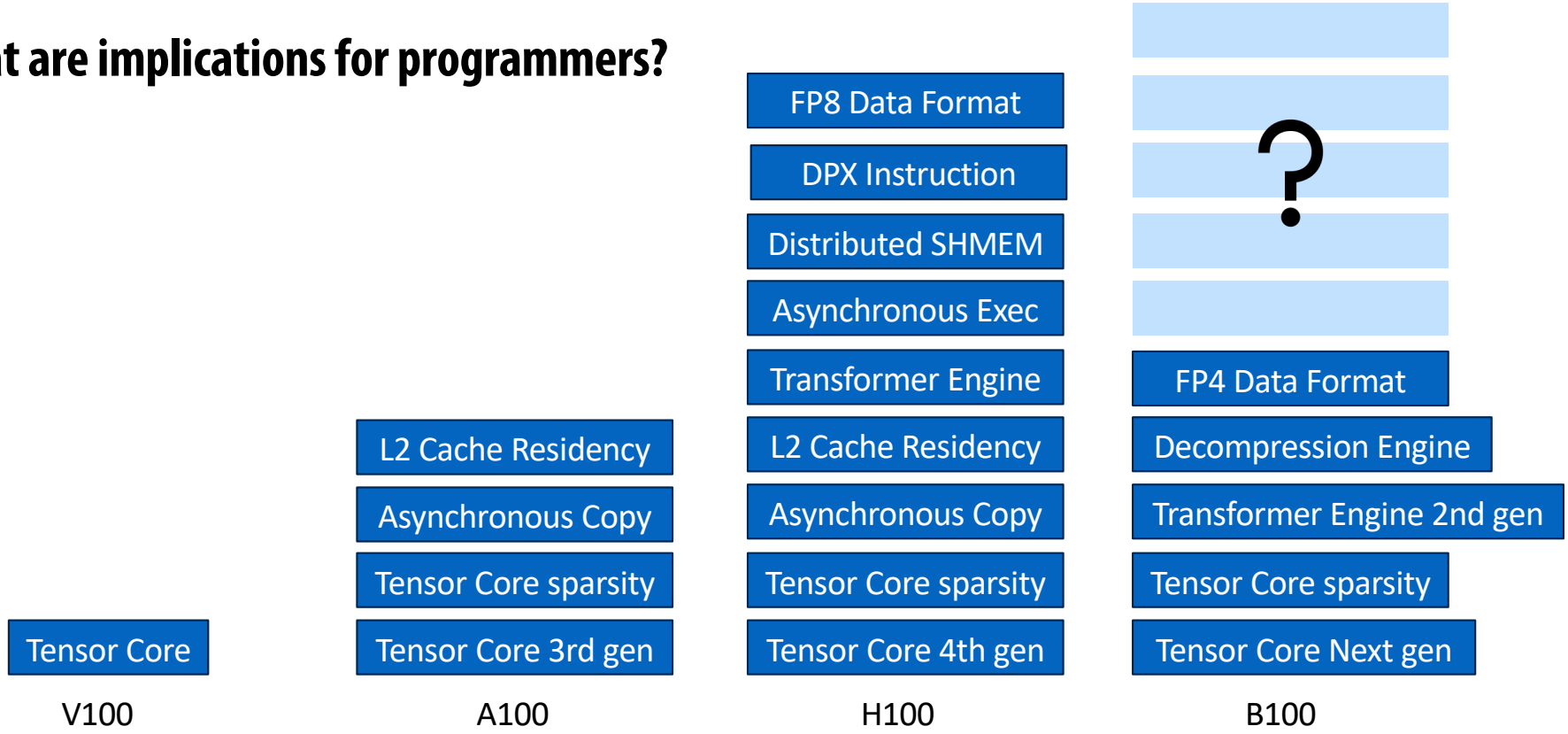


All the TFLOPS are in the Tensor Cores



Nvidia Chips Becoming More Specialized

What are implications for programmers?



DSLs for GPU AI Kernels

ThunderKittens: Simple, Fast, and *Adorable* AI Kernels

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré

Stanford University

```
@cute.jit
def block_reduce(val: cute.Numeric,
                 op: Callable,
                 reduction_buffer: cute.Tensor,
                 init_val: cute.Numeric = 0.0) -> cute.Numeric:
    lane_idx, warp_idx = cute.arch.lane_idx(), cute.arch.warp_idx()
    warps_per_row      = reduction_buffer.shape[1]
    row_idx, col_idx   = warp_idx // warps_per_row, warp_idx % warps_per_row
    if lane_idx == 0:
        # thread in lane 0 of each warp will write the warp-reduced value to the
reduction buffer
        reduction_buffer[row_idx, col_idx] = val
    # synchronize the write results
    cute.arch.barrier()
    block_reduce_val = init_val
    if lane_idx < warps_per_row:
        # top-laned threads of each warp will read from the buffer
        block_reduce_val = reduction_buffer[row_idx, lane_idx]
    # then warp-reduce to get the block-reduced result
    return warp_reduce(block_reduce_val, op)
```

Cute-DSL
(CUTLASS in Python)

Mojo🔥

```
@parameter
for n_mma in range(num_n_mmas):
    alias mma_id = n_mma * num_m_mmas + m_mma

    var mask_frag_row = mask_warp_row + m_mma *
MMA_M
    var mask_frag_col = mask_warp_col + n_mma *
MMA_N

    @parameter
    if is_nvidia_gpu():
        mask_frag_row += lane // (MMA_N //
p_frag_simdwidth)
        mask_frag_col += lane * p_frag_simdwidth %
MMA_N
    elif is_amd_gpu():
        mask_frag_row += (lane // MMA_N) *
```

Mosaic GPU

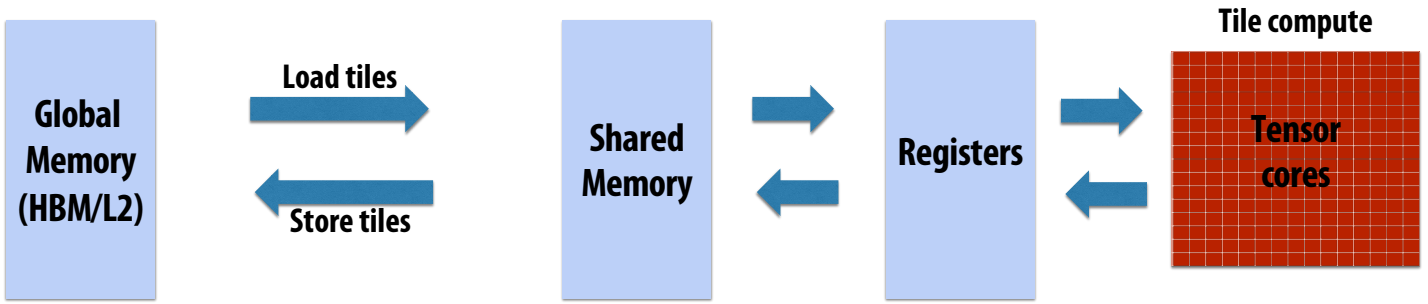
```
buffers = 3  # In reality you might want even more
assert a_smem.shape == (buffers, m, k)
assert b_smem.shape == (buffers, k, n)
assert acc_ref.shape == (m, n)

def fetch_a_b(ki, slot):
  a_slice = ... # Replace with the right M/K slice
  b_slice = ... # Replace with the right K/N slice
  plgpu.copy_gmem_to_smem(a_gmem.at[a_slice], a_smem.at[slot], a_loaded.at[slot])
  plgpu.copy_gmem_to_smem(b_gmem.at[b_slice], b_smem.at[slot], b_loaded.at[slot])

def loop_body(i, _):
  slot = jax.lax.rem(i, buffers)
  plgpu.barrier_wait(a_loaded.at[slot])
  plgpu.barrier_wait(b_loaded.at[slot])
  plgpu.wgmma(acc_ref, a_smem.at[slot], b_smem.at[slot])
  # We know that only the last issued WGMMA is running, so we can issue a async load in
  # into the other buffer
  load_i = i + buffers - 1
  load_slot = jax.lax.rem(load_i, buffers)
  @pl.when(jnp.logical_and(load_i >= buffers, load_i < num_steps))
  def _do_fetch():
    fetch_a_b(load_i, slot)
for slot in range(buffers):
  fetch_a_b(slot, slot)
jax.lax.fori_loop(0, num_steps, loop_body, None)
```

Extracting Peak Performance from the H100

Kernels that keep the Tensor cores busy (>90% of TFLOPS)

- Use 16 x 16 tiles of fp16 data ⇒ matches Tensor core compute
- Make sure compute is never idle
- Overlap memory access and compute ⇒ use asynchrony

A tile processing pipeline



ThunderKittens

A Simple Embedded DSL for AI kernels

Ben Spector et. al.

- Design principle #1: tile of 16x16 as primitive data type
  - TK manages layouts
  - TK provides basic operations
- Design principle #2: Asynchrony, everywhere
  - Expose primitives for user to manage, if top performance needed
- Design principle #3: High-level GPU coordination patterns
  - Producer-consumer processing

Embedded CUDA DSL template library

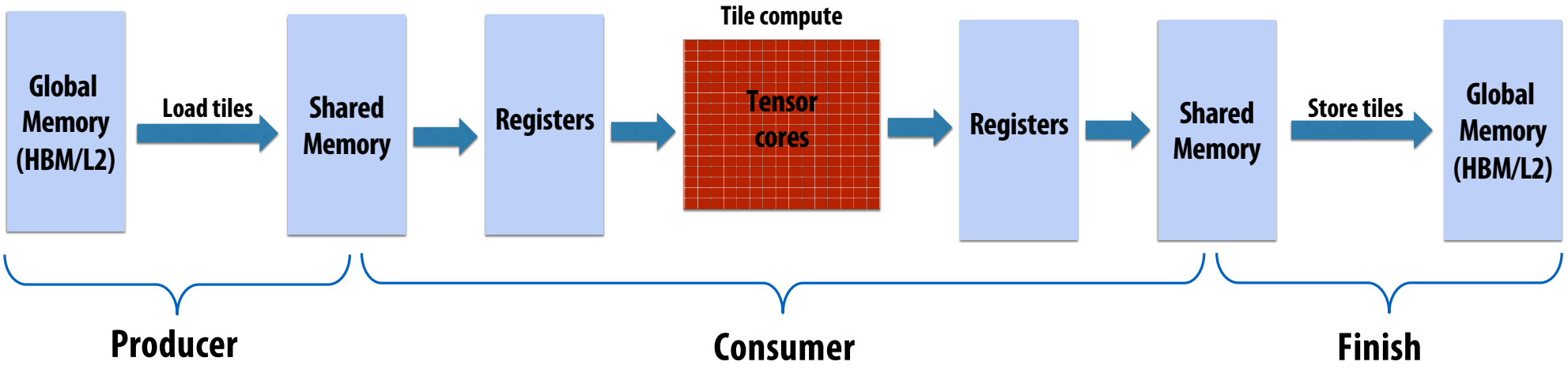
Templated Data Types

- Register tiles: 2D tensors on the register file
  - height, width, and layout
- Register vectors: 1D tensors on the register file
  - length and layout
- Shared memory tiles: 2D tensors in shared memory
  - height, width, and layout
- Shared memory vectors: 1D tensors in shared memory
  - Length

Operations

- Initializers -- zero out a shared vector, for example.
- Unary ops, like exp
- Binary ops, like mul
- Row / column ops, like a row_sum

Tile Processing Pipeline with ThunderKittens



TK Matmul

Step 1: Define layouts

```
#include "kittens.cuh"
#include "prototype.cuh"

using namespace kittens;
using namespace kittens::prototype;
using namespace kittens::prototype::lcf;
struct matmul_layout {
   using  a_global_layout = gl<bf16, 1, 1, -1, -1, st_bf<64, 64>>; // create a TMA descriptor for a 64x64 tile
   using  b_global_layout = gl<bf16, 1, 1, -1, -1, st_bf<64, 256>>; // create a TMA descriptor for a 64x256 tile
   using  c_global_layout = gl<bf16, 1, 1, -1, -1>; // no TMA descriptor needed for C
   struct globals        { a_global_layout A; b_global_layout B; c_global_layout C; };
   struct input_block    { st_bf<64, 64> a[2]; st_bf<64, 256> b; } // shared memory tile for input
   struct finish_block   { st_bf<64, 256> c[2]; }; // shared memory tiles for result
   struct consumer_state { rt_fl<16, 256> accum; }; // register tile
};
```

TK Matmul

Step 2: Define pipeline and producers

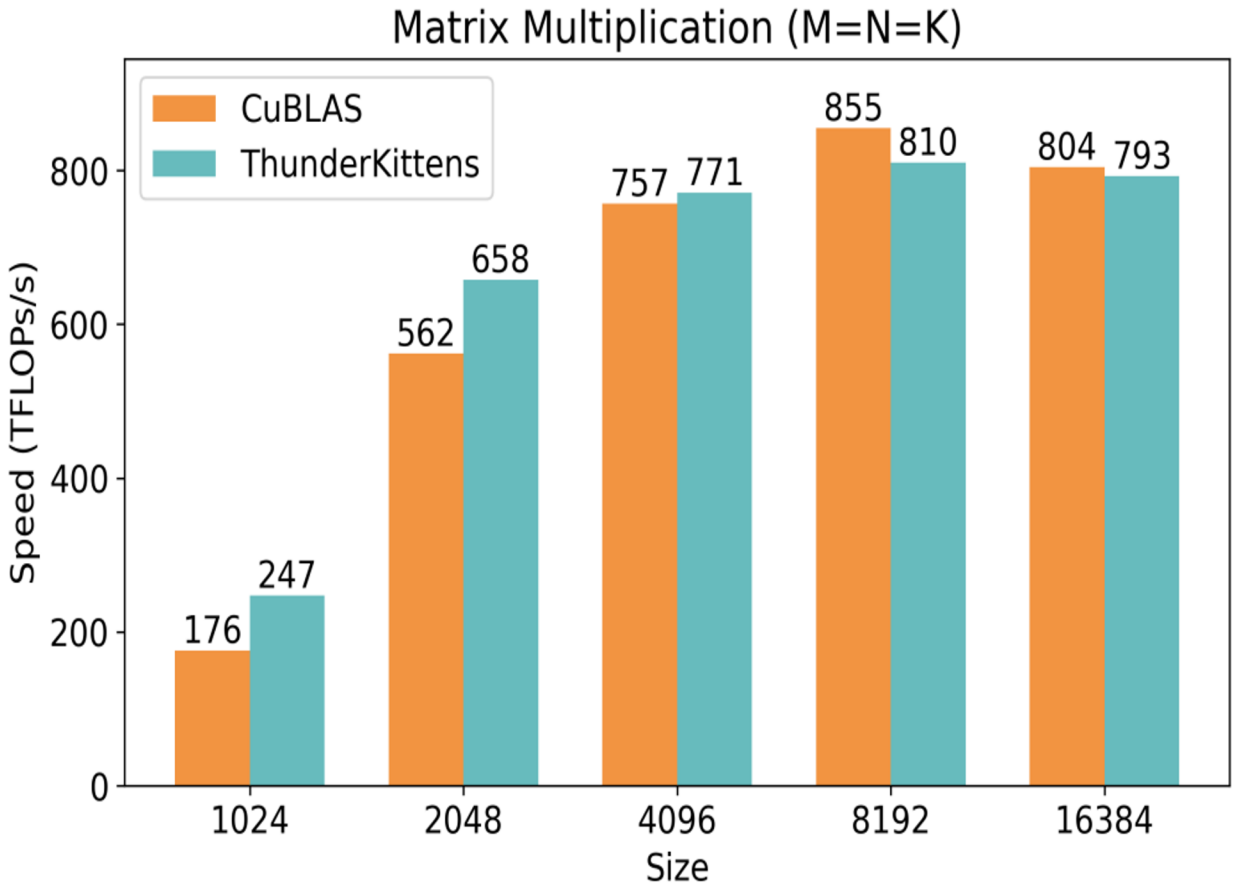
```
struct matmul_template {
   using layout = matmul_layout;
   static constexpr int NUM_CONSUMER_WARPS=8, INPUT_PIPE_STAGES=4; // 8 active consumer warps, 4 active producer warps (default), and a 4-stage pipeline
   static constexpr int PRODUCER_BARRIER_ARRIVALS=1, CONSUMER_BARRIER_ARRIVALS=2; // Producers need to arrive just once, and each consumer wargroups arrives.
   __device__ static inline void common_setup(common_setup_args<layout> args) {
       args.num_iters = args.task_iter == 0 ? args.globals.A.cols/64 : -1; // Tell the template we have a single task of (reduce dim) / 64 tiles to handle.
   }
   struct producer {
       __device__ static void setup(producer_setup_args<layout> args) {
           warpgroup::decrease_registers<40>(); // decrease registers for producers, to leave more for the consumers.
       }
       __device__ static void load(producer_load_args<layout> args) { // Template waits for the input block to be ready to write before launching
           if(warpgroup::warpid() == 0) { // We only actually need one warp (in fact, one thread) to tell TMA to go launch loads
               tma::expect(args.inputs_arrived, args.input); // Tell the mbarrier sempahore how many bytes to expect (inferred from the input struct type)
               for(int i = 0; i < 2; i++) { // Load the A tiles -- one per consumer wargroup -- for this input phase. Each is 64x64, strided vertically.
                   tma::load_async(args.input.a[i], args.globals.A, {blockIdx.x*2+i, args.iter}, args.inputs_arrived);
               }
               // Load the B tile for this input phase (just one 64x256 tile, shared by all consumer wargroups)
               tma::load_async(args.input.b, args.globals.B, {args.iter, blockIdx.y}, args.inputs_arrived);
           }
       }
   };
```

TK Matmul

Step 3: Compute!

```
struct consumer {
       __device__ static void setup(consumer_setup_args<layout> args) {
           warpgroup::increase_registers<232>(); // increase registers for consumers
           zero(args.state.accum); // zero the matrix accumulators
       }
       __device__ static void compute(consumer_compute_args<layout> args) { // Template waits for input block to be ready to use first
           warpgroup::mma_AB(args.state.accum, args.input.a[warpgroup::groupid()], args.input.b);
           warpgroup::mma_async_wait();
           if(warpgroup::laneid() == 0) arrive(args.inputs_finished); // A single thread marks that the memory is now finished.
       }
       __device__ static void finish(consumer_finish_args<layout> args) {
           int wg = warpgroup::groupid(); // Which consumer warpgroup worker am I?
           warpgroup::store(args.finish.c[wg], args.state.accum);
           warpgroup::sync(); // storing to shared memory first reorganizes for better coalescing to HBM
           warpgroup::store(args.globals.C, args.finish.c[wg], args.state.accum, {blockIdx.x*2+wg, blockIdx.y});
       }
   };
};
```

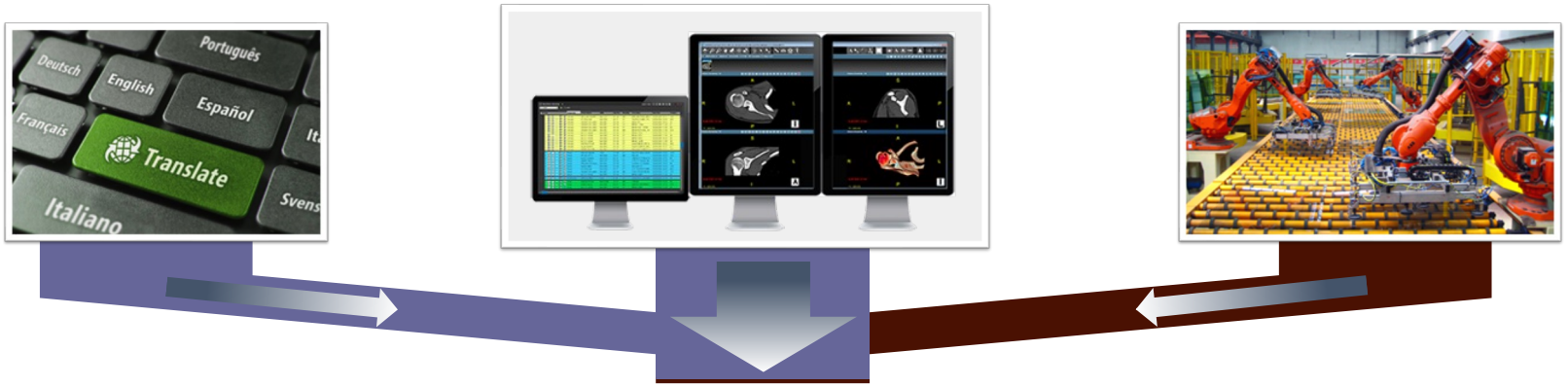
TK Matmul Performance



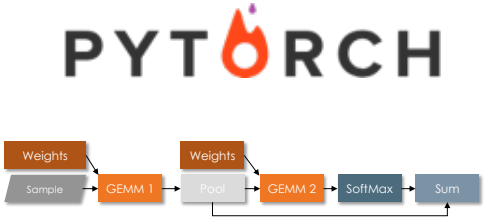
Can we have asynchrony with a simpler programming model?

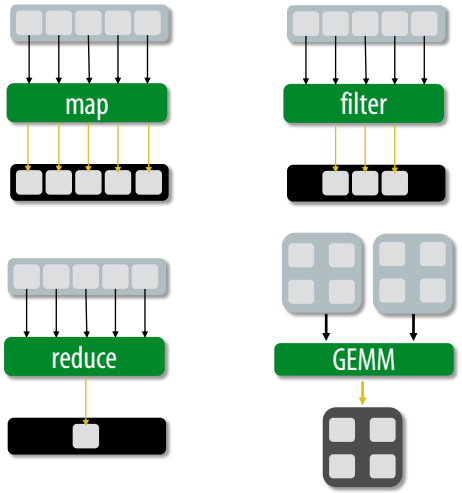
(Hint: Take a data-centric view)

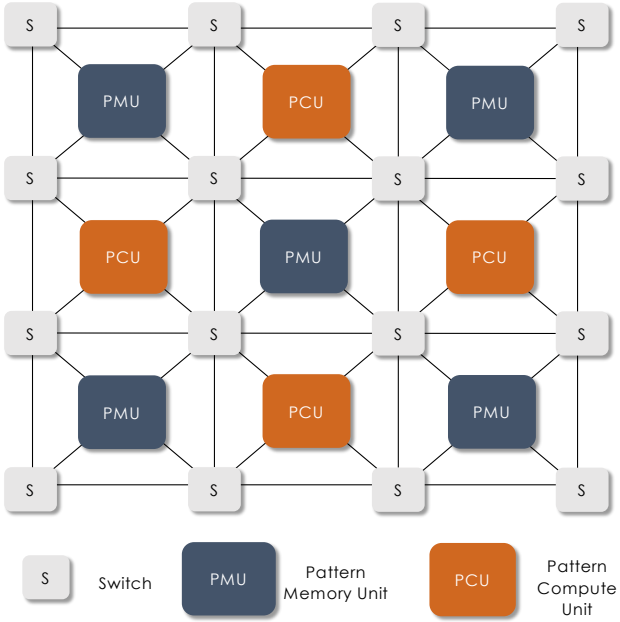
Recall: AI Models are Dataflow Graphs



AI Models ⇒ Dataflow Architecture







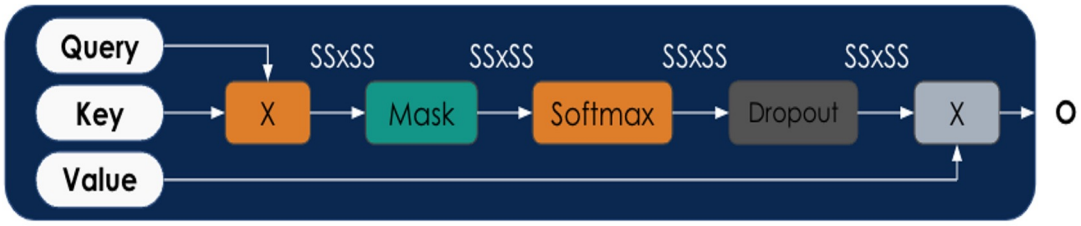
AI Models

Dataflow graph:
GEMM + Parallel Patterns

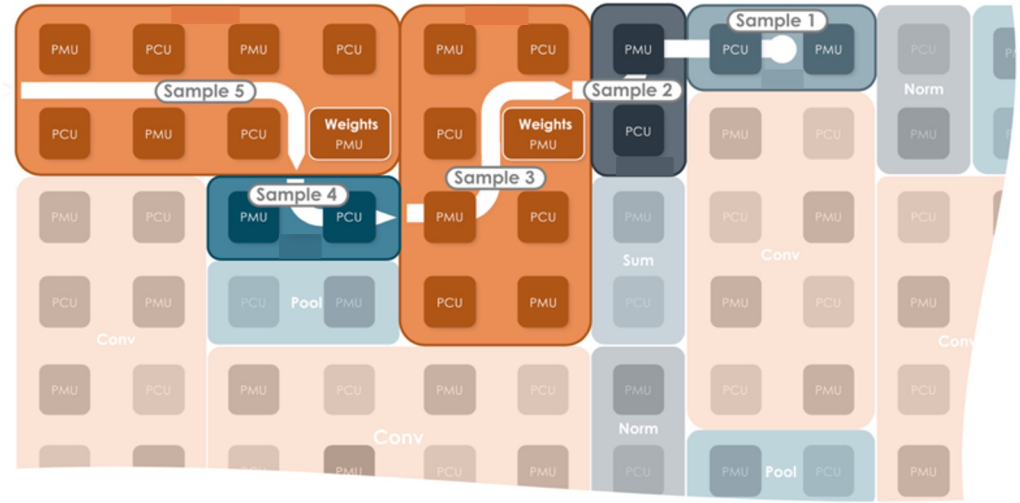
Plasticine
Reconfigurable Dataflow Architecture

Prabhakar, Zhang, et. al. ISCA 2017

Streaming Dataflow ⇒ Kernel Fusion



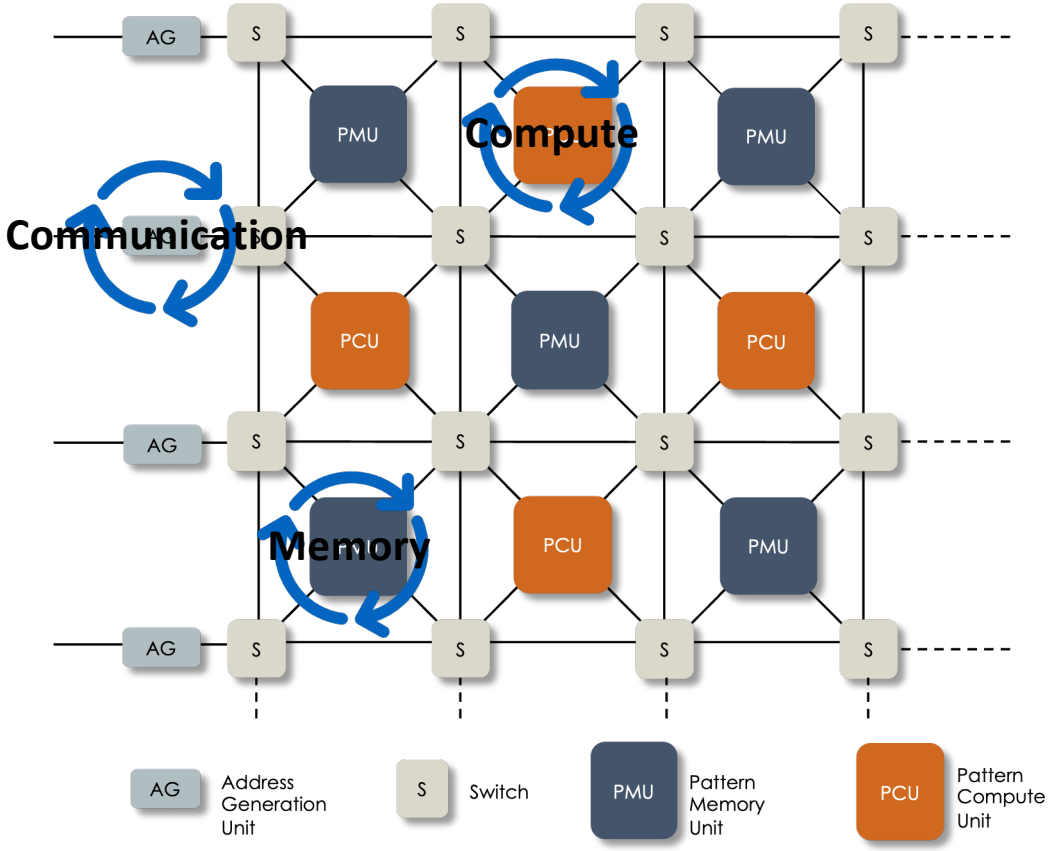
Attention Algorithm



Attention Algorithm on RDA

Coarse -grained pipelining

Reconfigurable Dataflow Architecture vs Ideal Accelerator

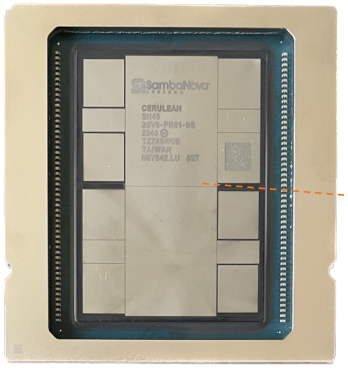


Feature	Why?
Tiled tensors (e.g. 16 x 16, 32 x 32)	Max TFLOPS on GEMM Low instr. overhead
Asynchronous compute	Overlap compute and memory access
Asynchronous memory access	Overlap compute and memory access
Asynchronous chip-to-chip communication	Overlap compute, memory and communication
Compute unit to compute unit comm.	Fusion and pipelining Streaming Dataflow

No instructions ⇒ No instruction fetch/decode overhead

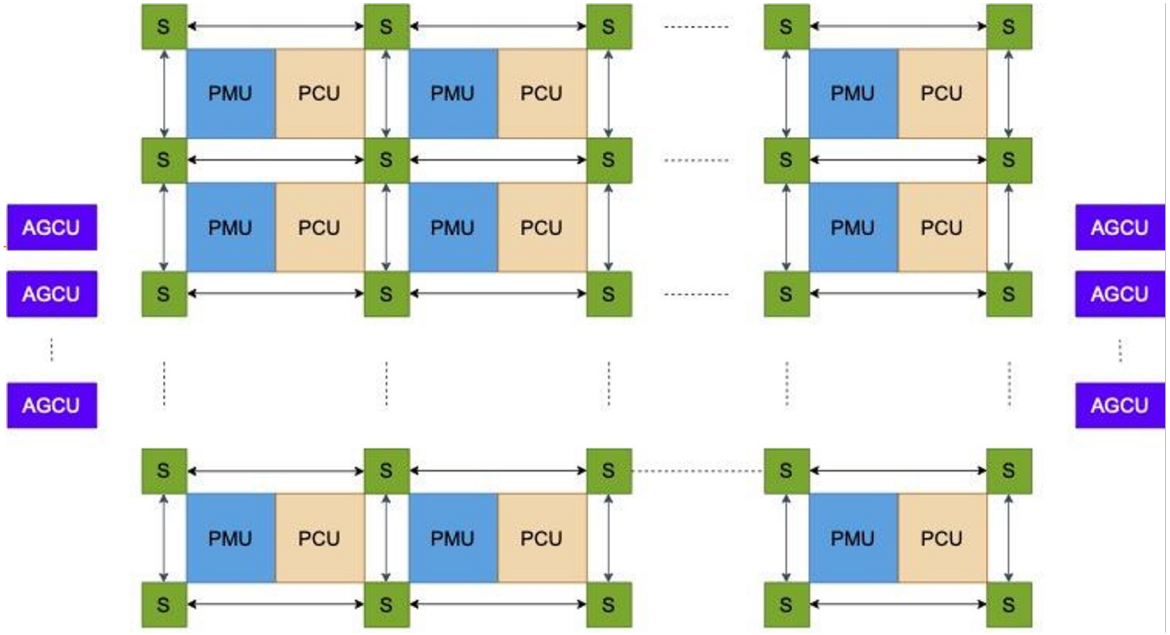
Extreme asynchrony: no sequential instruction execution

Reconfigurable Dataflow



SambaNova SN40L RDU

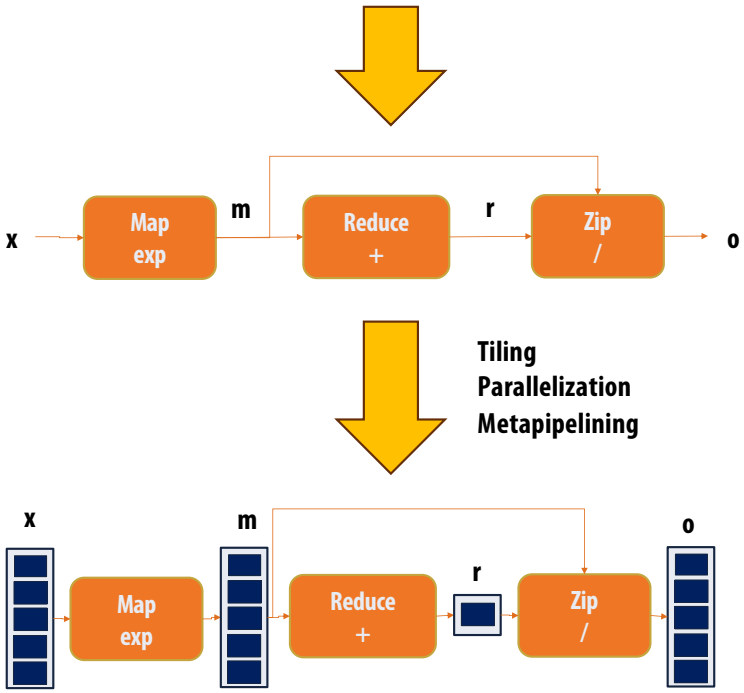
- 1,040 PCUs and PMUs
- 638 TFLOPS (bf16)
- 520 MB on-chip SRAM
- 64 GB HBM
- 1.5 TB DDR

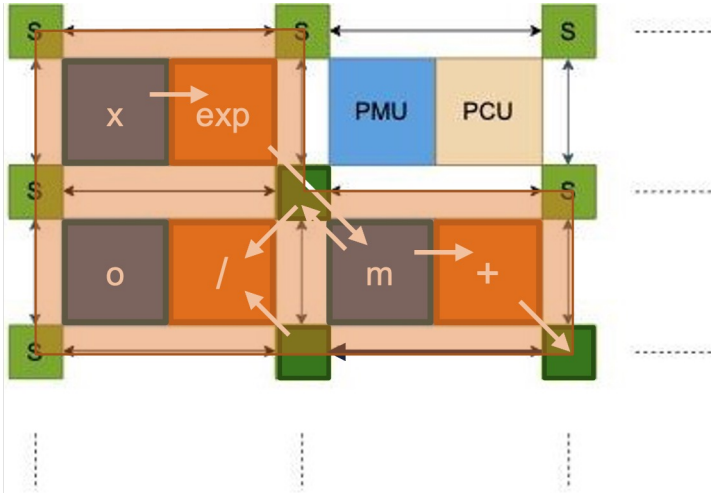


- PCU: Pattern Compute Unit
  - systolic and SIMD compute (16 x 8 bf16)
- PMU: Pattern Memory Unit
  - High address generation flexibility and bandwidth (0.5 MB)
- S: Mesh switches
  - High on-chip interconnect flexibility and bandwidth
- AGCU: Address Generator and Coalescing Unit
  - Portal to off-chip memory and IO

Dataflow Programming with Data Parallel Patterns

SIMPLIFIED SOFTMAX $\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$





Place & Route
Codegen

- **Composable Compute Primitives: MM, Map, Zip, Reduce, Gather, Scatter …**
- **Flexible scheduling in space and time ⇒ spatial execution**

Metapipelining

Hierarchical coarse-grained pipeline: A "pipeline of pipelines"

- **Exploits nested-loop parallelism**

Convert parallel pattern (loop) into a streaming pipeline

- **Insert pipe stages in the body of the loop**
- **Pipe stages execute in parallel**
- **Overlap execution of multiple loop iterations**

Intermediate data between stages stored in double buffers

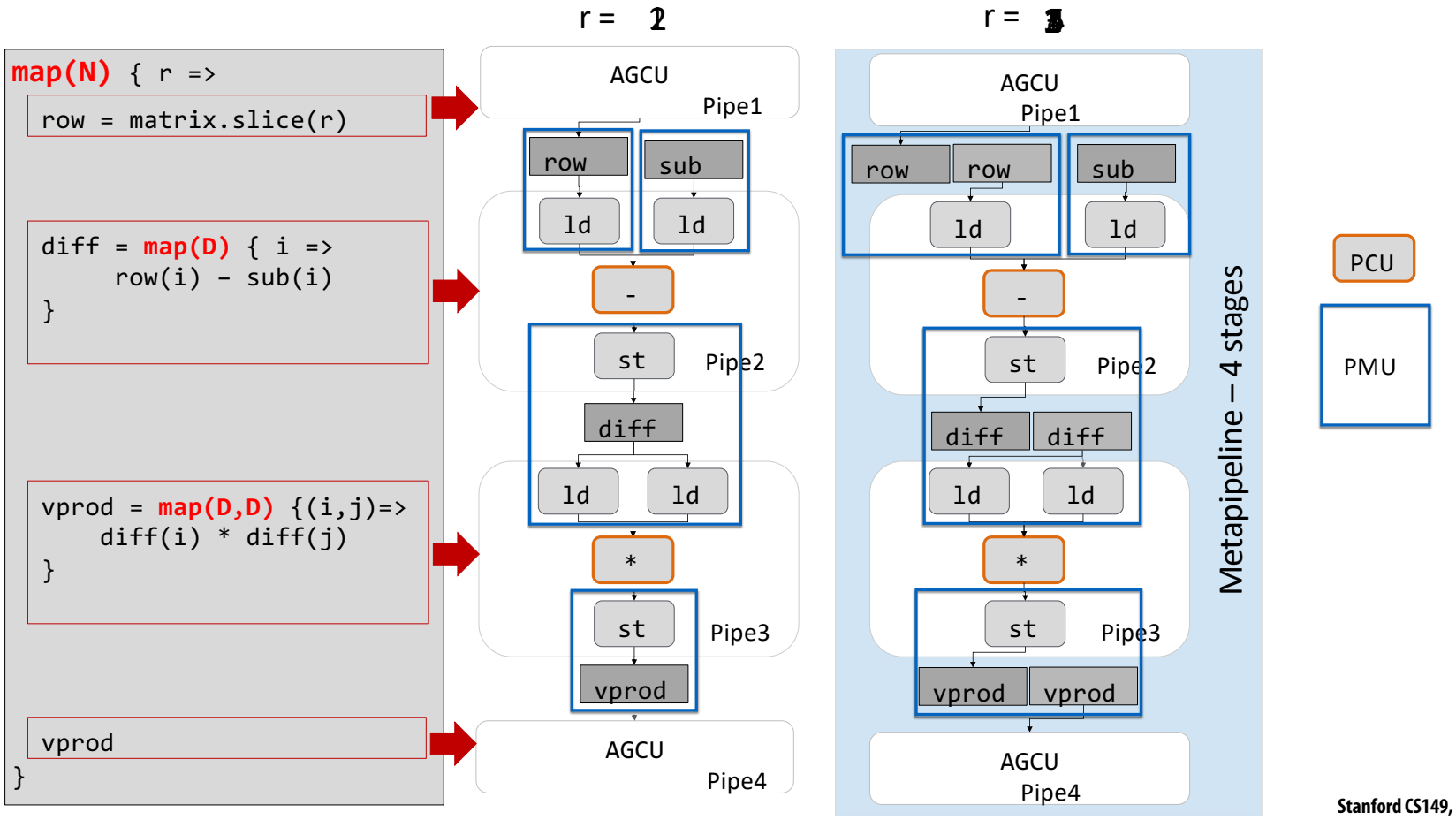
- **Handles imbalanced stages with varying execution times**

Tiling and fusion

- **Works well with tiling**
- **Buffers can be used to change access pattern (e.g. transpose data)**
- **Metapipelining can work when fusion does not**

Metapipelining Intuition

Gaussian Discriminant Analysis (GDA)

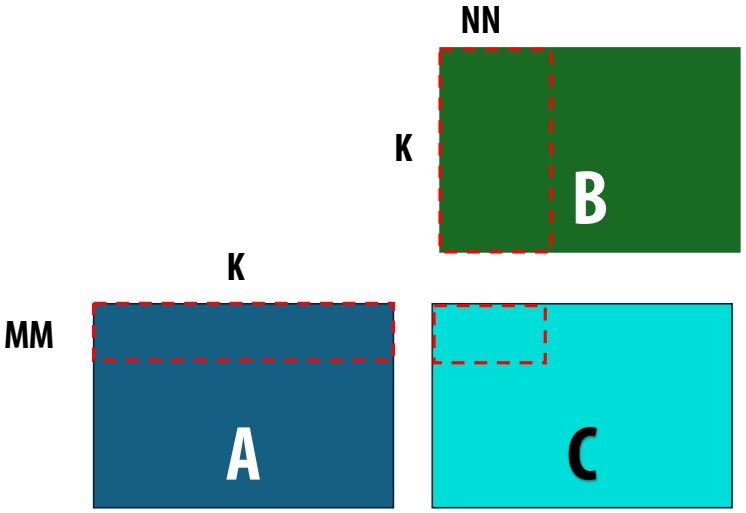


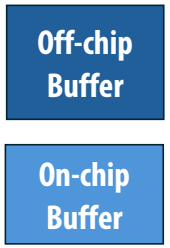
Matmul Metapipeline

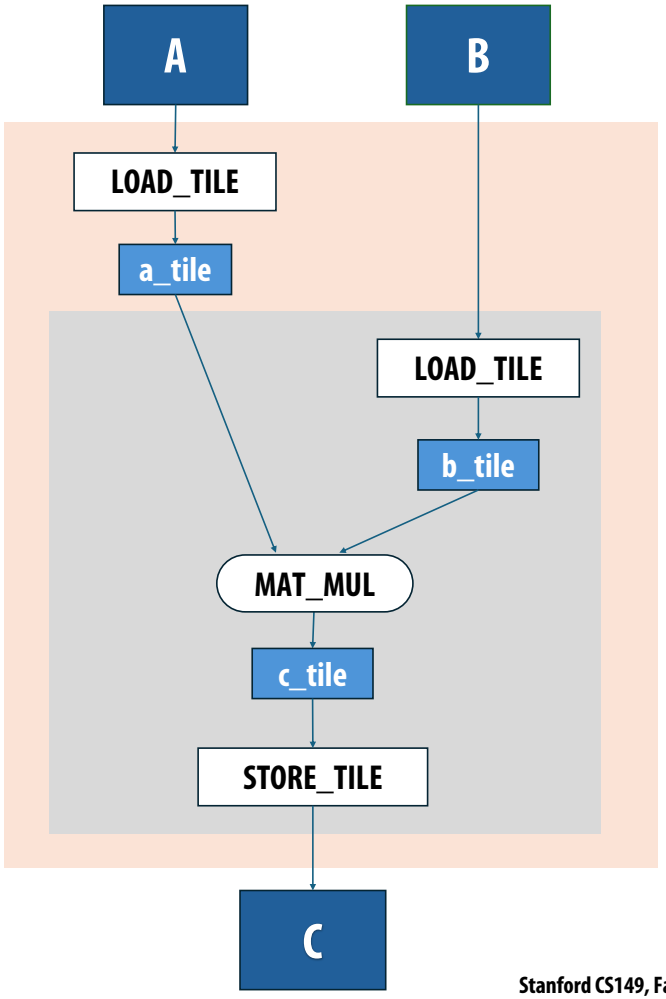
```
auto format = DataFormat::kBF16;

int64_t M = args::M.getValue();
int64_t N = args::N.getValue();
int64_t K = args::K.getValue();


auto A = INPUT_REGION("A", (M, K), format);
auto B = INPUT_REGION("B", (K, N), format);
auto C = OUTPUT_REGION("C", (M, N), format);

auto MM = 256; // Tile size along M, assumes to evenly divide M
auto NN =  64; // Tile size along N, assumes to evenly divide N

auto a_tile_shape = std::vector<int64_t>({MM, K});
auto b_tile_shape = std::vector<int64_t>({K, NN});
auto c_tile_shape = std::vector<int64_t>({MM, NN});

METAPIPE(M / MM, [&]() {
     auto a_tile = LOAD_TILE(A, a_tile_shape);
     METAPIPE(N / NN, [&]() {
          auto b_tile = LOAD_TILE(B, b_tile_shape, row_par = 4);
          auto c = MAT_MUL(a_tile, b_tile);
        auto c_tile = BUFFER(c);
          STORE_TILE(C, c_tile);
     });
});
```

Matmul Metapipe

```
METAPIPE(M, MM) {
    a_tile = LOAD_TILE(A, a_tile_shape)
    METAPIPE(N, NN) {
        b_tile = LOAD_TILE(B, b_tile_shape)
        c = MAT_MUL(a_tile, b_tile, row_par = 4)
        c_tile = BUFFER(c)
        STORE_TILE(C,c_tile)
    }
}
```

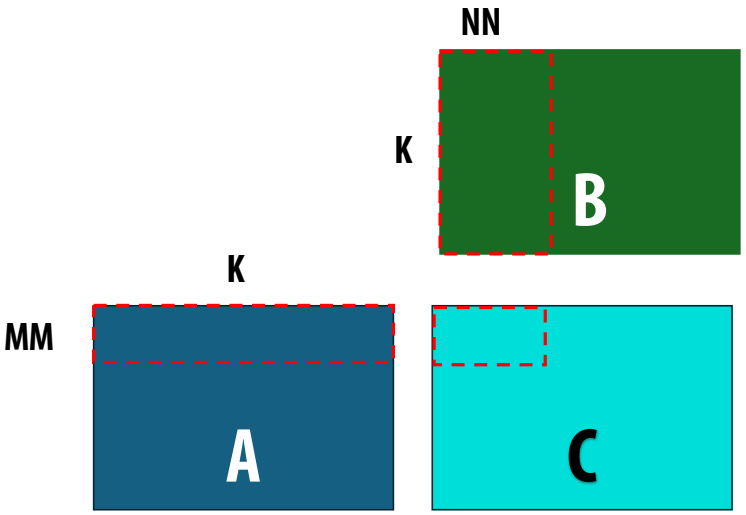


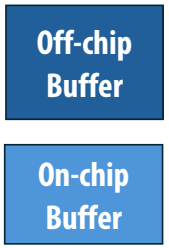


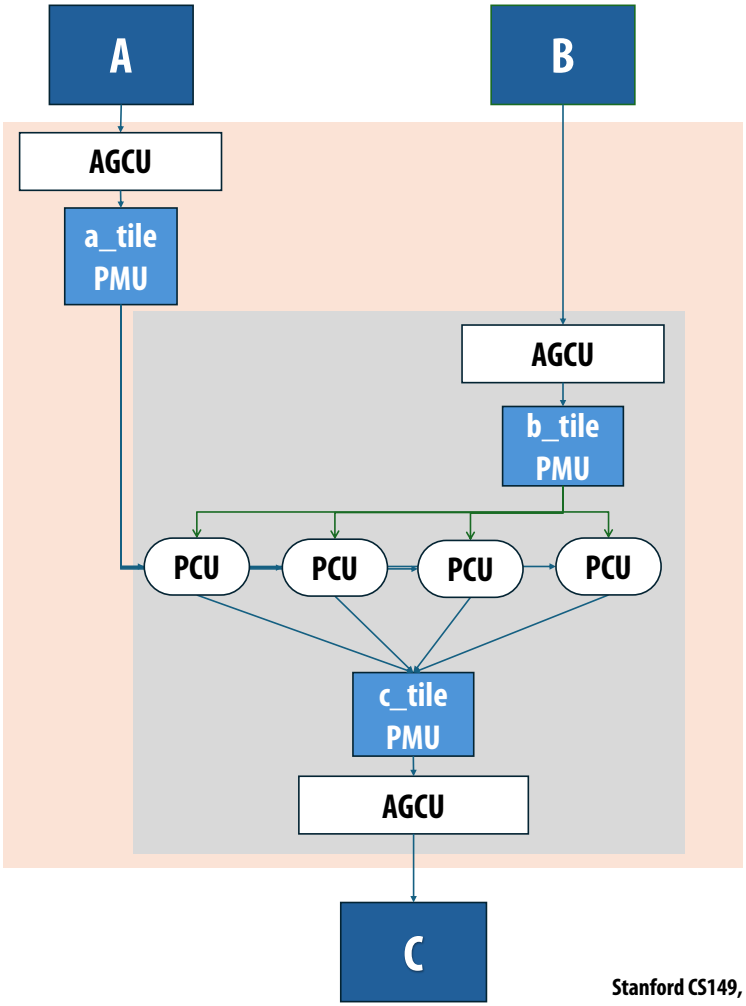


Matmul Metapipe Mapping

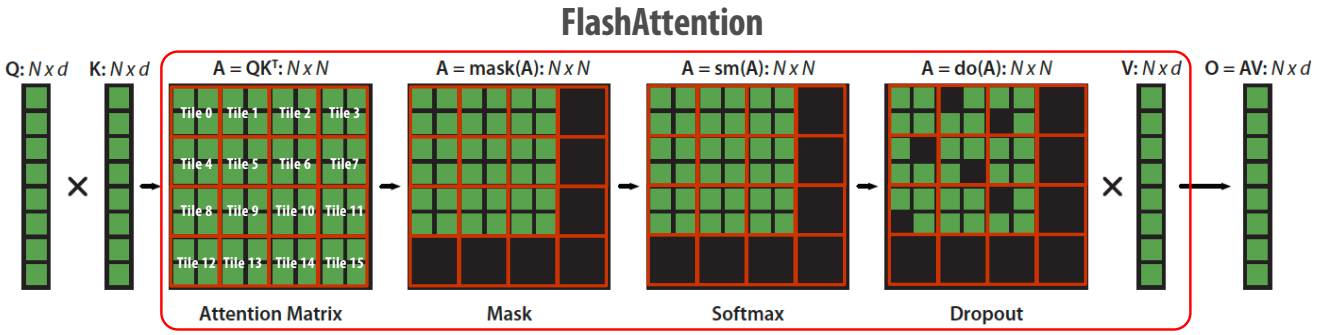
```
METAPIPE(M, MM) {
    a_tile = LOAD_TILE(A, a_tile_shape)
    METAPIPE(N, NN) {
        b_tile = LOAD_TILE(B, b_tile_shape)
        c = MAT_MUL(a_tile, b_tile, row_par = 4)
        c_tile = BUFFER(c)
        STORE_TILE(C,c_tile)
    }
}
```

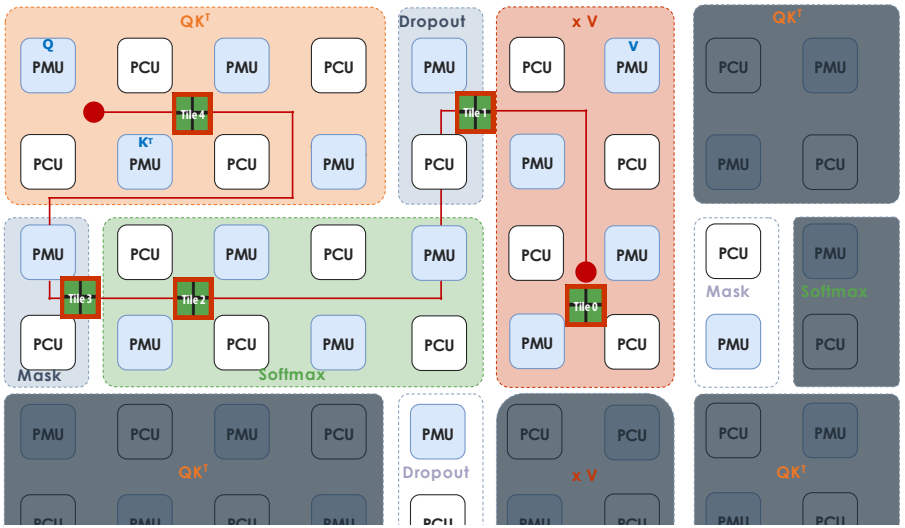




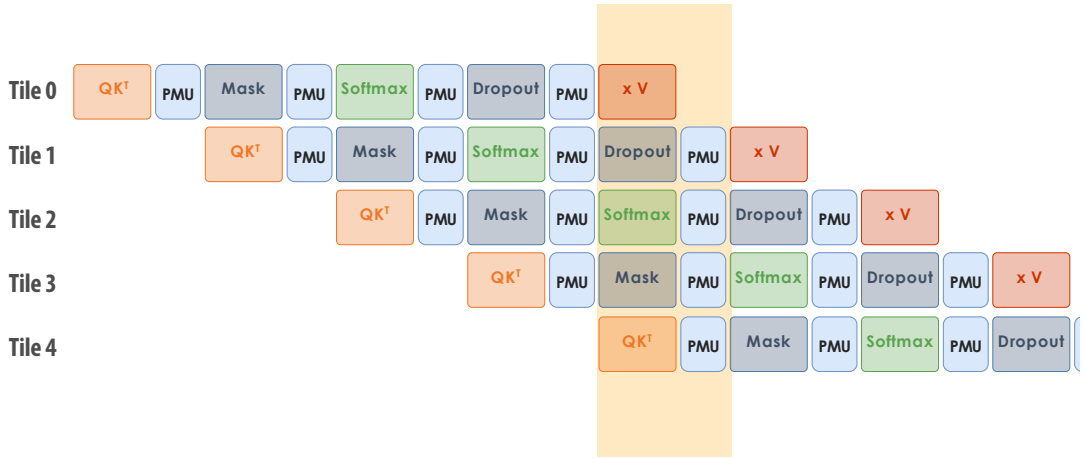


FlashAttention Metapipeline



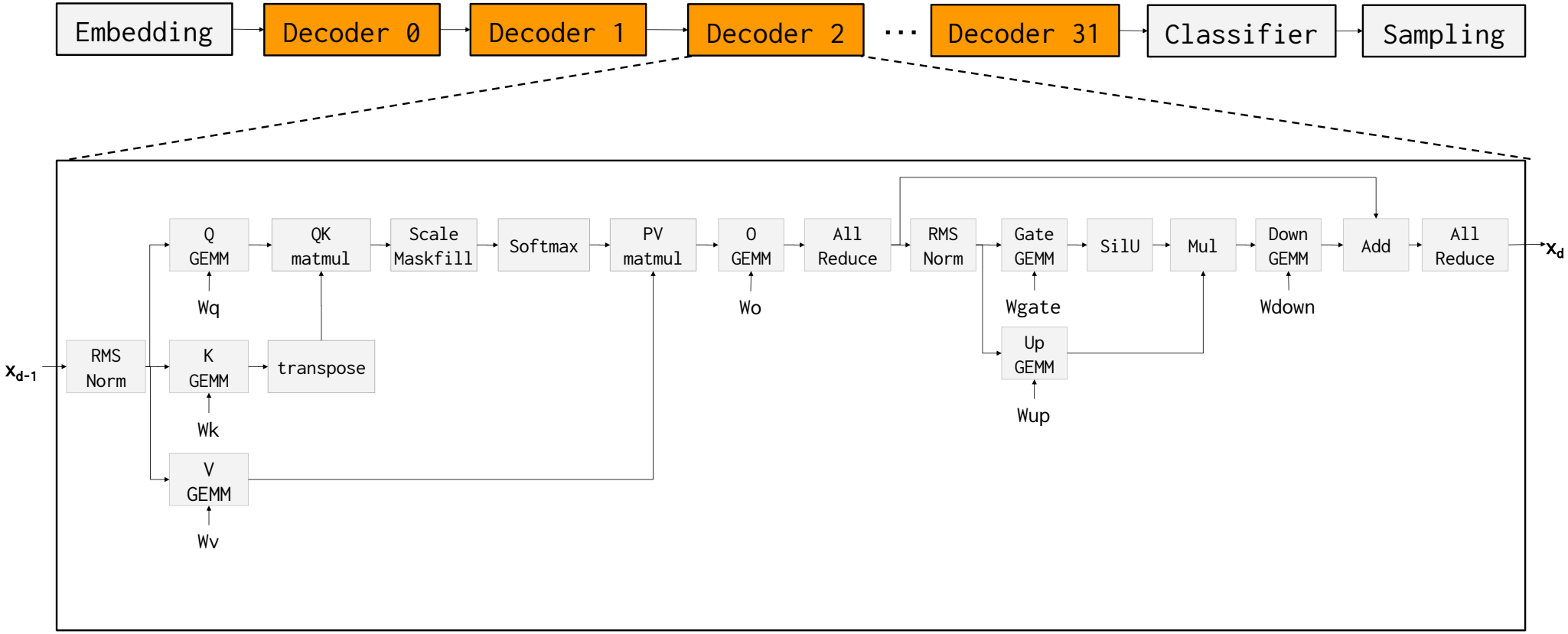


Dataflow execution with token control ⇒ no lock-based synchronization



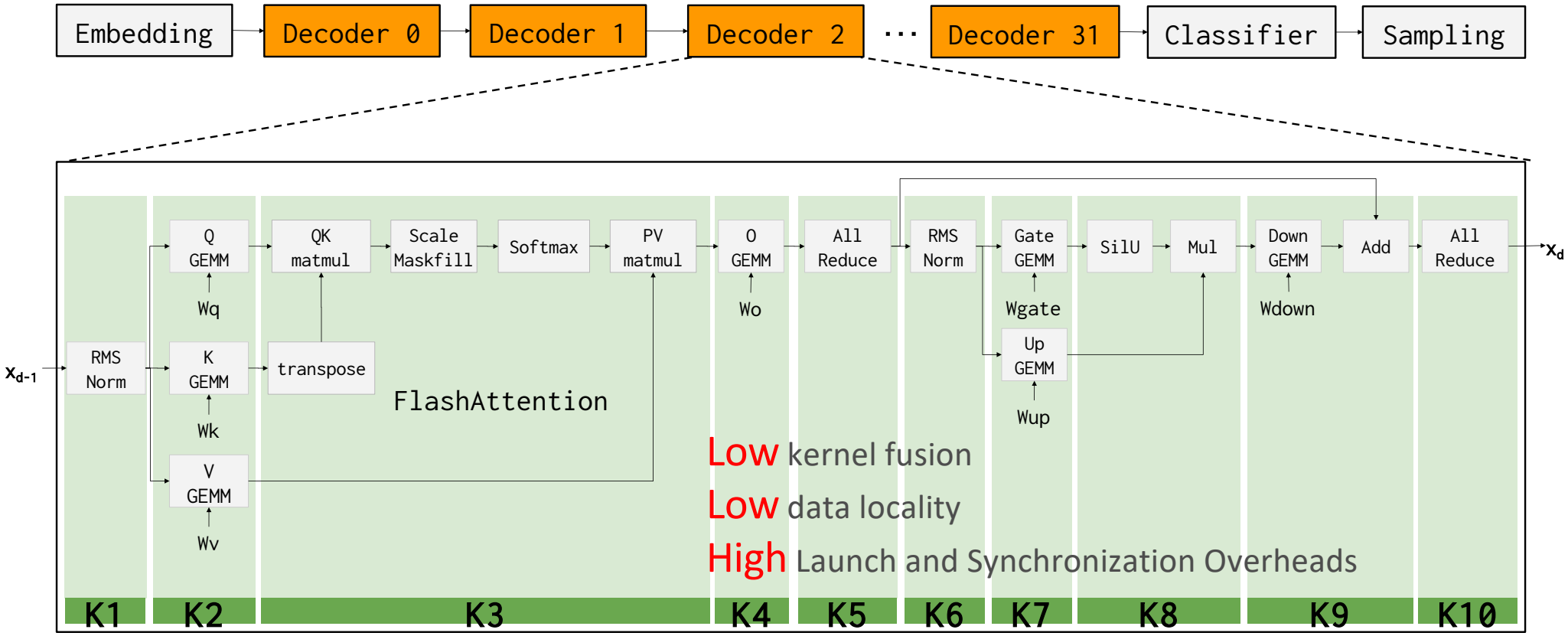
MetaPipeline = Streaming Dataflow

Llama3.1 8B


Embedding
Decoder 0
Decoder 1
Decoder 2
···
Decoder 31
Classifier
Sampling
x_{d-1}
RMS Norm
Q GEMM
Wq
K GEMM
Wk
transpose
V GEMM
Wv
QK matmul
Scale Maskfill
Softmax
PV matmul
O GEMM
Wo
All Reduce
RMS Norm
Gate GEMM
Wgate
Up GEMM
Wup
SiLU
Mul
Down GEMM
Wdown
Add
All Reduce
x_d

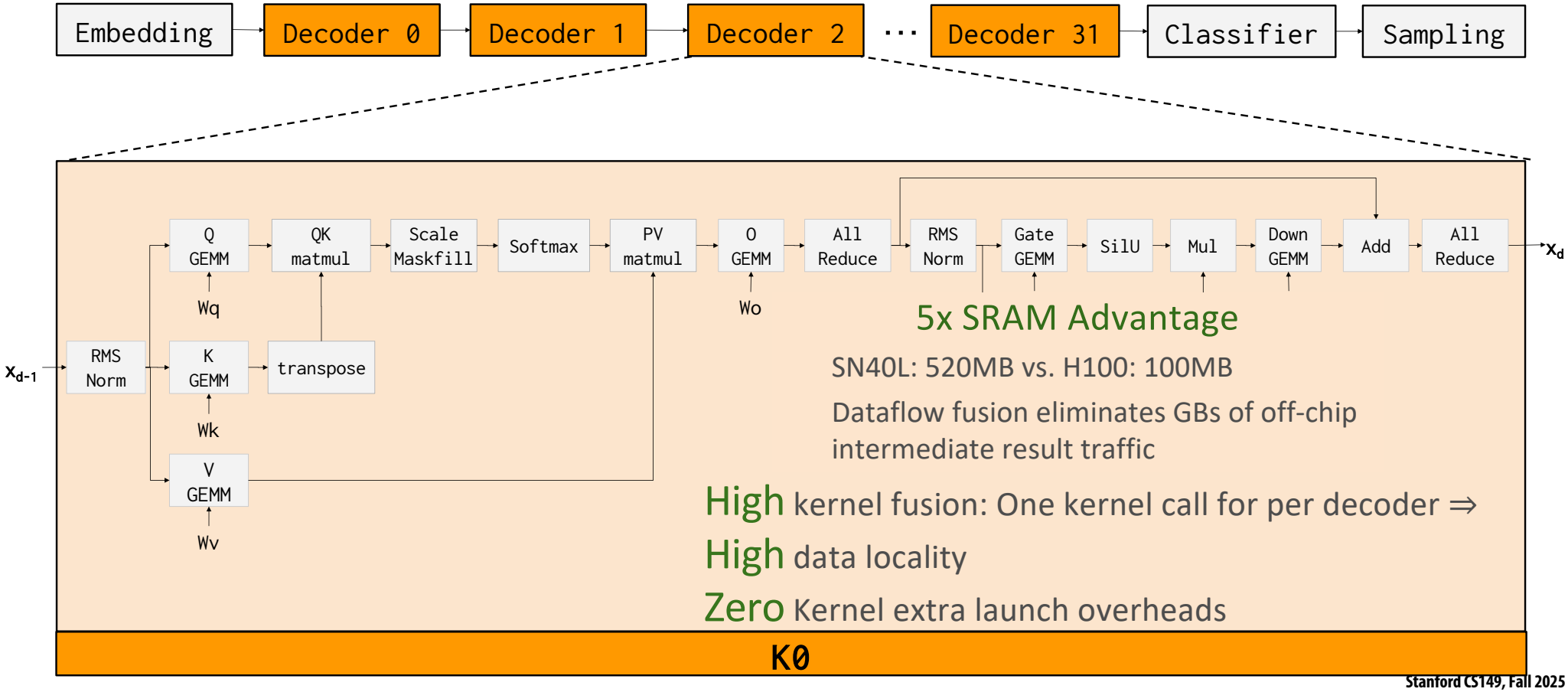
Limited Kernel Fusion on GPUs

Llama3.1 8B with Tensor-RT LLM



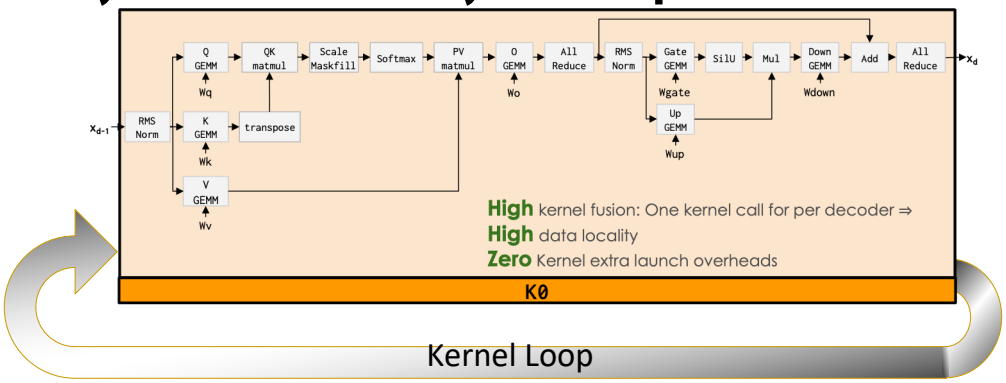
RDU Fuses Entire Decoder into One Kernel !

Llama3.1 8B with aggressive kernel fusion



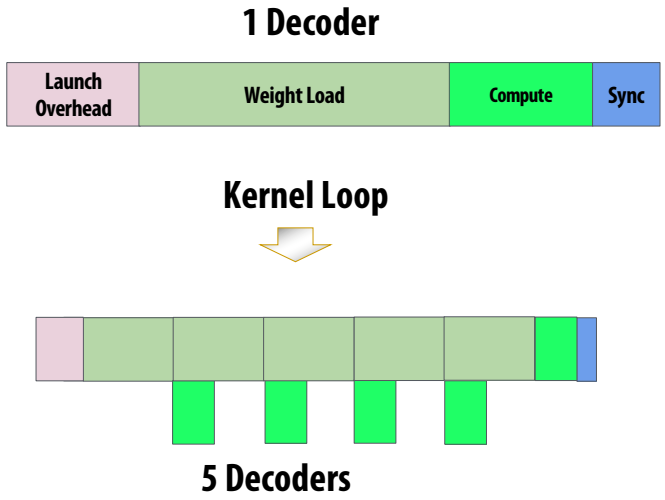
Kernel Loop

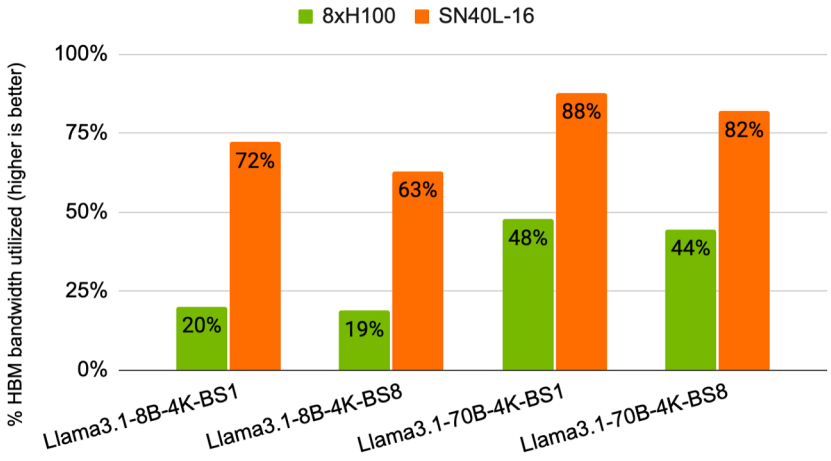
Asynchronous memory and compute



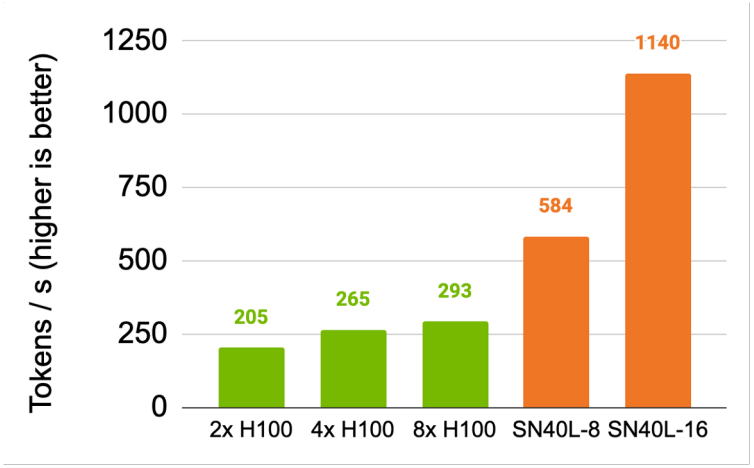
One kernel call for all decoders

- 3 calls per token on RDU
- ~800 calls per token on GPU
- 100x fewer kernel calls



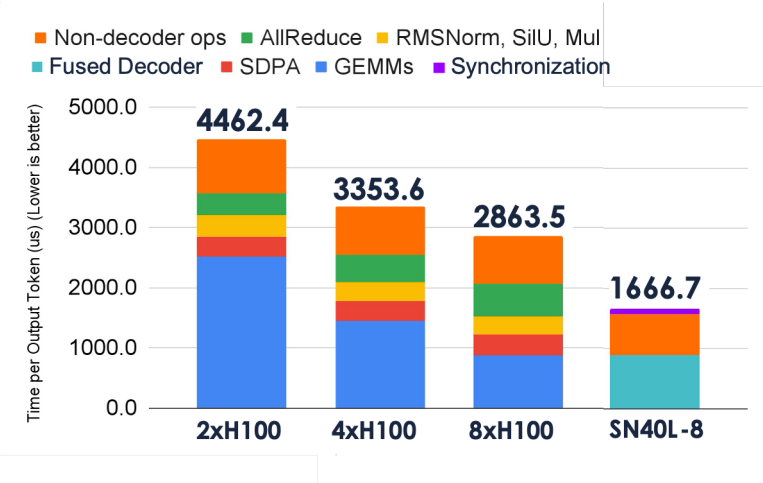


Dataflow ⇒ High Performance



Overlap compute, memory access, chip-to-chip communication

- Fully overlap allreduce with weight load and compute
- Allreduce does not consume HBM capacity or bandwidth



Summary: Specialized Hardware and Programming for AI Models

Specialized hardware for executing key AI computations efficiently

Feature large/many matrix multiply units implemented with systolic arrays

Customized/configurable datapaths to directly move intermediate data values between processing units (schedule computation by laying it out spatially on the chip)

Large amounts of on-chip storage for fast access to intermediates

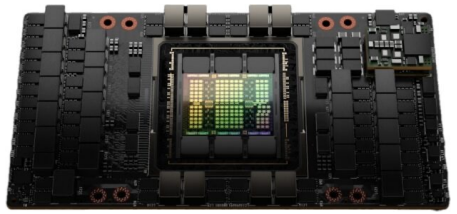
H100: Asynchronous compute and memory mechanisms ⇒ complex programming

- Need ThunderKittens and other DSLS to manage complexity

SN40L: Dataflow model with metapipelining ⇒ simpler programming model

- Sophisticated compiler to optimize and map to dataflow hardware

Minimizing synchronization overheads required for high performance



H100

SN40L

TPU supercomputer (1024 TPU v3 chips)