Lecture 12:

# Mapping AI Applications to the AI Datacenter

Parallel Computing
Stanford CS149, Fall 2025

# Today's Theme

How do you design specialized HW for DNNs?

How do you program specialized hardware?

Google TPU

- Efficient dense matrix multiply ⇒systolic array

Nvidia H100 and B100

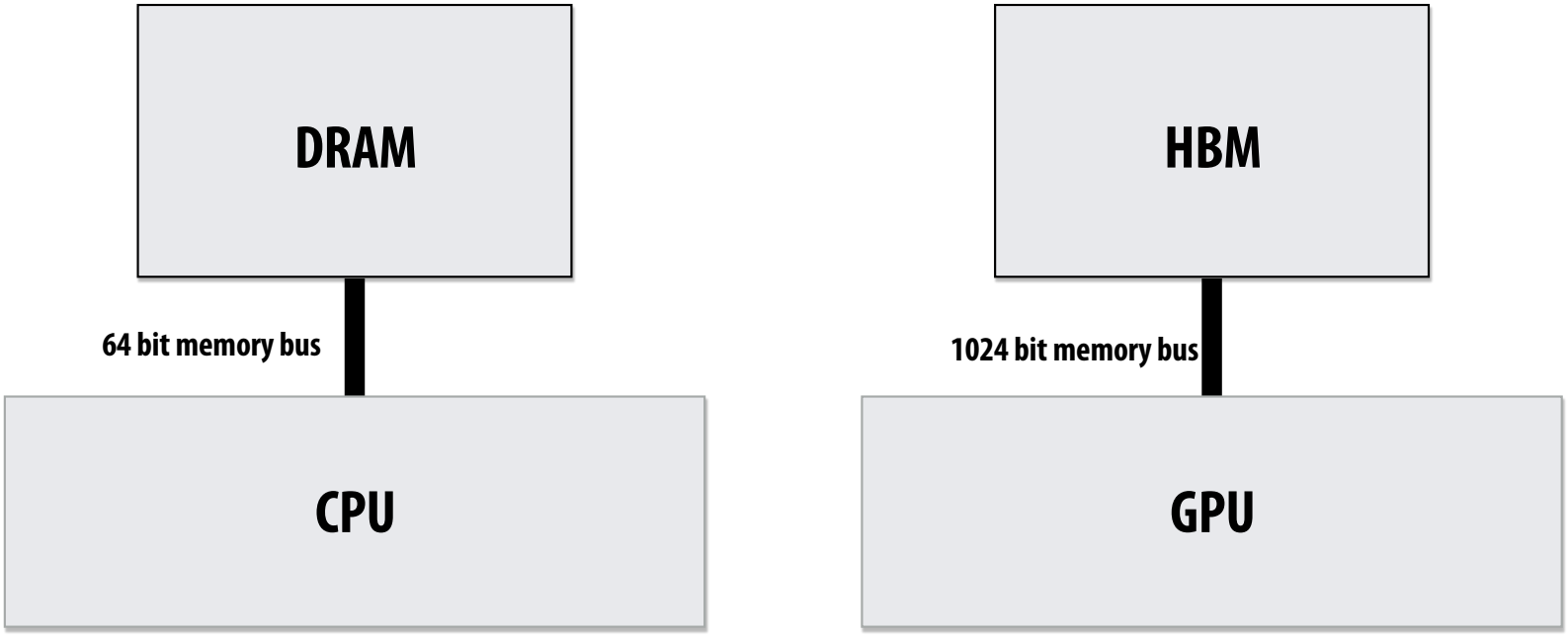
- Asynchronous compute and memory mechanisms ⇒ complex programing
- Simplify with Thunderkittens DSL

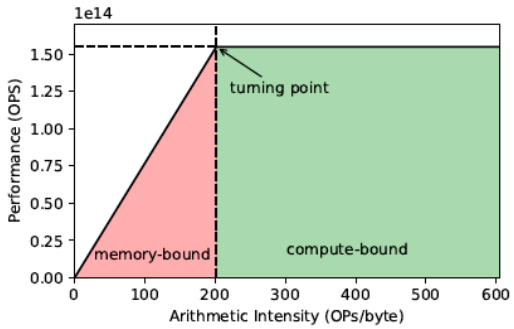
SambaNova SN40L

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

# Short Primer on Memory

# CPU vs GPU Memory

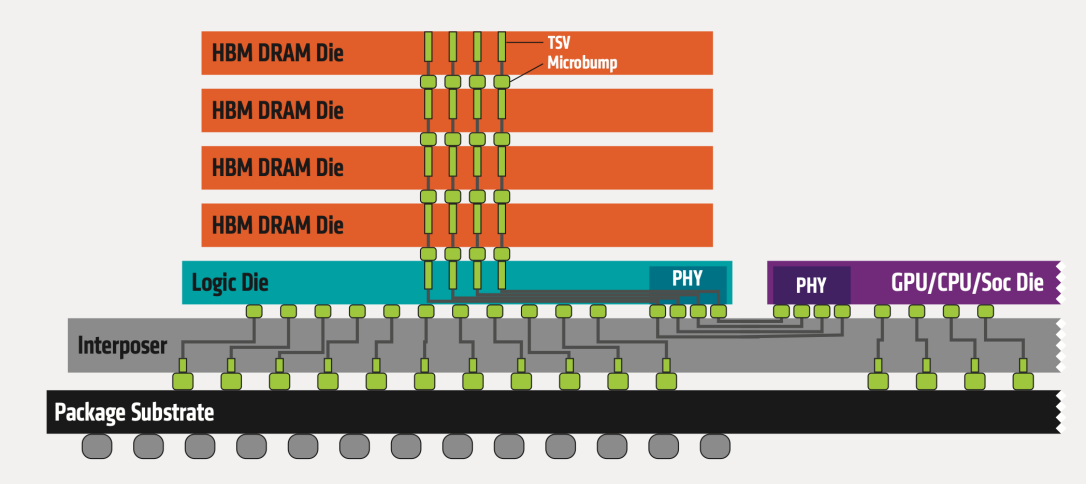


# Increase bandwidth, reduce power by chip stacking

## Enabling technology: 3D stacking of DRAM chips

- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
- TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack "logic layer" is memory controller, manages requests from processor
- Silicon "interposer" serves as high-bandwidth interconnect between DRAM stack and processor




**Technologies:**
**Micron/Intel Hybrid Memory Cube (HBC)**
**High-bandwidth memory (HBM) - 1024 bit interface to stack**

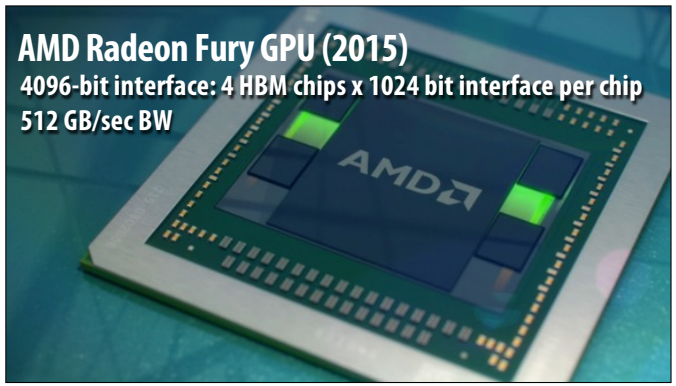
Image credit: AMD

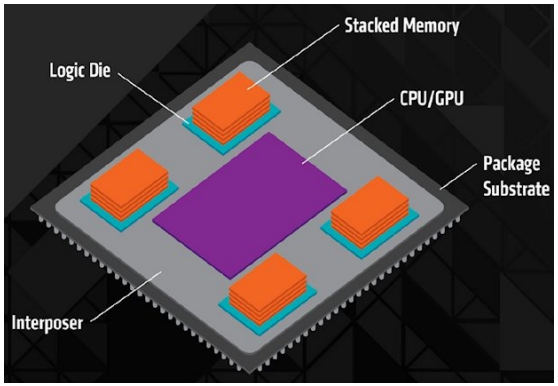
# HBM Advantages

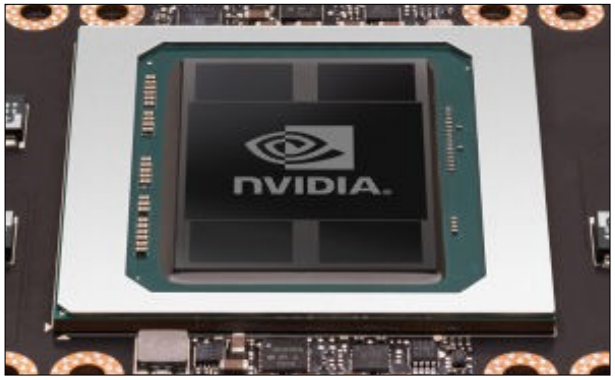
## More Bandwidth
## High Power Efficiency
## Small Form Factor

| | DDR4 | LPDDR4(X) | GDDR6 | HBM2 | HBM2E (JEDEC) | HBM3 (TBD) |
|---|---|---|---|---|---|---|
| Data rate | 3200Mbps | 3200Mbps (up to 4266 Mbps) | 14Gbps (up to 16Gb ps) | 2.4Gbps | 2.8Gbps | >3.2Gbps (TBD) |
| Pin count | x4/x8/x16 | x16/ch (2ch per die) | x16/x32 | x1024 | x1024 | x1024 |
| Bandwidth | 5.4GB/s | 12.8(17)GB/s | 56GB/s | 307GB/s | 358GB/s | >500GB/s |
| Density (per package) | 4Gb/8Gb | 8Gb/16Gb/2 4Gb/32Gb | 8Gb/16Gb | 4GB/8GB | 8GB/16GB | 8GB/16GB/ 24GB (TBD) |

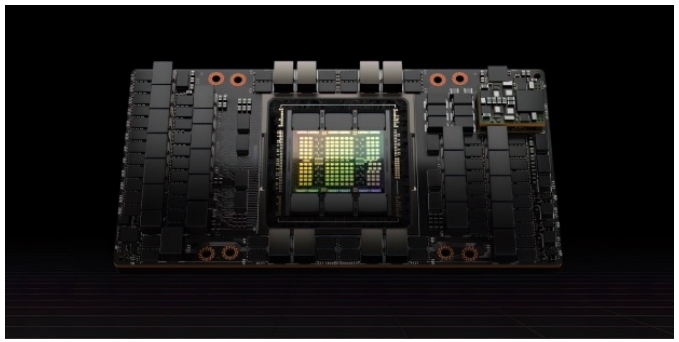
# GPUs are adopting HBM technologies




**NVIDIA P100 GPU (2016)**
**4096-bit interface: 4 HBM2 chips x 1024 bit interface per chip**
**720 GB/sec peak BW**
**4 x 4 GB = 16 GB capacity**



**NVIDIA H100 GPU (2022)**
**6144-bit interface: 6 HBM3 stacks x 1024 bit interface per stack**
**3.2 TB/sec peak BW**
**80 GB capacity**

# Nvidia HBM Roadmap

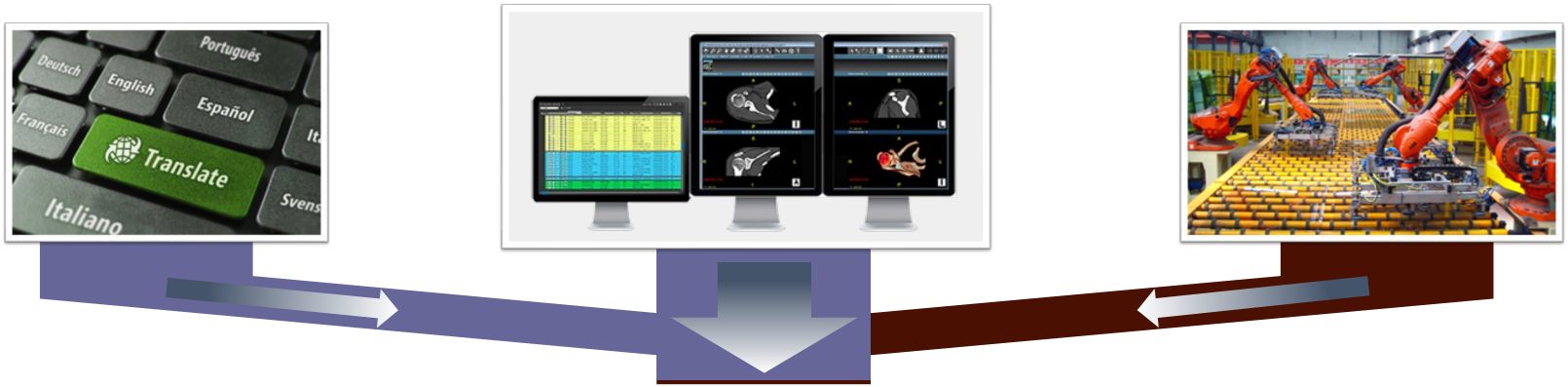
| | A100 80GB | H100 | H200 | GB200 NVL72 | GB300 NVL72 | VR200 NVL144 | VR300 NVL576 |
|---|---|---|---|---|---|---|---|
| HBM Type | 2E | 3 | 3E | 3E | 3E | 4-24Gb | 4E |
| Gb per Layer (Gb)(1) | 16 | 16 | 24 | 24 | 24 | 24 | 32 |
| Layers per Stack (#) | 8 | 8 | 8 | 8 | 12 | 12 | 16 |
| GB per Stack (GB) | 16 | 16 | 24 | 24 | 36 | 36 | 64 |
| HBM Stacks (#) | 5 | 5 | 6 | 8 | 8 | 8 | 16 |
| **Total Capacity (GB)** | **80** | **80** | **144** | **192** | **288** | **288** | **1,024** |
| **Increase vs A100** | **1.0x** | **1.0x** | **1.8x** | **2.4x** | **3.6x** | **3.6x** | **12.8x** |
| **Memory BW (TB/s)** | **2.0** | **3.4** | **4.8** | **8.0** | **8.0** | **13.0** | **32.0** |
| **Increase vs A100** | **1.0x** | **1.6x** | **2.4x** | **3.9x** | **3.9x** | **6.4x** | **15.7x** |

*Note: (1) 8Gb = 1GB.*

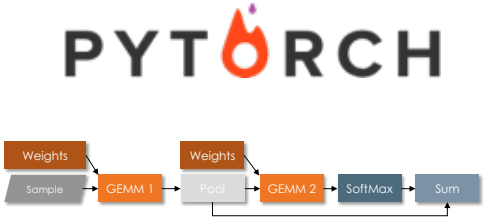
# Can we have asynchrony with a simpler programming model?

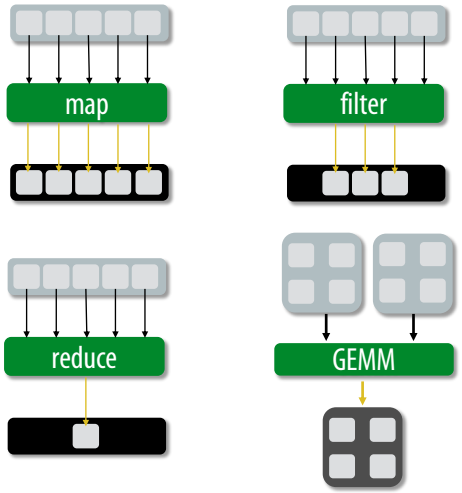
**(Hint: Take a data-centric view)**

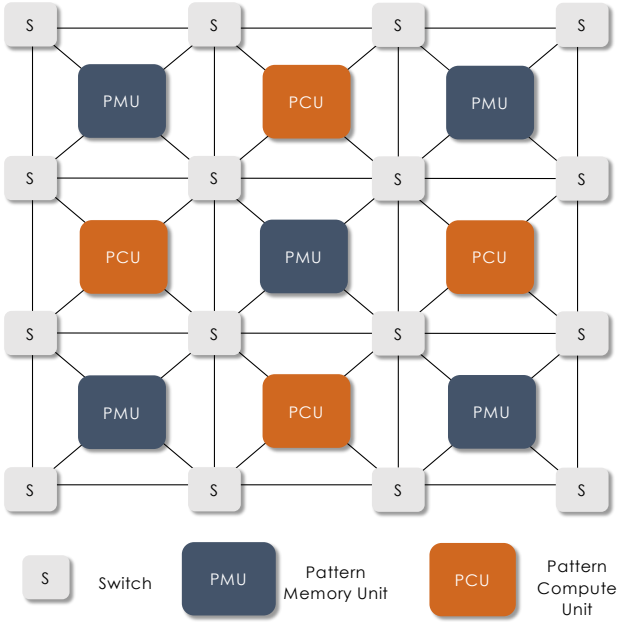
# Recall: AI Models are Dataflow Graphs



# AI Models ⇒ Dataflow Architecture



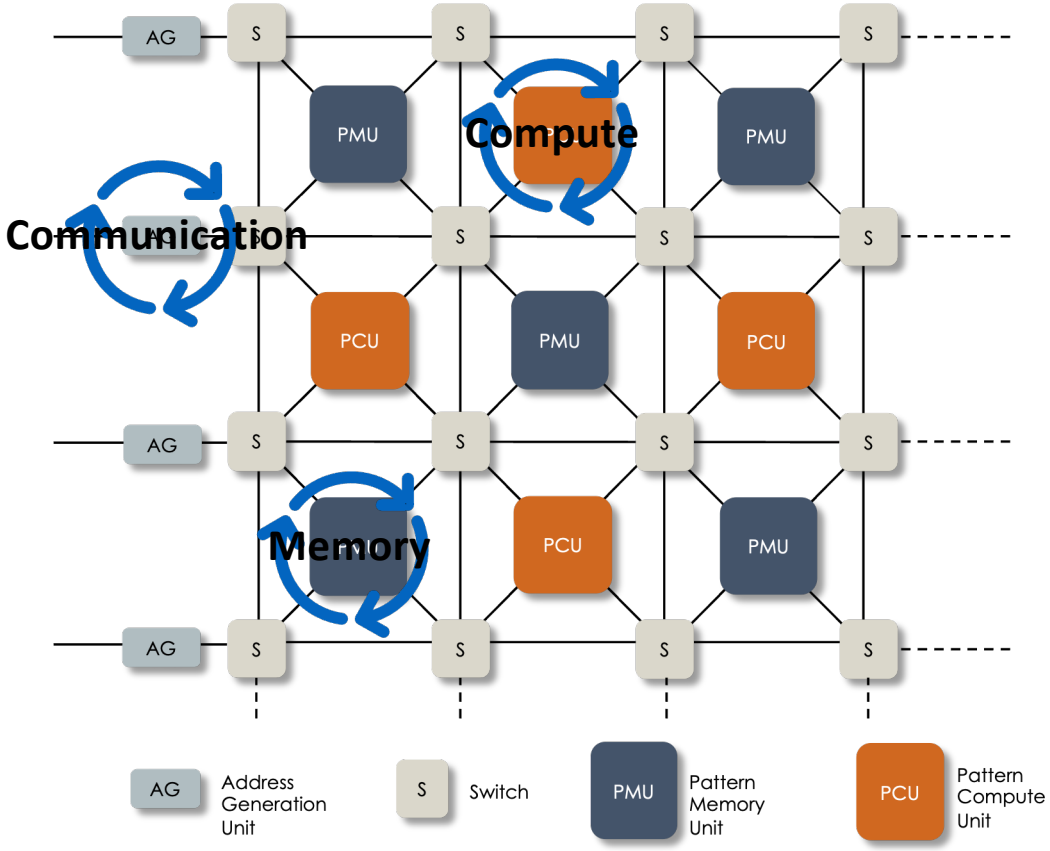

AI Models

**Dataflow graph:**
**GEMM + Parallel Patterns**

**Plasticine**
**Reconfigurable Dataflow Architecture**

Prabhakar, Zhang, et. al. ISCA 2017

# Reconfigurable Dataflow Architecture vs Ideal Accelerator

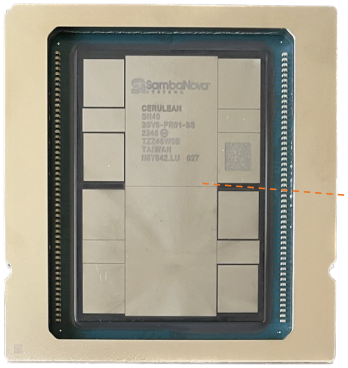



| Feature | Why? |
|---|---|
| Tiled tensors (e.g. 16 x 16, 32 x 32) | Max TFLOPS on GEMM<br>Low instr. overhead |
| Asynchronous compute | Overlap compute and memory access |
| Asynchronous memory access | Overlap compute and memory access |
| Asynchronous chip-to-chip communication | Overlap compute,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

**No instructions ⇒ No instruction fetch/decode overhead**

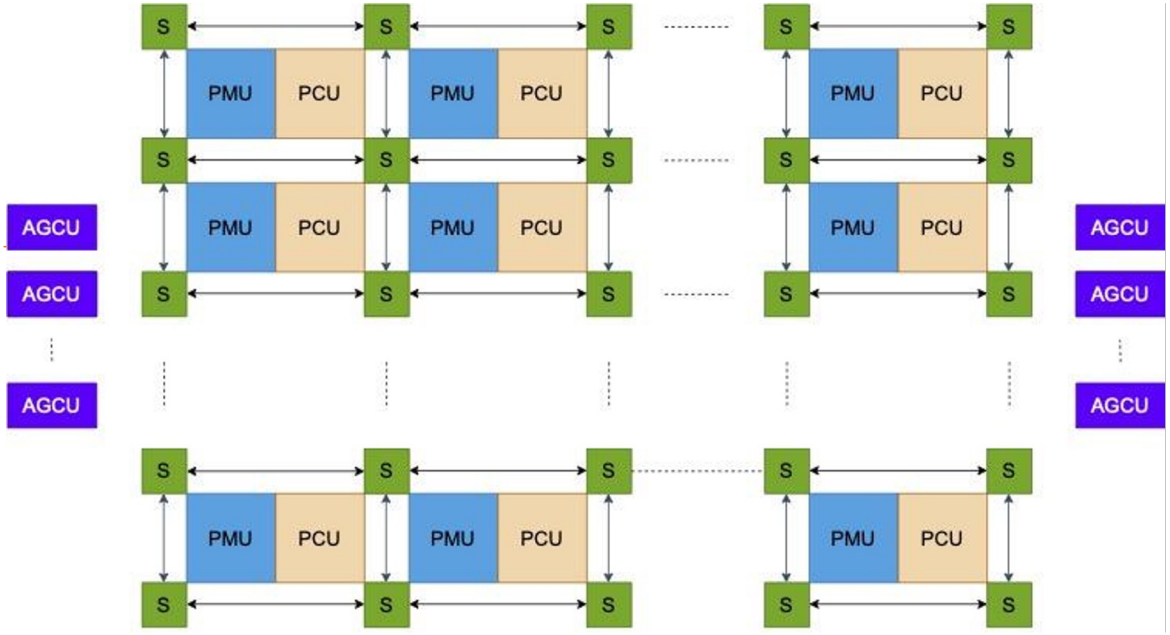
**Extreme asynchrony: no sequential instruction execution**

# Reconfigurable Dataflow



## SambaNova SN40L RDU

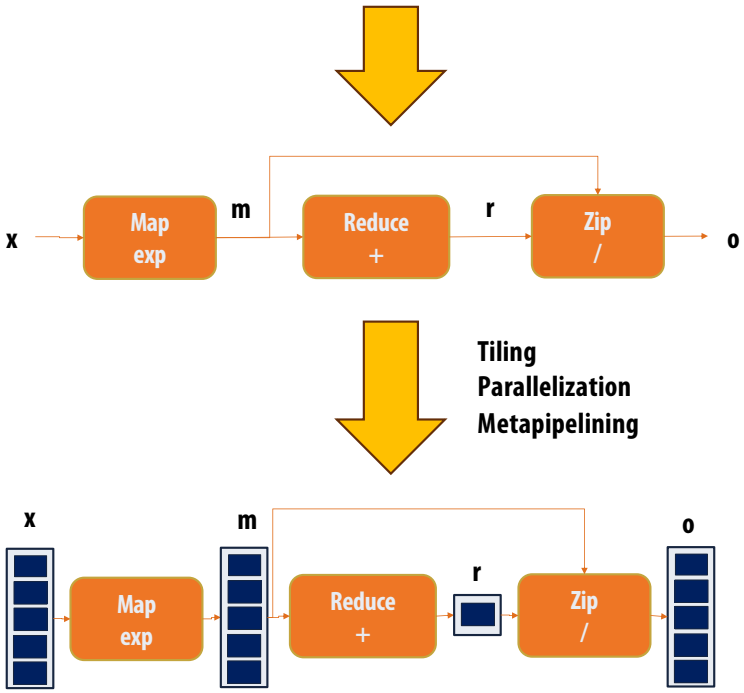
- 1,040 PCUs and PMUs
- 638 TFLOPS (bf16)
- 520 MB on-chip SRAM
- 64 GB HBM
- 1.5 TB DDR

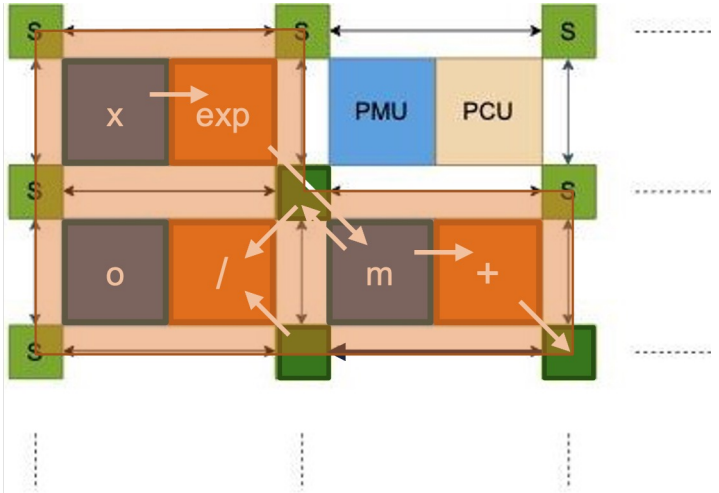



- PCU: Pattern Compute Unit
  - systolic and SIMD compute (16 x 8 bf16)
- PMU: Pattern Memory Unit
  - High address generation flexibility and bandwidth (0.5 MB)
- S: Mesh switches
  - High on-chip interconnect flexibility and bandwidth
- AGCU: Address Generator and Coalescing Unit
  - Portal to off-chip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

# Dataflow Programming with Data Parallel Patterns

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

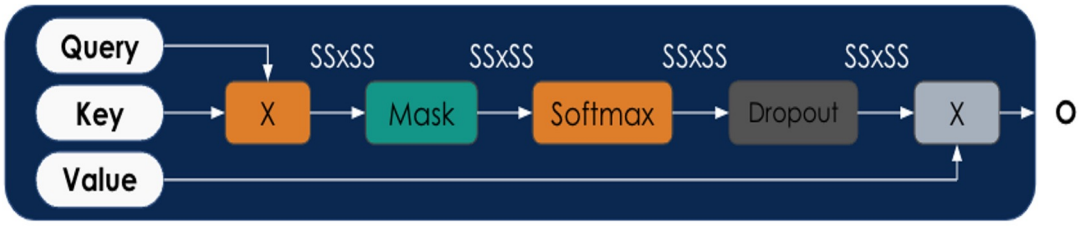


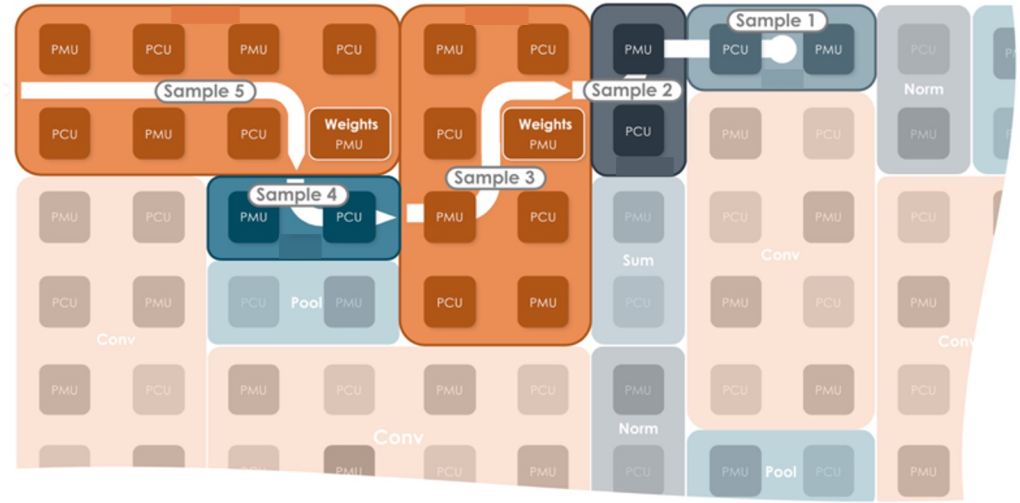

Place & Route
Codegen

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

# Streaming Dataflow ⇒ Kernel Fusion




Attention Algorithm




Attention Algorithm on RDA

Coarse -grained pipelining

# 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**

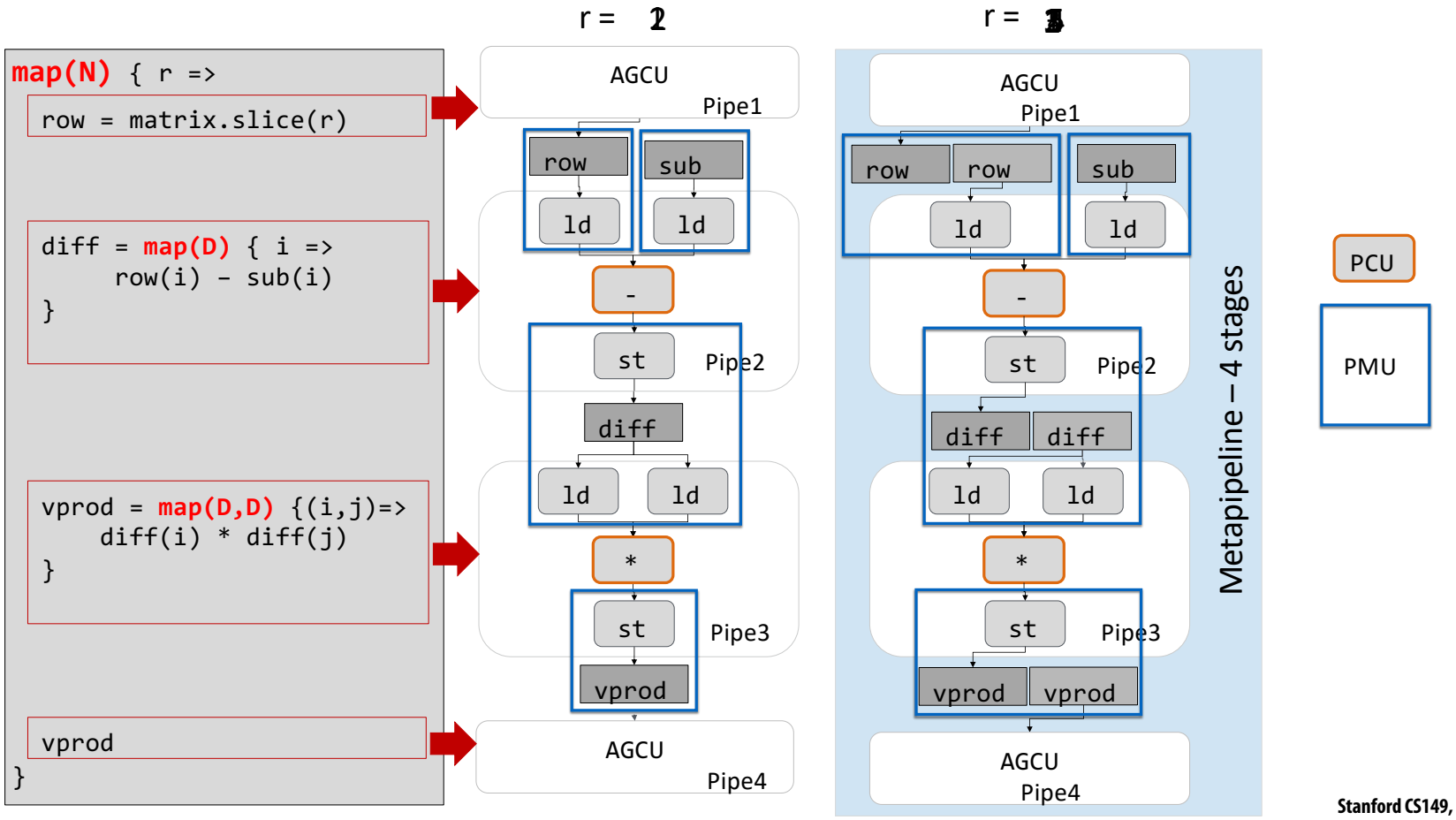
- **Handles imbalanced stages with varying execution times**

**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)

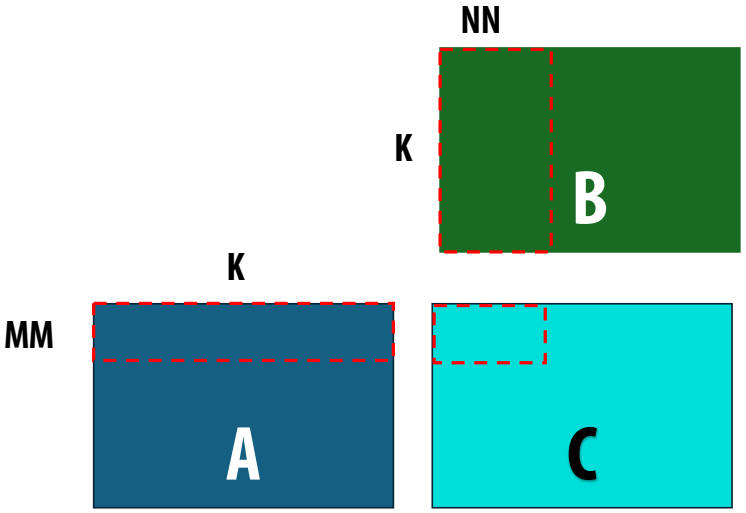


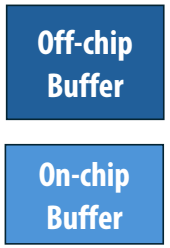
# Matmul Metapipeline

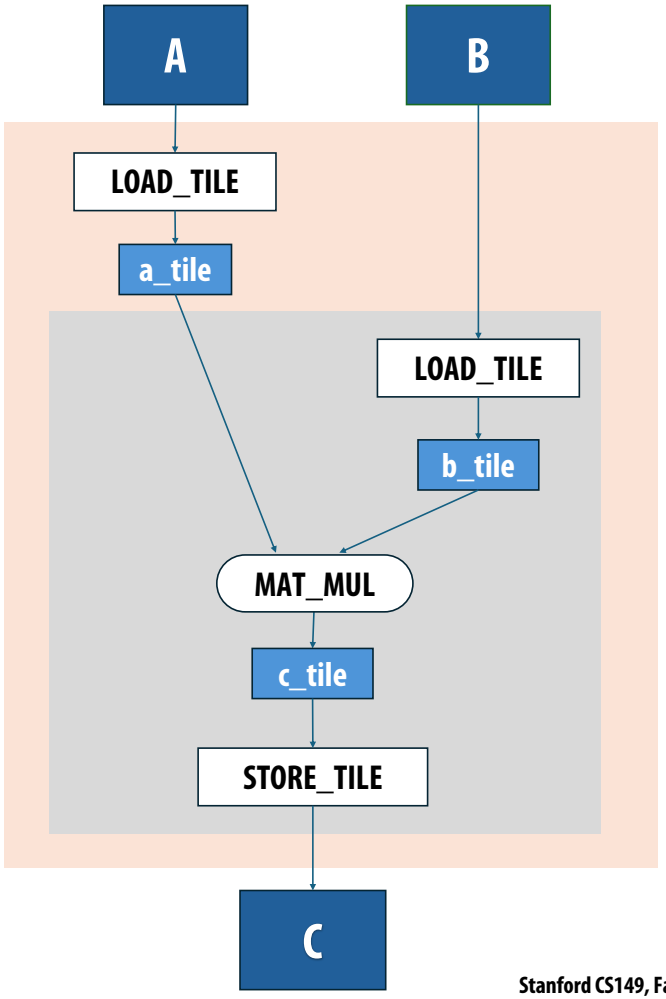
```
auto format = DataFormat::kBF16;

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)
    }
}
```

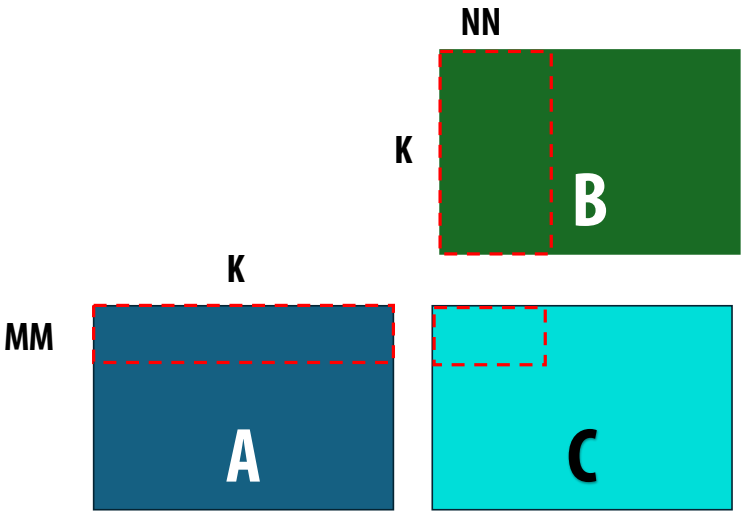


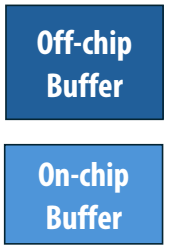



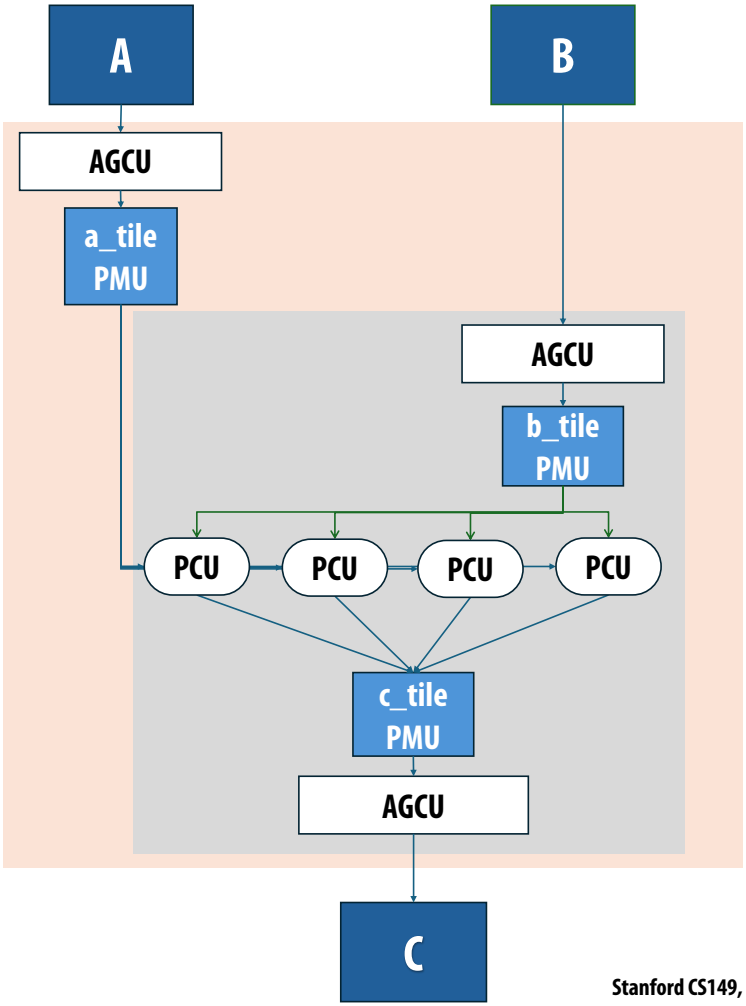



# Matmul Metapipe Mapping

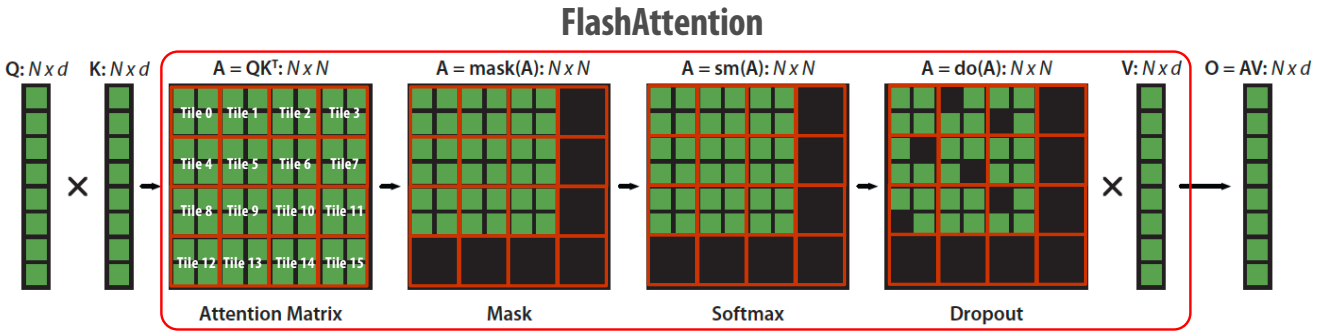
```
METAPIPE(M, MM) {
    a_tile = LOAD_TILE(A, a_tile_shape)
    METAPIPE(N, NN) {
        b_tile = LOAD_TILE(B, b_tile_shape)
        c = MAT_MUL(a_tile, b_tile, row_par = 4)
        c_tile = BUFFER(c)
        STORE_TILE(C,c_tile)
    }
}
```

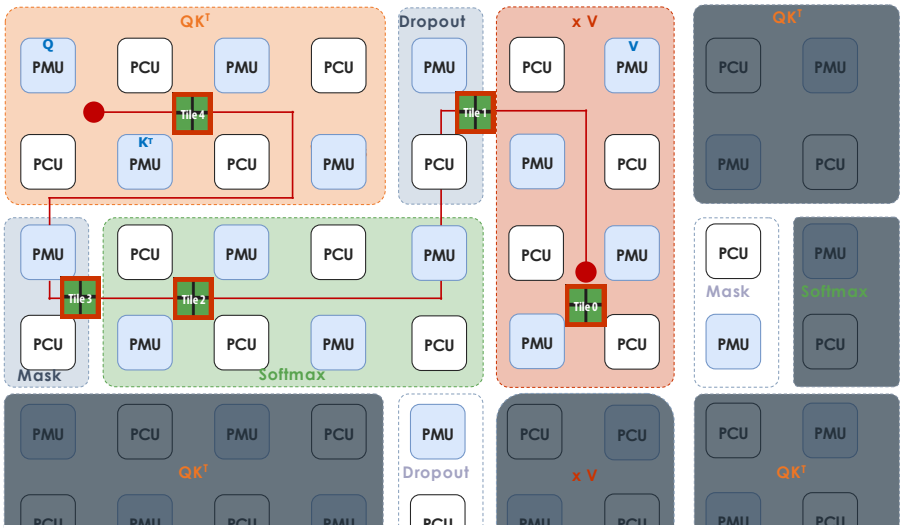


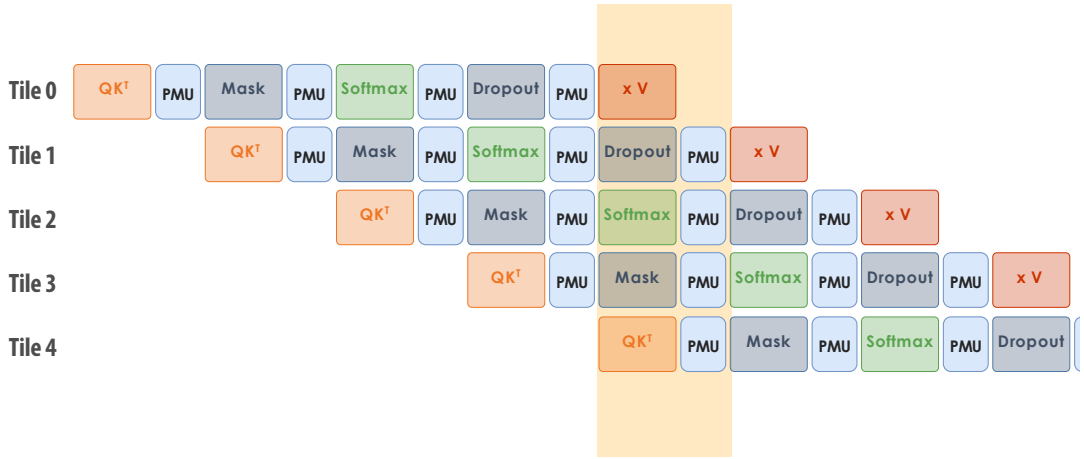



# FlashAttention Metapipeline



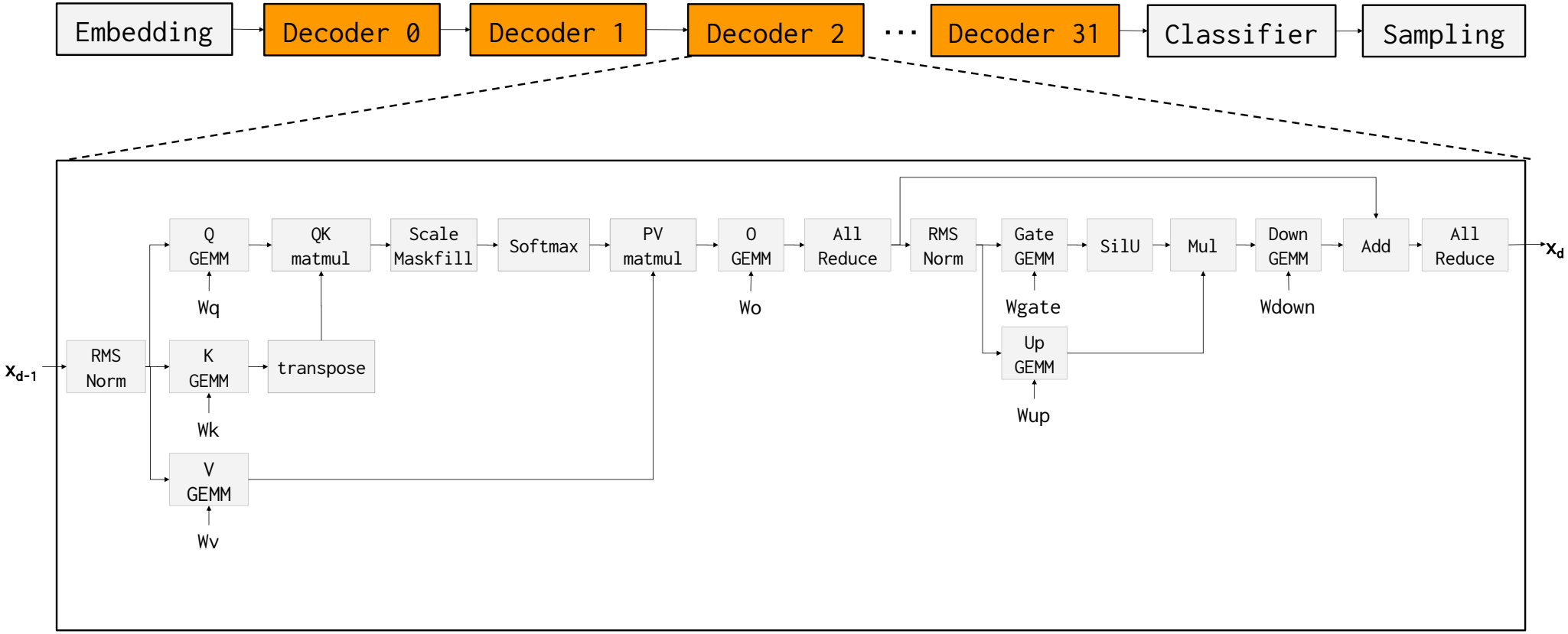




Dataflow execution with token control ⇒ no lock-based synchronization



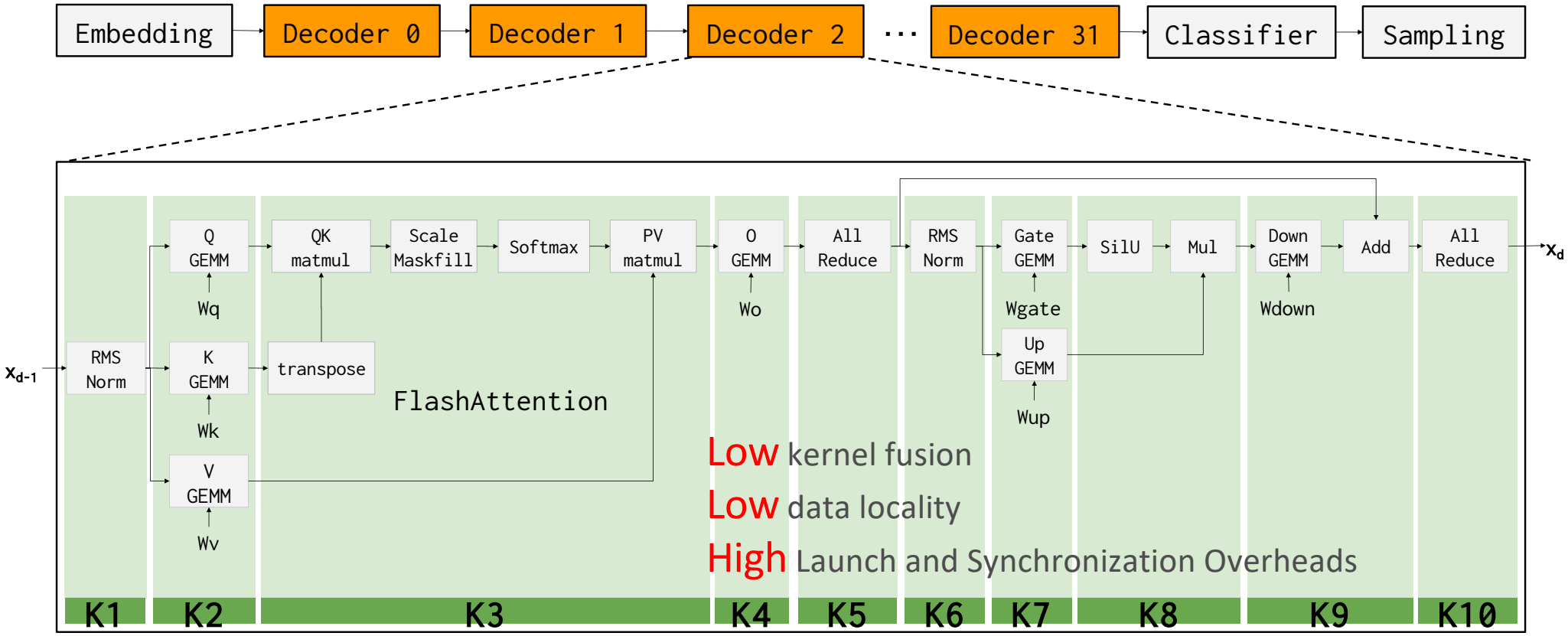

MetaPipeline = Streaming Dataflow

# 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
$x_d$

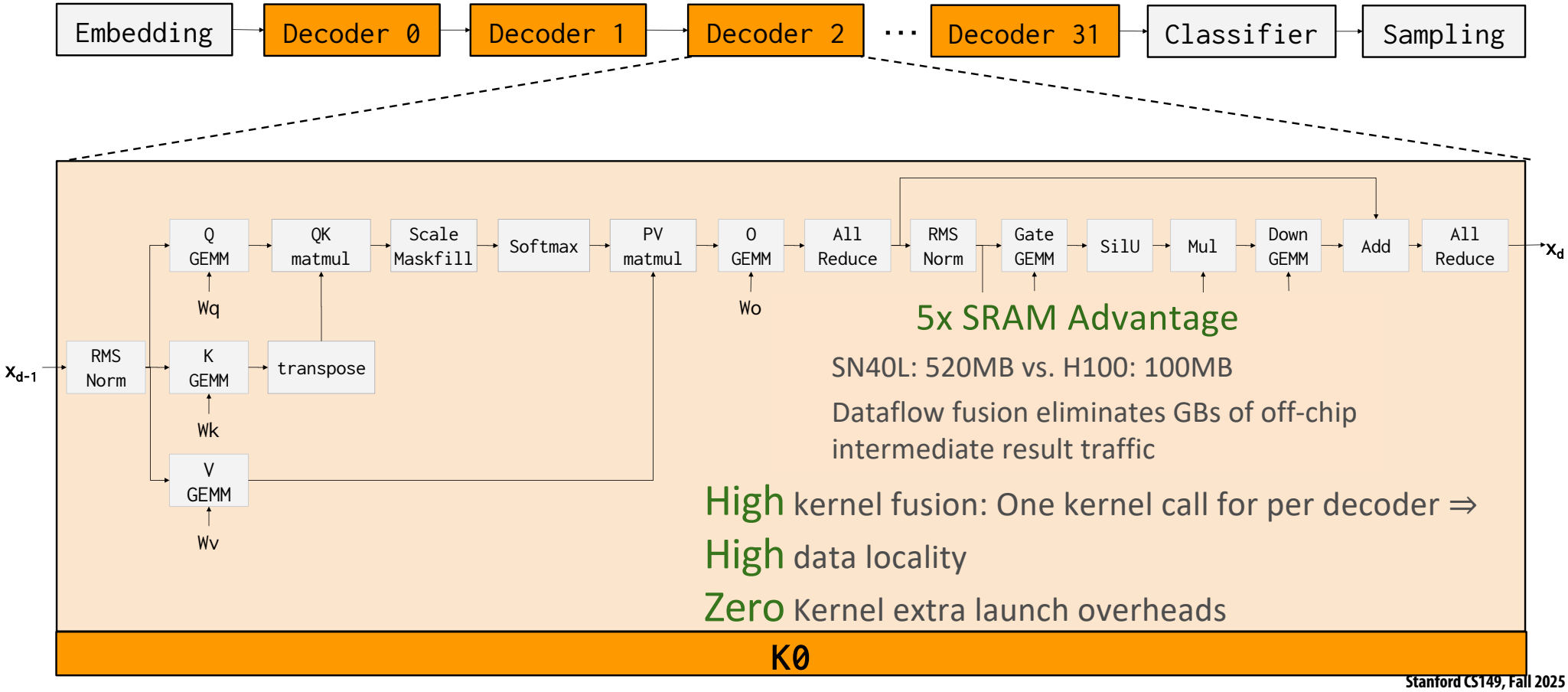
# Limited Kernel Fusion on GPUs

**Llama3.1 8B with Tensor-RT LLM**



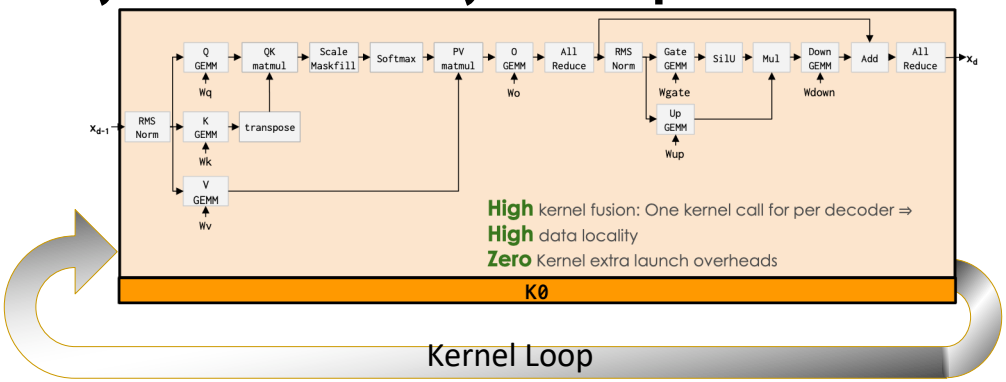
# RDU Fuses Entire Decoder into One Kernel !

**Llama3.1 8B with aggressive kernel fusion**



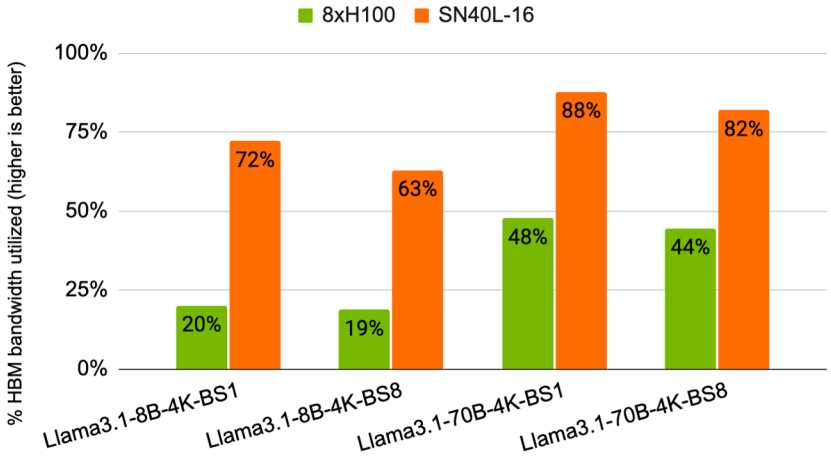
# Kernel Loop

## Asynchronous memory and compute




## One kernel call for all decoders

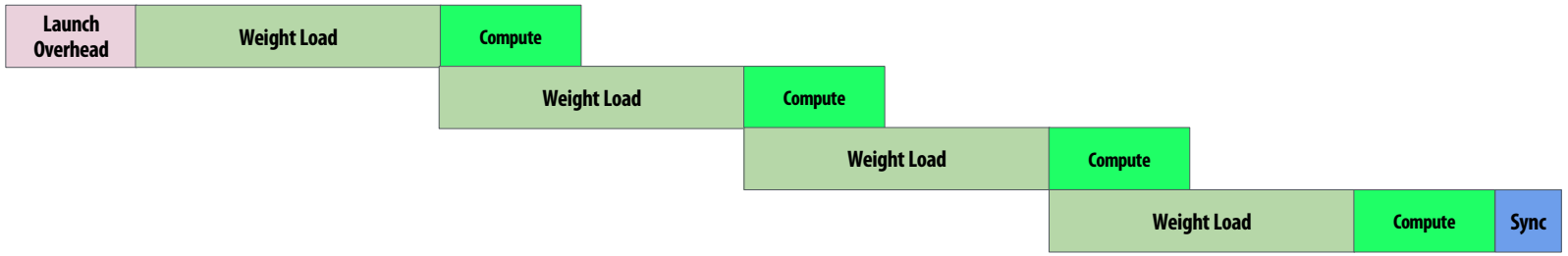
- 3 calls per token on RDU
- ~800 calls per token on GPU
- 100x fewer kernel calls



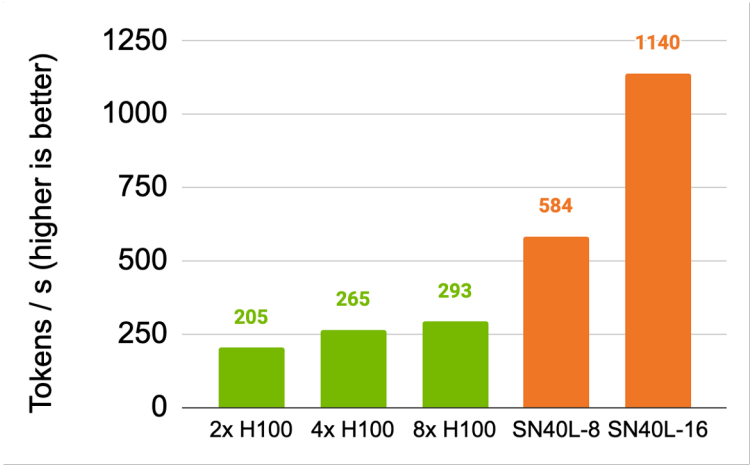

**1 Decoder**


**Kernel Loop**

**4 Decoders**

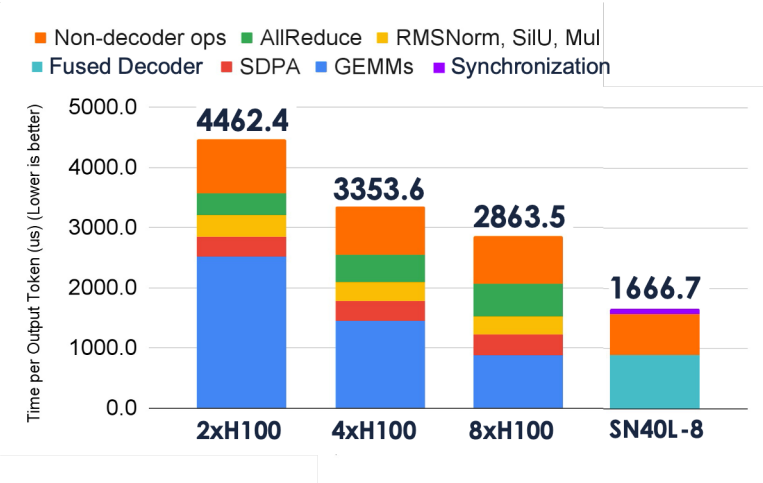


# Dataflow ⇒ High Performance




**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

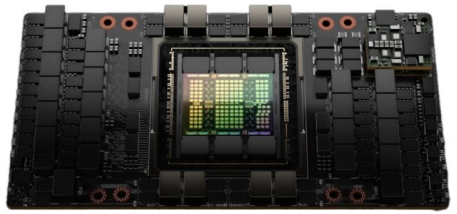
H100: Asynchronous compute and memory mechanisms ⇒ complex programming

- 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

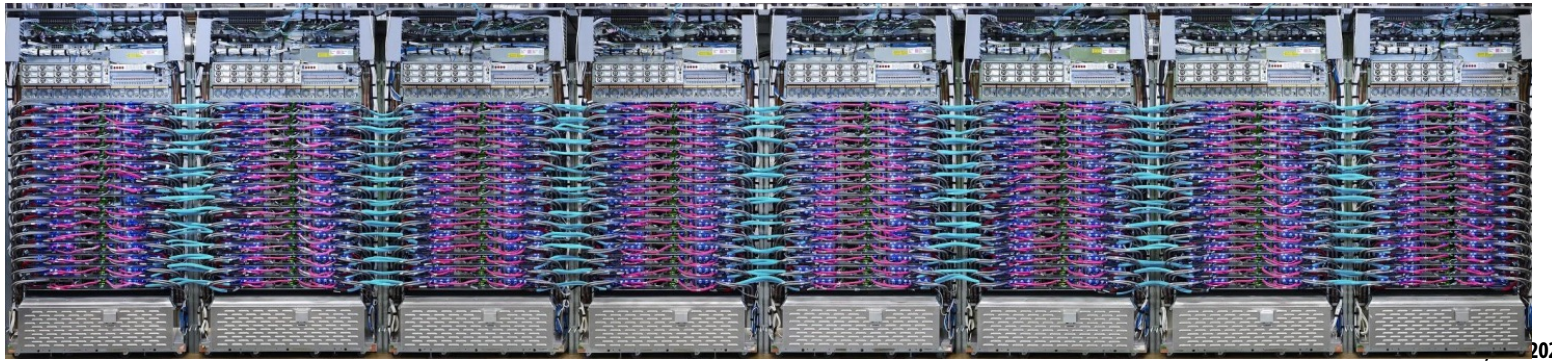
Minimizing synchronization overheads required for high performance



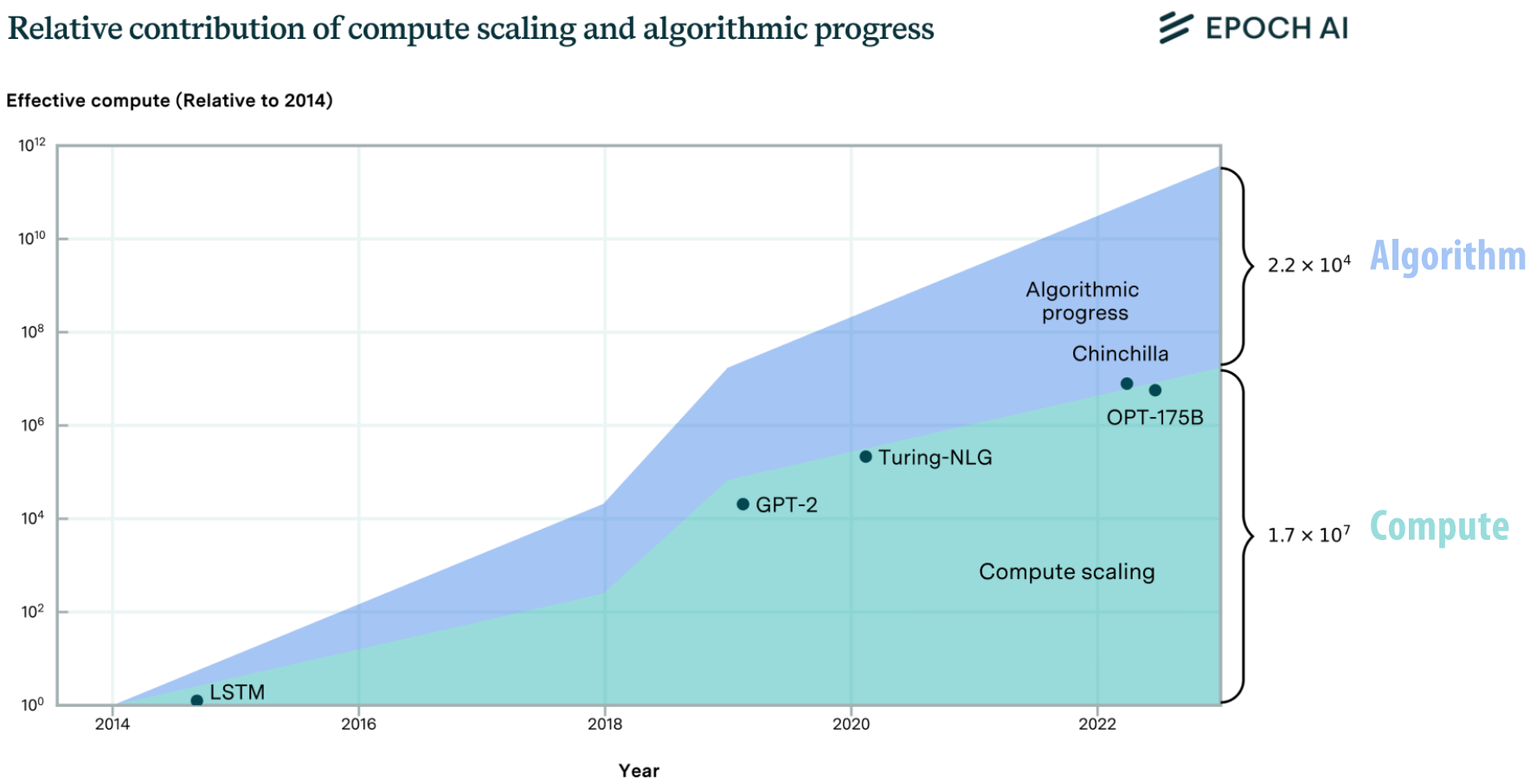
H100

SN40L

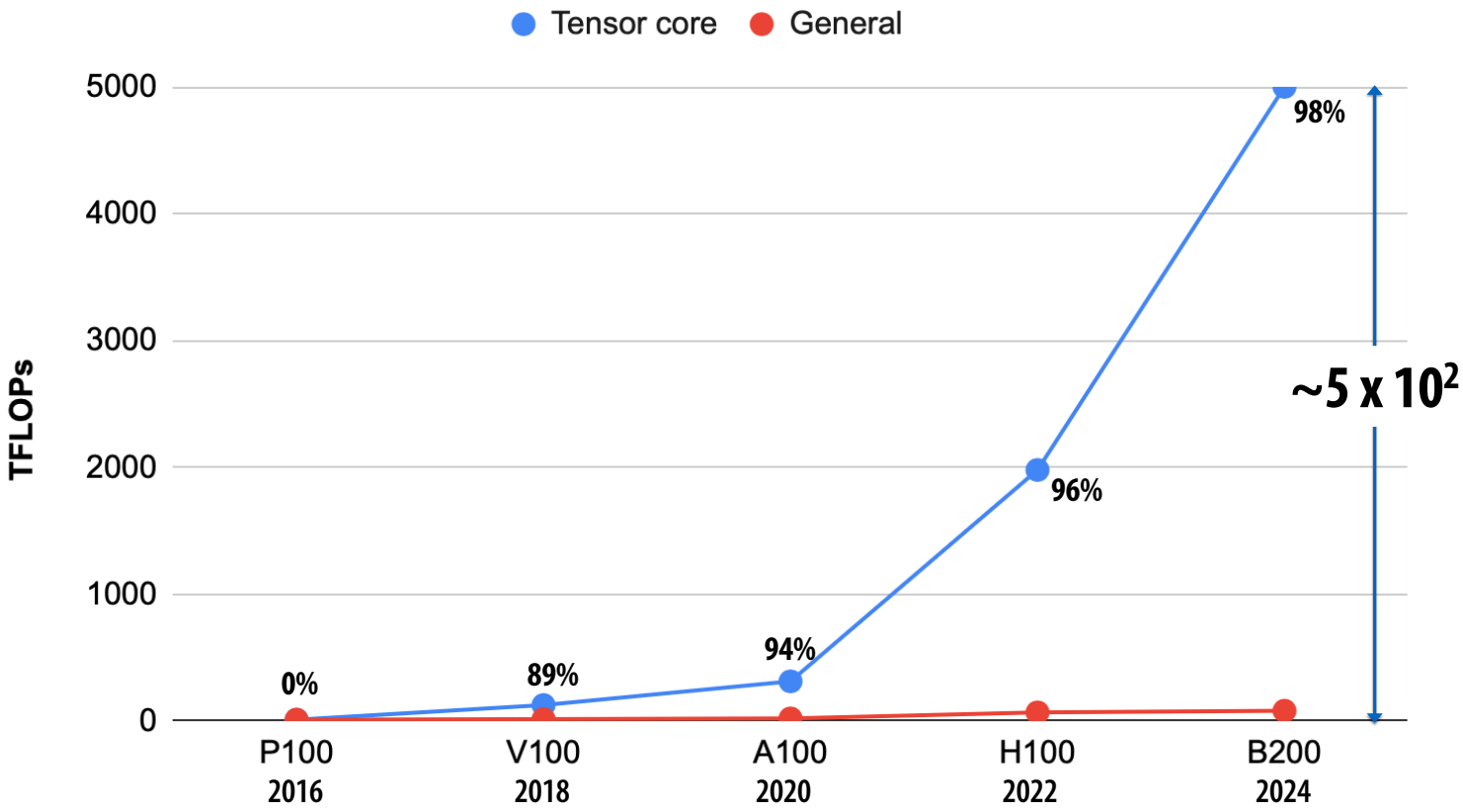
TPU supercomputer (1024 TPU v3 chips)



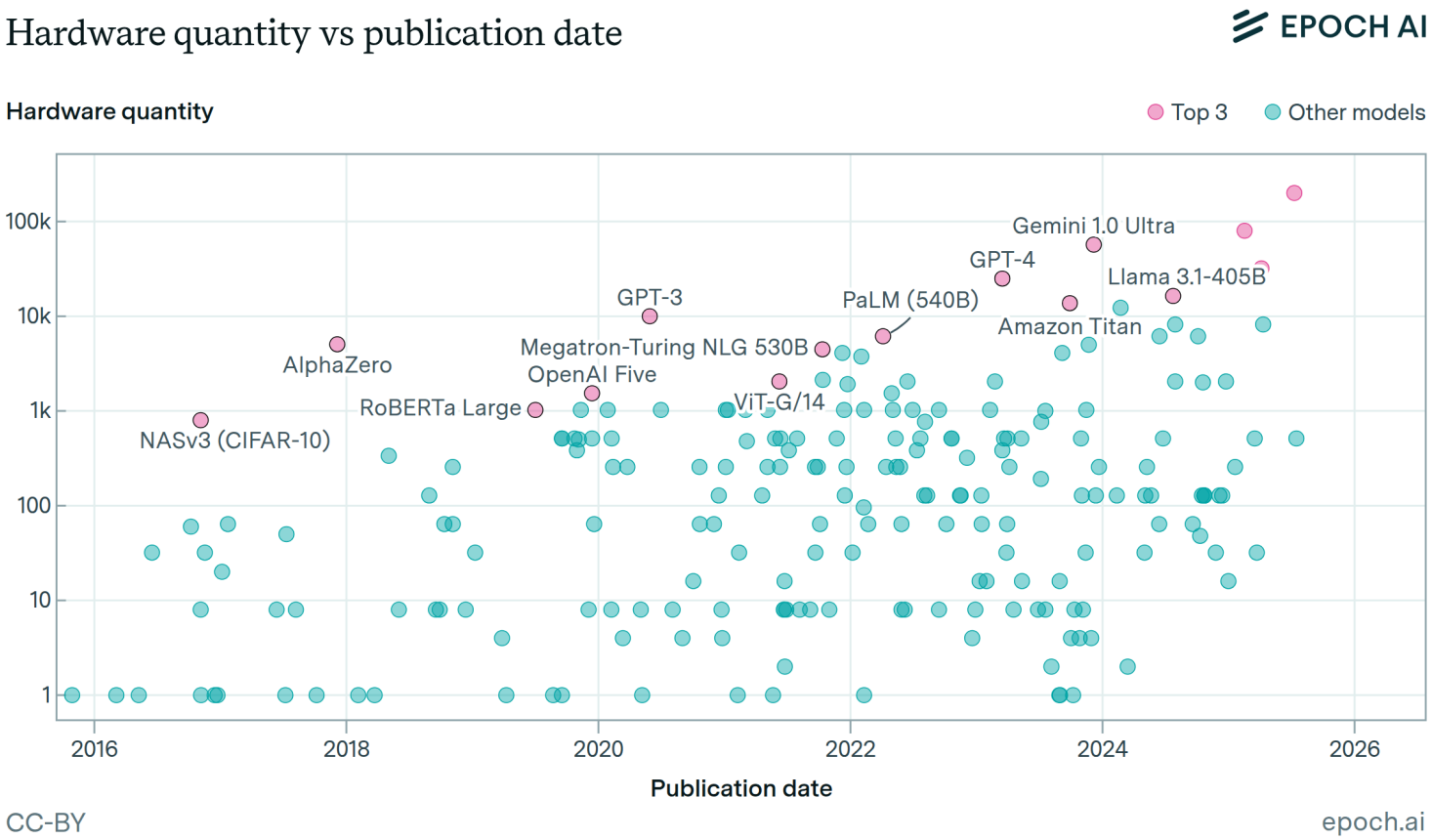
# AI Progress Relies on Hardware Improvement



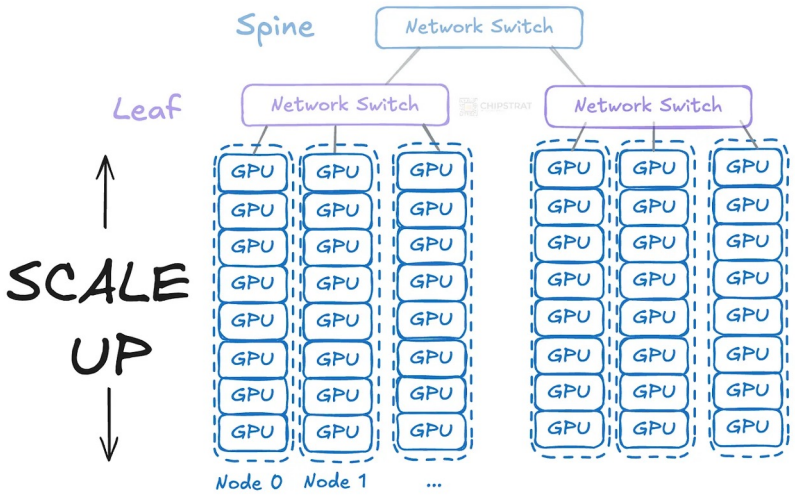
# All the TFLOPS are in the Tensor Cores

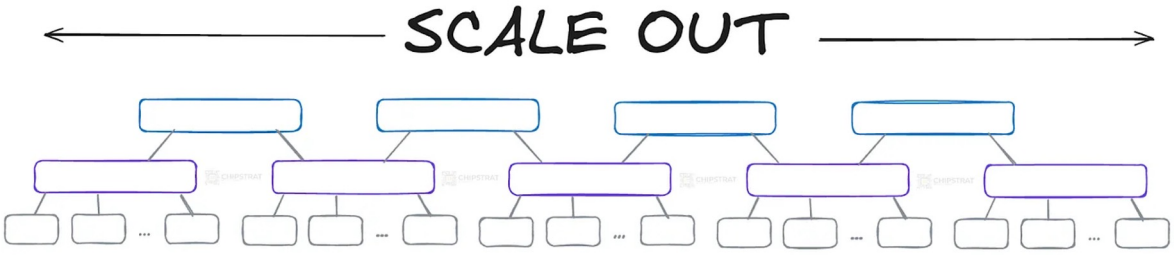


# AI Cluster Size



# Scale Up and Scale Out




Both figures from https://creativestrategies.com/gpu-networking-basics/

# DGX SUPERPOD

## Modular Architecture

### 1K GPU SuperPOD Cluster

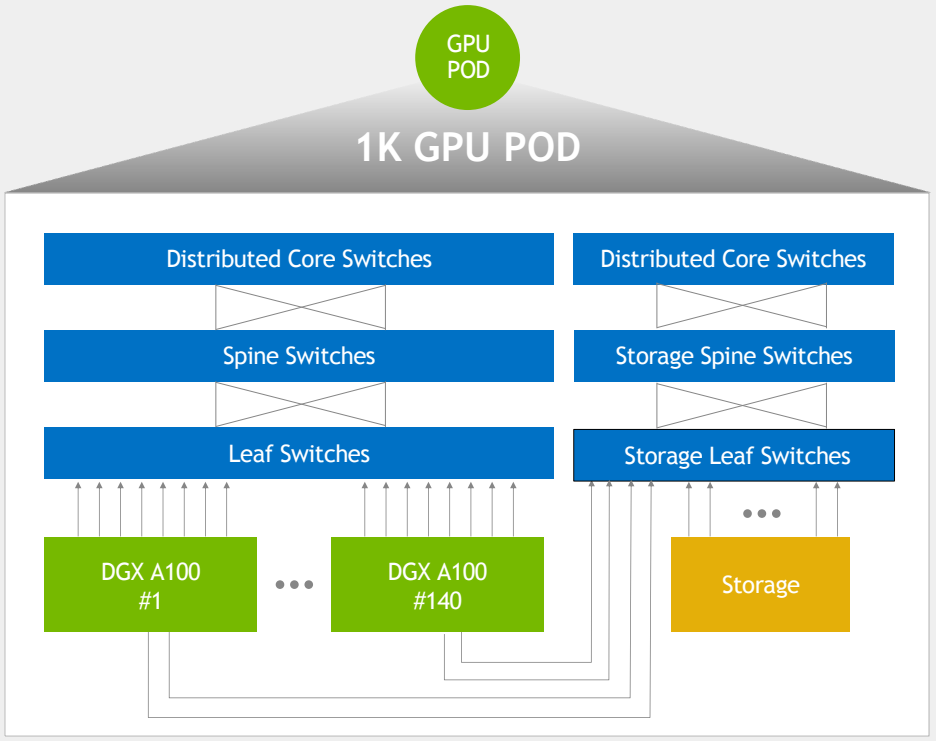
- 140 DGX A100 nodes (1,120 GPUs) in a GPU POD
- 1st tier fast storage - DDN AI400x with Lustre
- Mellanox HDR 200Gb/s InfiniBand - Full Fat-tree
- Network optimized for AI and HPC

### DGX A100 Nodes

- 2x AMD 7742 EPYC CPUs + 8x A100 GPUs
- NVLINK 3.0 Fully Connected Switch
- 8 Compute + 2 Storage HDR IB Ports

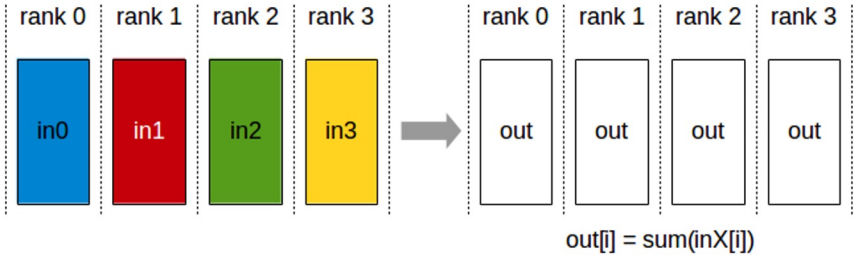
### A Fast Interconnect

- Modular IB Fat-tree
- Separate network for Compute vs Storage
- Adaptive routing and SharpV2 support for offload



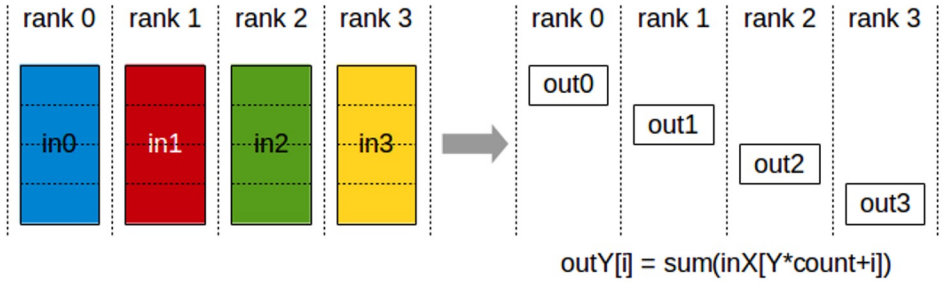
# Message Passing Communication Primitives: AllReduce, ReduceScatter, AllGather

rank = accelerator node



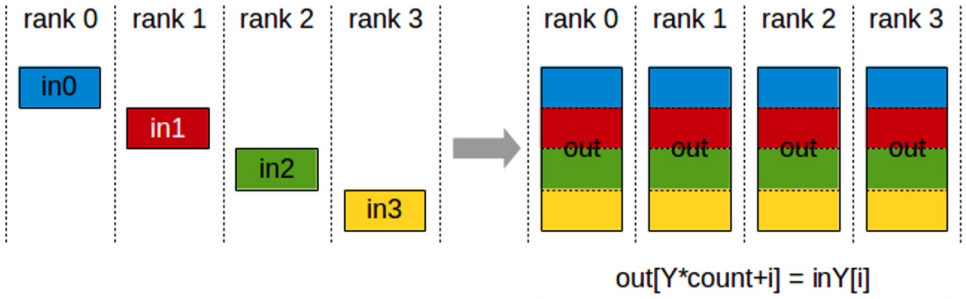

Allreduce

=




ReduceScatter

+




AllGather

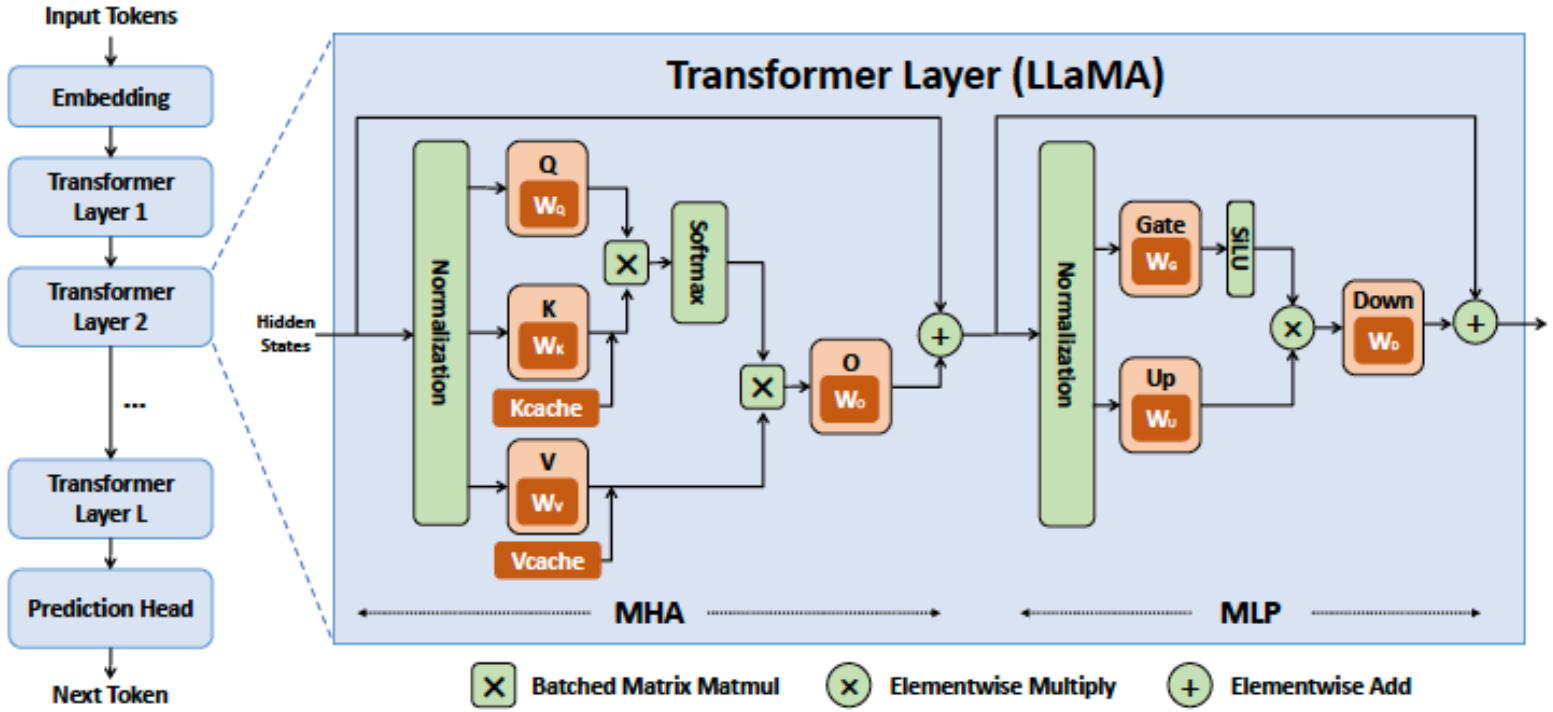
# Message Passing Communication Primitives: All-to-All

| | | | | |
|---|---|---|---|---|
| rank 0 | A0 | A1 | A2 | A3 |
| rank 1 | B0 | B1 | B2 | B3 |
| rank 2 | C0 | C1 | C2 | C3 |
| rank 3 | D0 | D1 | D2 | D3 |

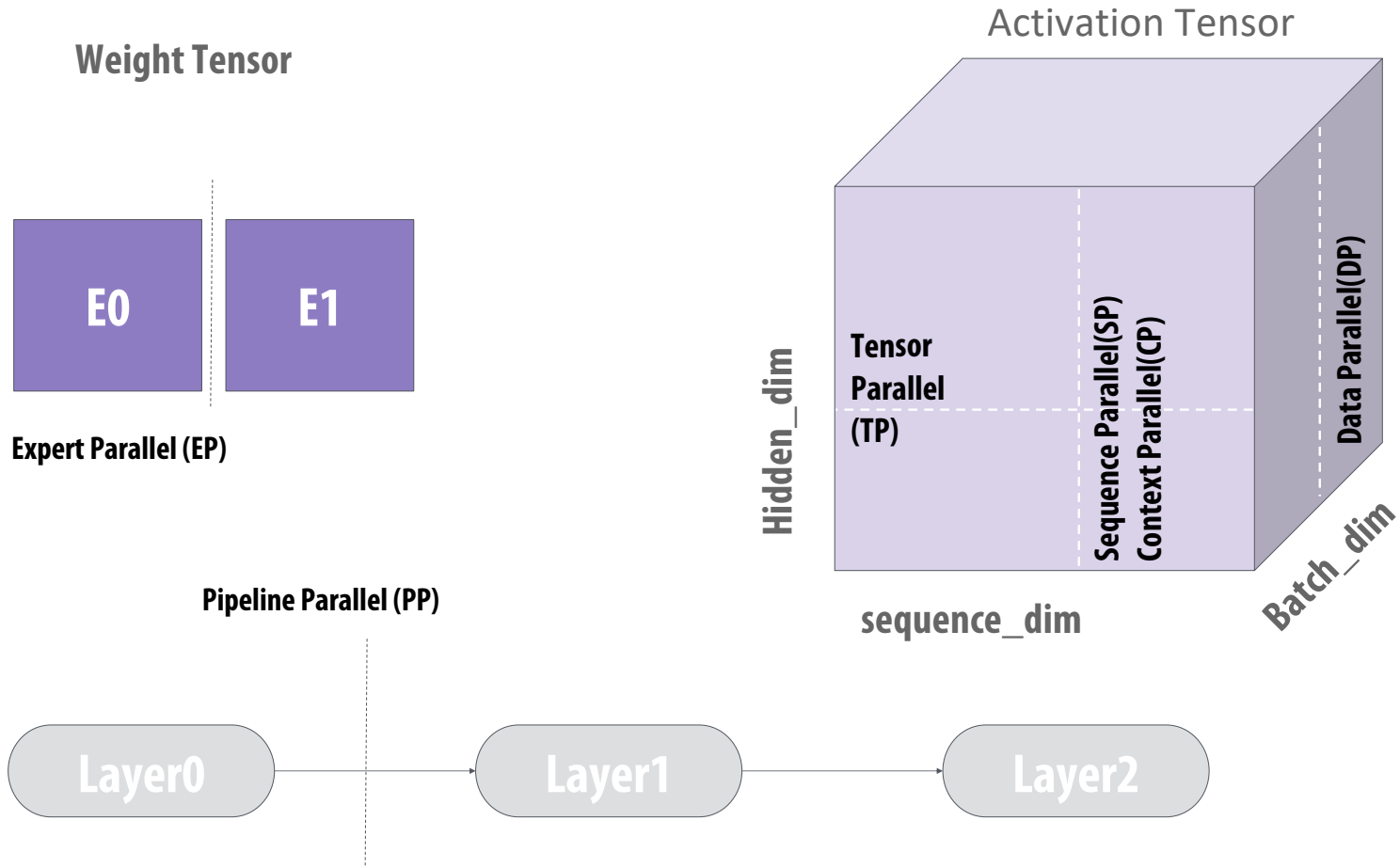
All-to-All

| | | | |
|---|---|---|---|
| A0 | B0 | C0 | D0 |
| A1 | B1 | C1 | D1 |
| A2 | B2 | C2 | D2 |
| A3 | B3 | C3 | D3 |

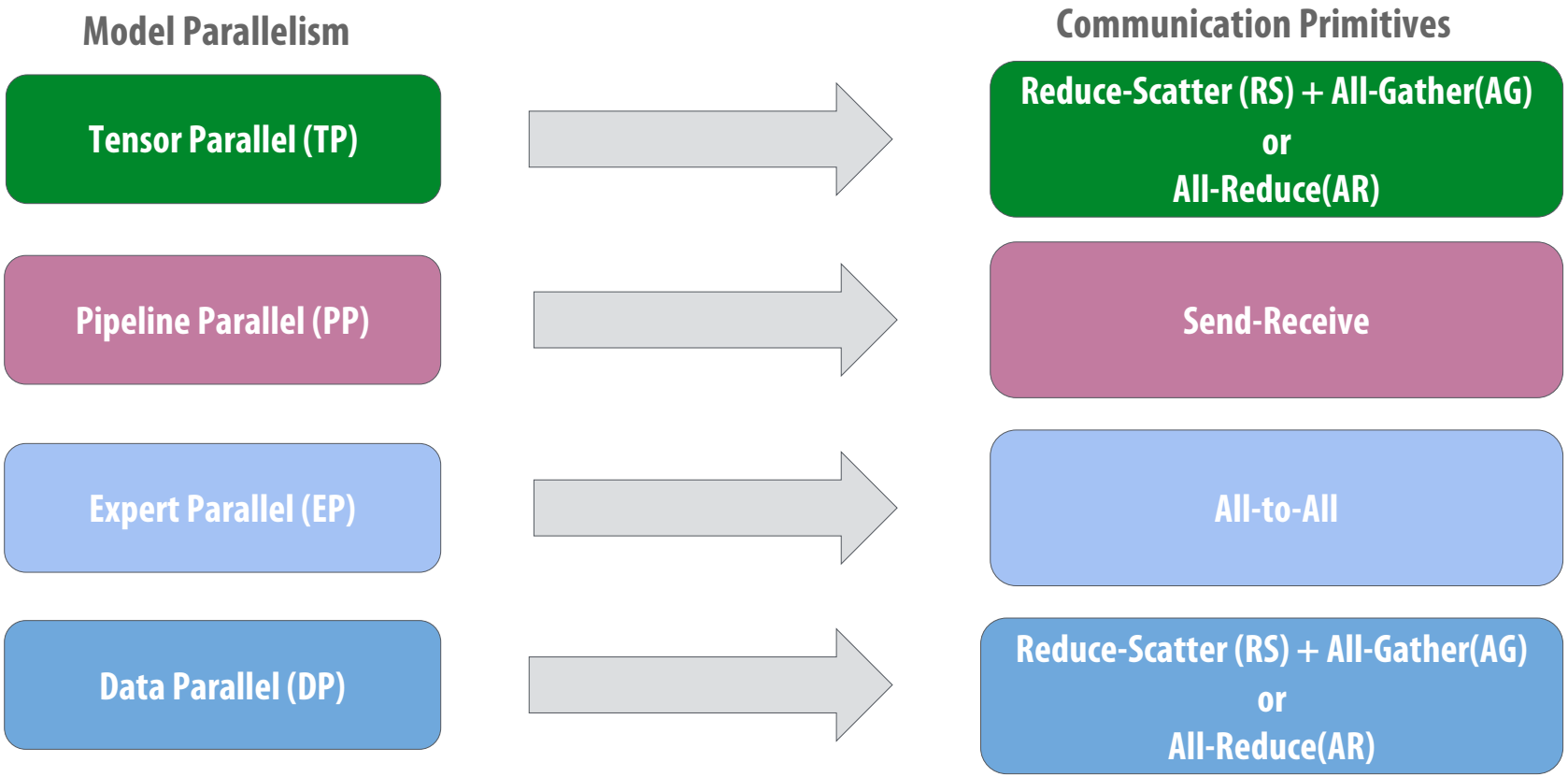
# Transformer



# Where is the Parallelism in AI Models?

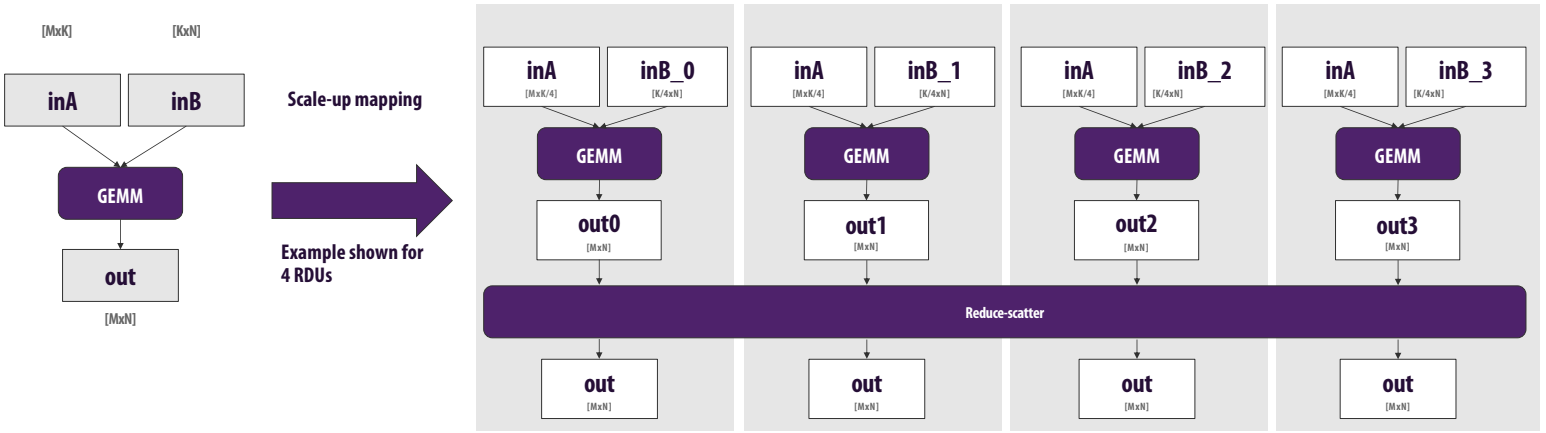


# Parallelism and Communication

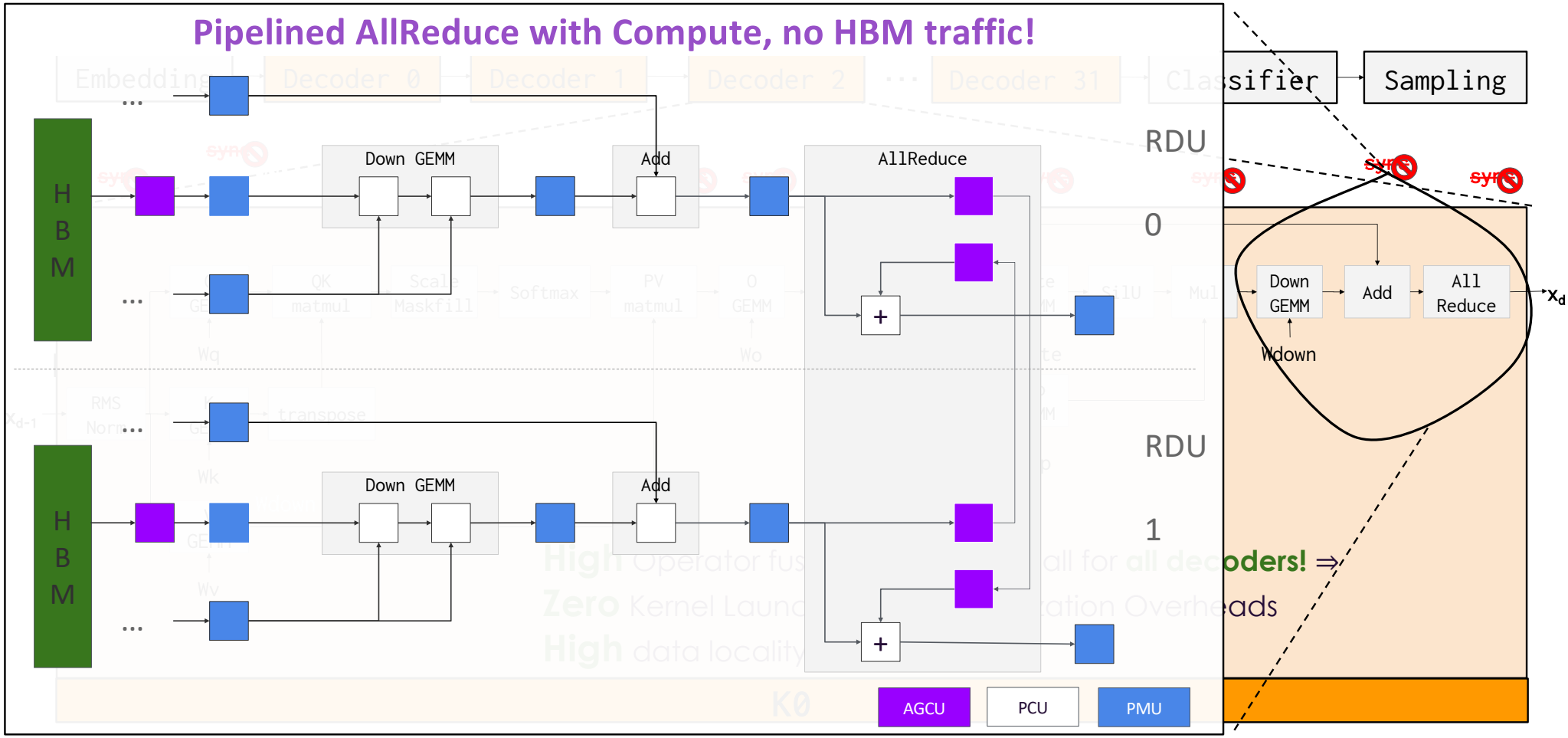


# Distributed Matrix-Multiply Example

- inputA[MxK] * inputB[KxN] = out[MxN]
- BS = 16, M = 24576, K = 131072, N = 8192
- Mapping: Distribute K dimension across S RDUs
  - Matrix multiply size per socket: [MxK/S] * [K/SxN] = [MxN]
  - Produces S partial results of size [MxN], one per socket
  - S-way reduce-scatter to combine the partial results



# Compute - Communication Overlap



# Importance of Overlap - Conceptual


| | Without Overlap (GPU) | With Overlap (RDU) |
|---|---|---|
| 8 sockets | | |
| 16 sockets | | |
| 32 sockets | | |

Communication time increases on GPUs with more sockets
Communication becomes the bottleneck without overlap
GPUs need need large interconnect bandwidth to get high utilization

# Importance of Overlap - Quantified on RDUs

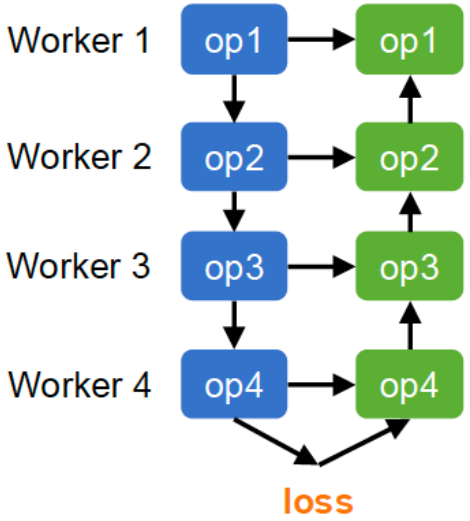
| Benchmark Tensor Dimensions | BS = 16, M = 24576, K = 131074, N = 8192 | | |
|---|---|---|---|
| Total Benchmark TFLOPs | 844.44 | | |
| Number RDUs | 8 | 16 | 32 |
| Total System TFLOPs | 12744 | 25488 | 50976 |
| Compute roofline time @100% utilization (ms) | 66.3 | 33.1 | 16.5 |
| Reduce-scatter time @100% link utilization (ms) | 8.6 | 9.7 | 15 |
| Theoretical Peak utilization without overlap | 88.5% | 77% | 52% |
| Measured Utilizations with overlap | 72% | 75% | 79% |

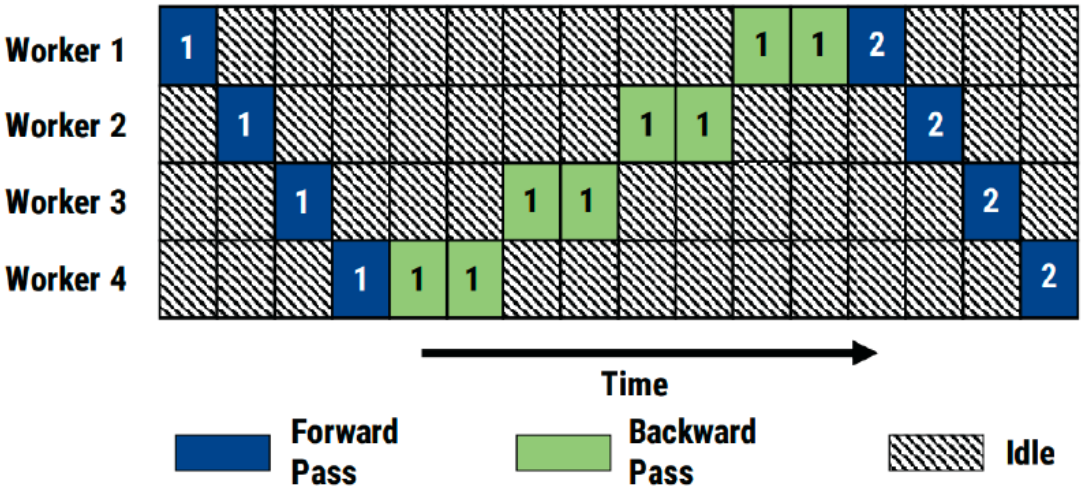
Sustained 70+% utilization across 32 sockets due to compute-communication overlap

# Pipeline Parallelism and Training

## Under-utiliization of compute resources

## Low overall throughput



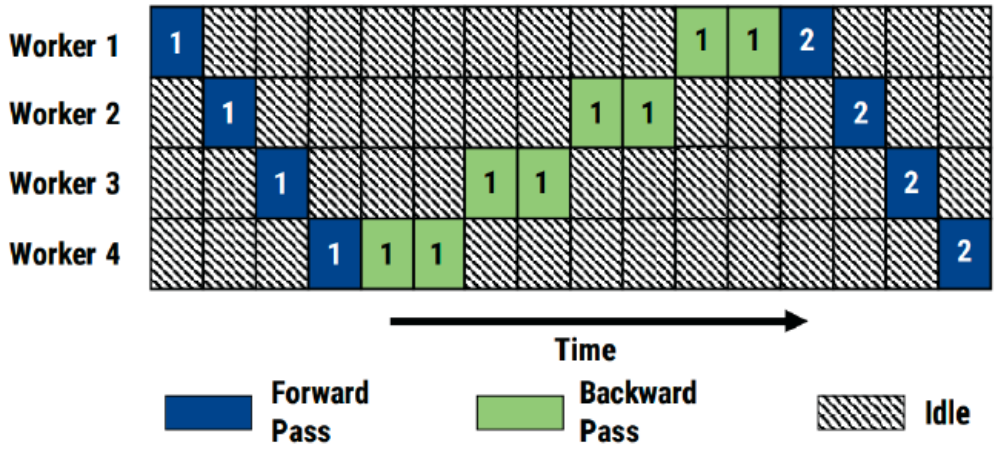



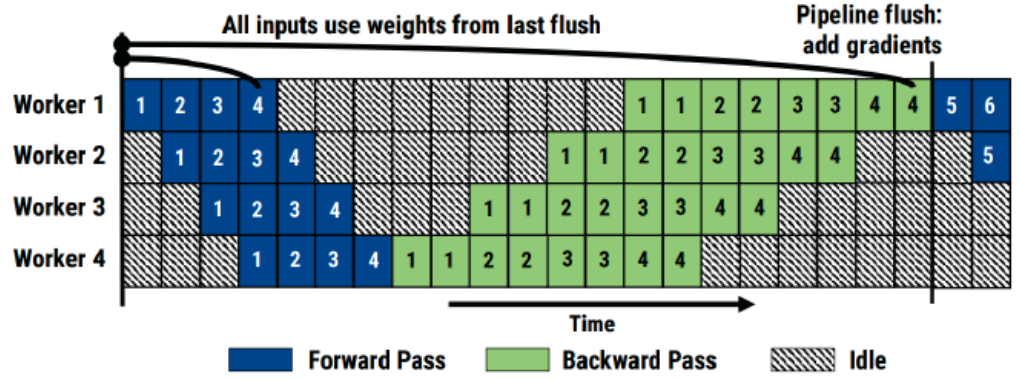
# Fine-grained Pipeline Parallelism

**Mini-batch: the number of samples processed in each iteration**

**Divide a mini-batch into multiple micro-batches**

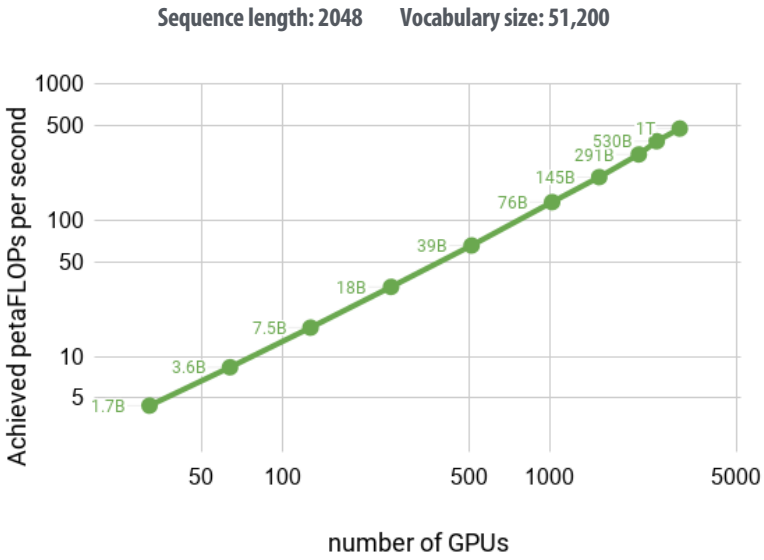
**Pipeline the forward and backward computations across micro-batches**



# Tensor, Data, Pipeline Parallelism

| Model size (parameters) | Attention heads | Hidden size | Number layers | Tensor parallel size | Pipeline parallel size | Model parallel size | Data parallel size | Number GPUs | Batch size | % peak flops |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.7B | 24 | 2304 | 24 | 1 | 1 | 1 | 32 | 32 | 512 | 44% |
| 3.6B | 32 | 3072 | 30 | 2 | 1 | 2 | 32 | 64 | 512 | 42% |
| 7.5B | 32 | 4096 | 36 | 4 | 1 | 4 | 32 | 128 | 512 | 41% |
| 18B | 48 | 6144 | 40 | 8 | 1 | 8 | 32 | 256 | 1024 | 41% |
| 39B | 64 | 8192 | 48 | 8 | 2 | 16 | 32 | 512 | 1536 | 41% |
| 76B | 80 | 10240 | 60 | 8 | 4 | 32 | 32 | 1024 | 1792 | 43% |
| 145B | 96 | 12288 | 80 | 8 | 8 | 64 | 24 | 1536 | 2304 | 44% |
| 291B | 128 | 16384 | 90 | 8 | 18 | 144 | 15 | 2160 | 2430 | 45% |
| 530B | 128 | 20480 | 105 | 8 | 35 | 280 | 9 | 2520 | 2520 | 49% |
| 1T | 160 | 25600 | 128 | 8 | 64 | 512 | 6 | 3072 | 3072 | 49% |




Degree of pipeline, tensor, and data parallelism

Pipelining schedule

Global batch size

Microbatch size

} Each of these influence amount of communication, size of pipeline bubble, memory footprint

# Reducing energy consumption idea 1: use specialized processing

**(use the right processor for the job)**

# Reducing energy consumption idea 2: move less data

# Data Access has high energy cost

**Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory**

- Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption

**"Ballpark" numbers** [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

- Integer op: ~ 1 pJ *
- Floating point op: ~20 pJ *
- Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
- Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

← Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!

**Implications**

- Reading 10 GB/sec from memory: ~1.6 watts
- Entire power budget for mobile GPU: ~1 watt (remember phone is also running CPU, display, radios, etc.)
- iPhone 16 battery: ~14 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
- Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

# Moving data is costly!

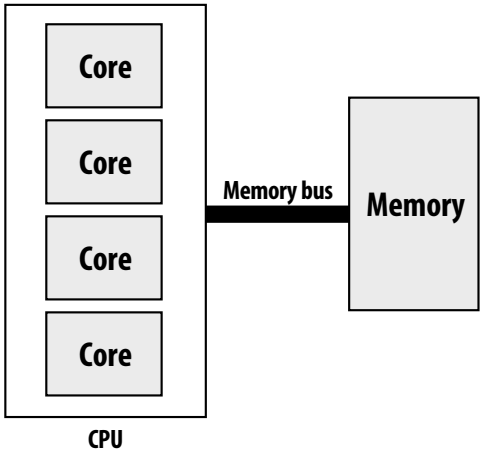
## Data movement limits performance

Many processing elements…

= higher overall rate of memory requests

= need for more memory bandwidth

(result: bandwidth-limited execution)

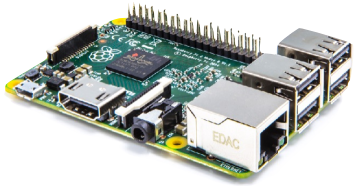



## Data movement has high energy cost

~ 0.9 pJ for a 32-bit floating-point math op *

~ 5 pJ for a local SRAM (on chip) data access

~ 640 pJ to load 32 bits from LPDDR memory

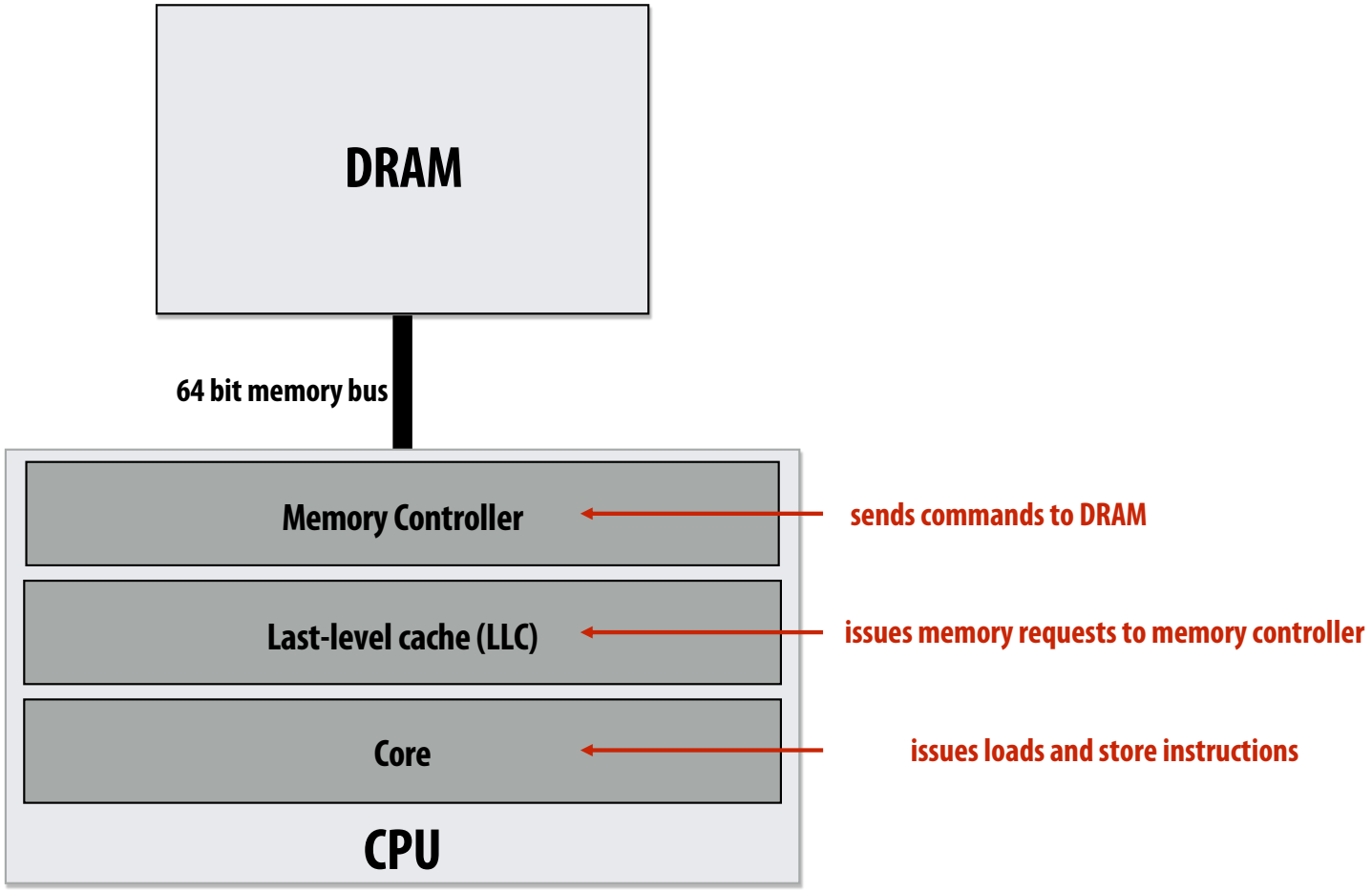


* Source: [Han, ICLR 2016], 45 nm CMOS assumption

# Accessing DRAM

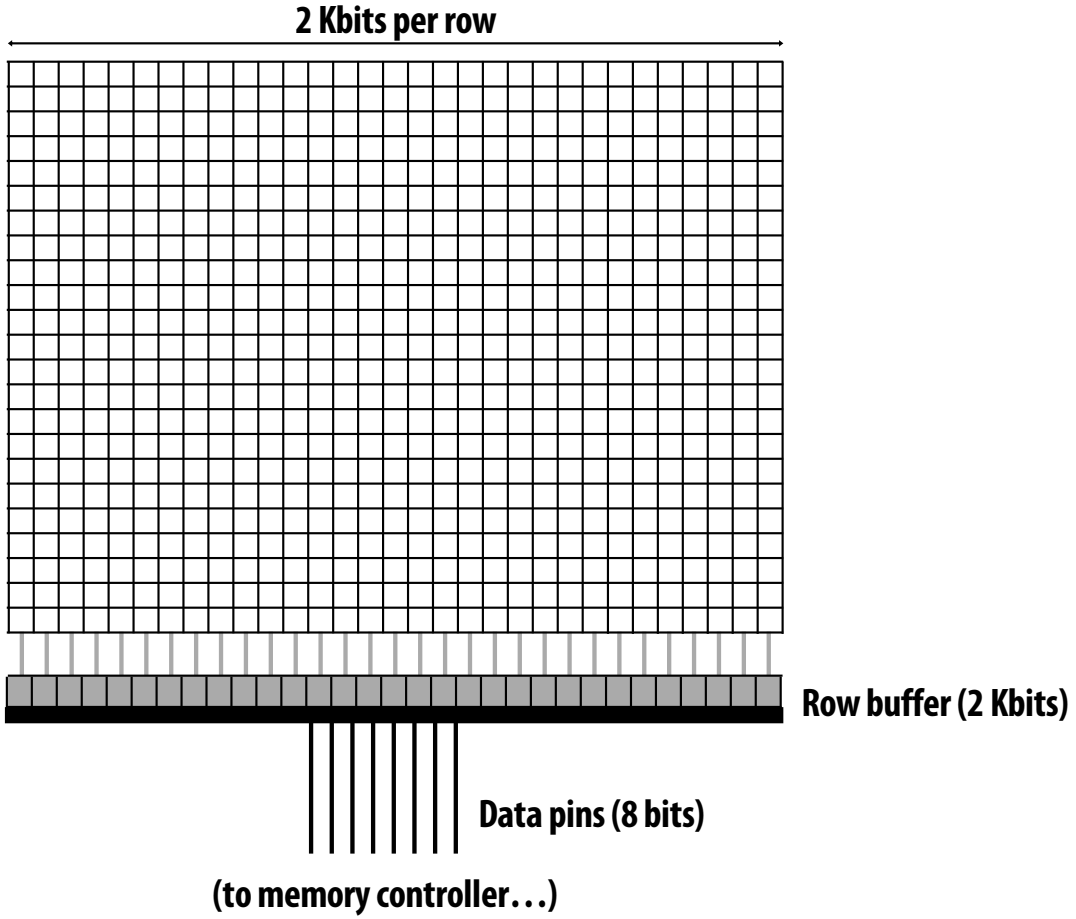
## (a basic tutorial on how DRAM works)

# The memory system



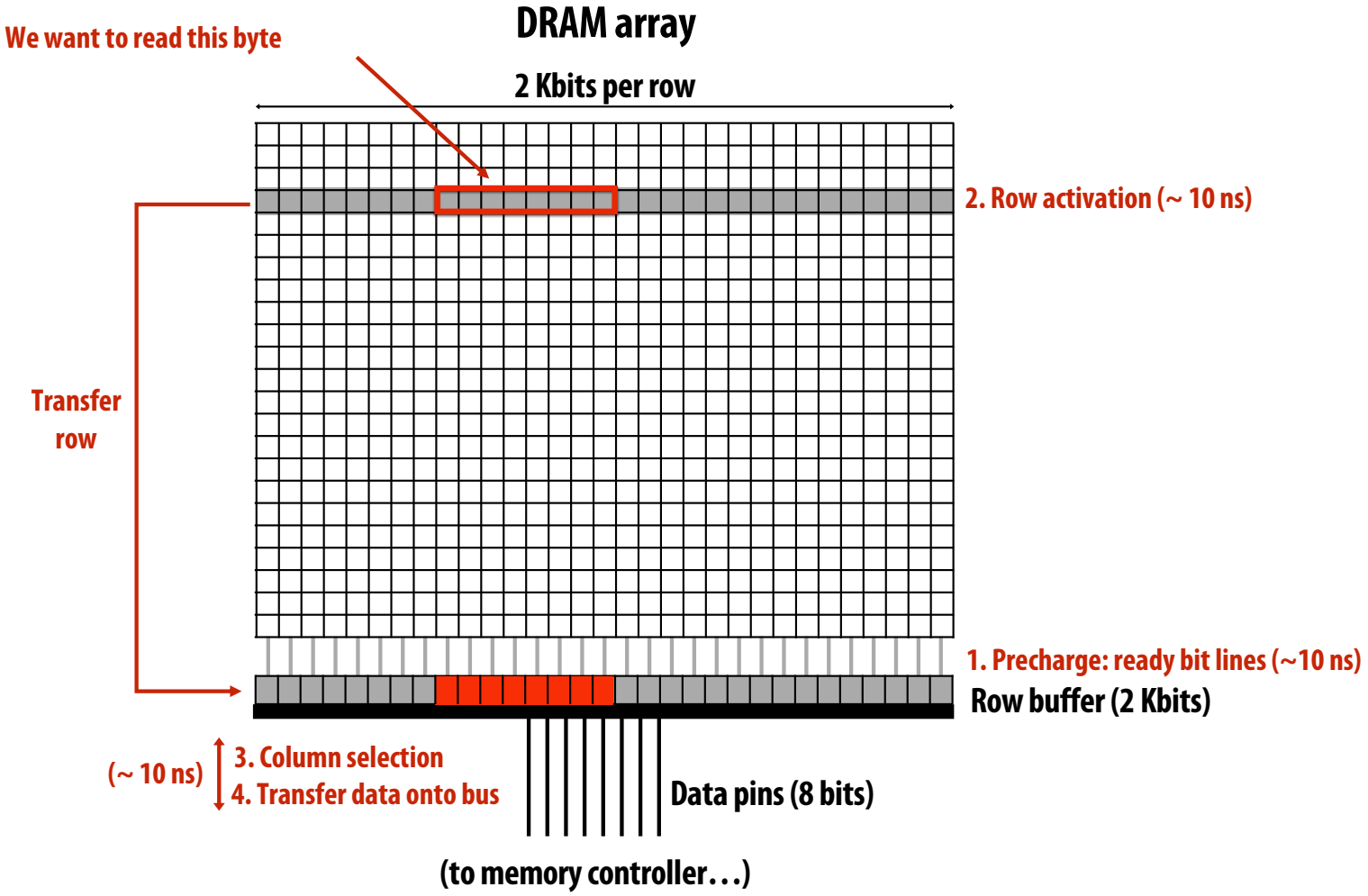
# DRAM array

1 transistor + capacitor per "bit" (Recall: a capacitor stores charge)



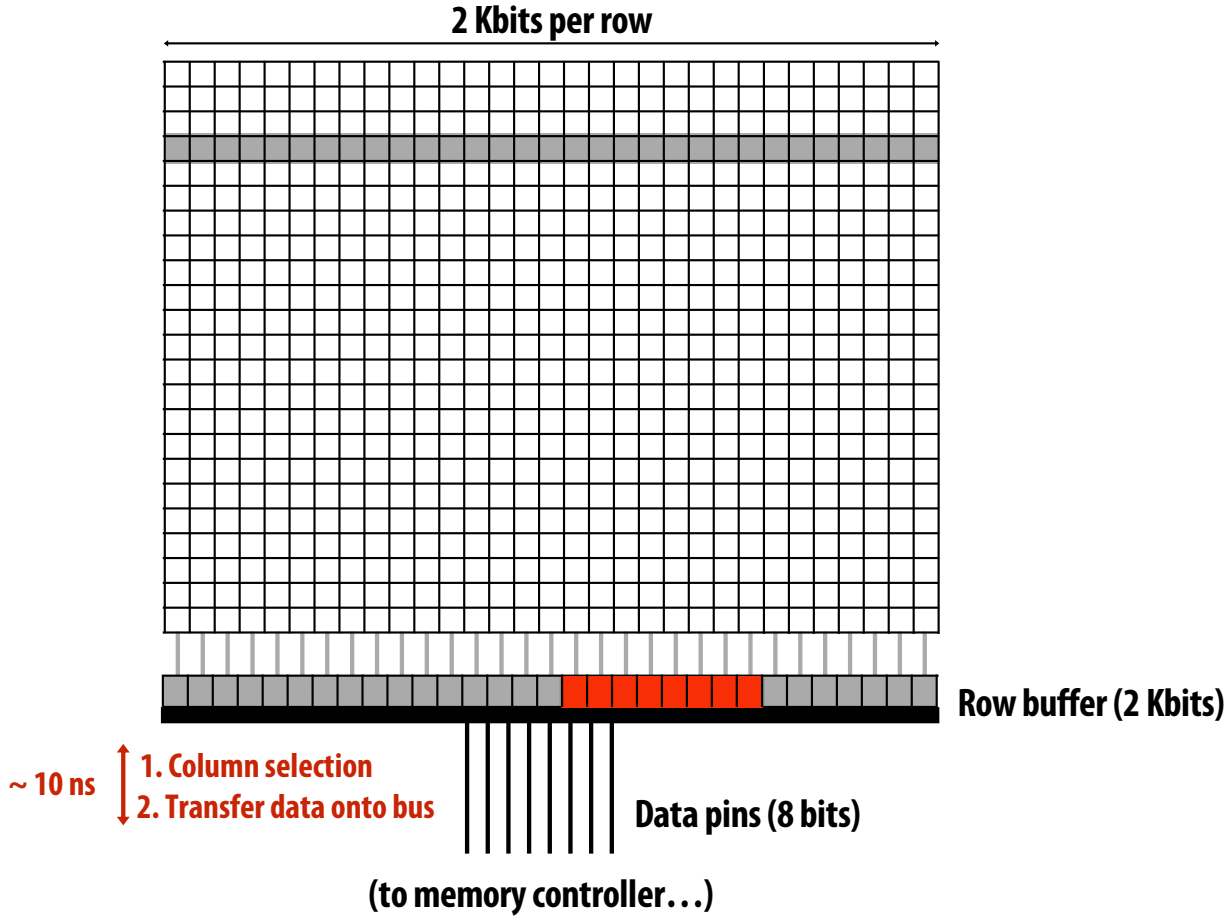
# DRAM operation (load one byte)

Estimated latencies are in units of memory clocks: DDR3-1600



# Load next byte from (already active) row

**Lower latency operation: can skip precharge and row activation steps**



# DRAM access latency is not fixed

## Best case latency: read from active row

- Column access time (CAS)

## Worst case latency: bit lines not ready, read from new row

- Precharge (PRE) + row activate (RAS) + column access (CAS)

Precharge readies bit lines and writes row buffer contents back into DRAM array (read was destructive)

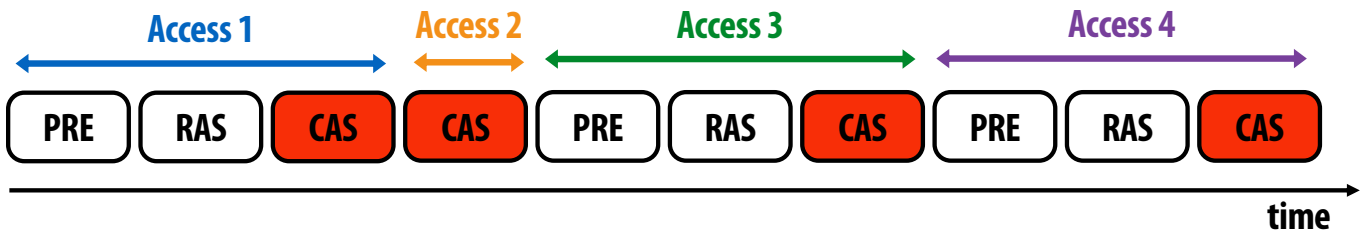
## Question 1: when to execute precharge?

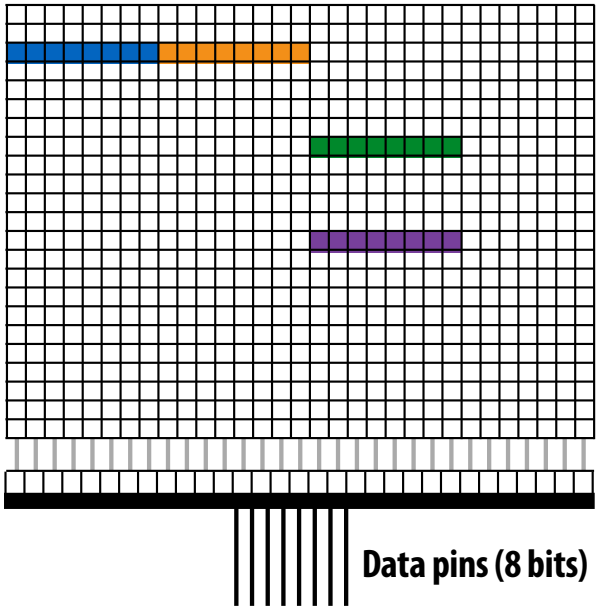
After each column access?

Only when new row is accessed?

## Question 2: how to handle latency of DRAM access?

# Problem: low pin utilization due to latency of access

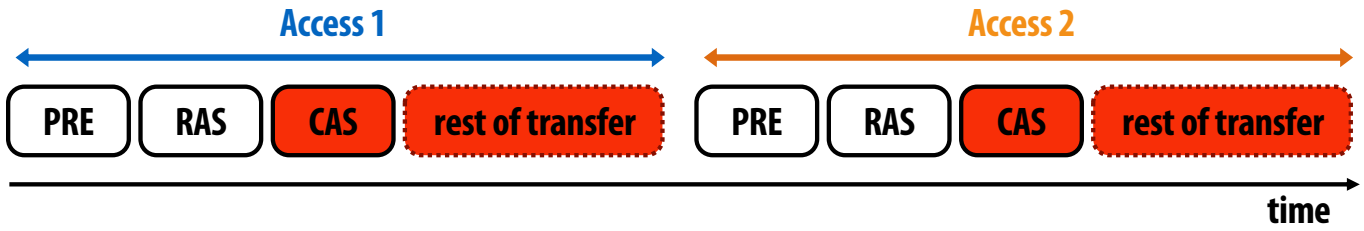


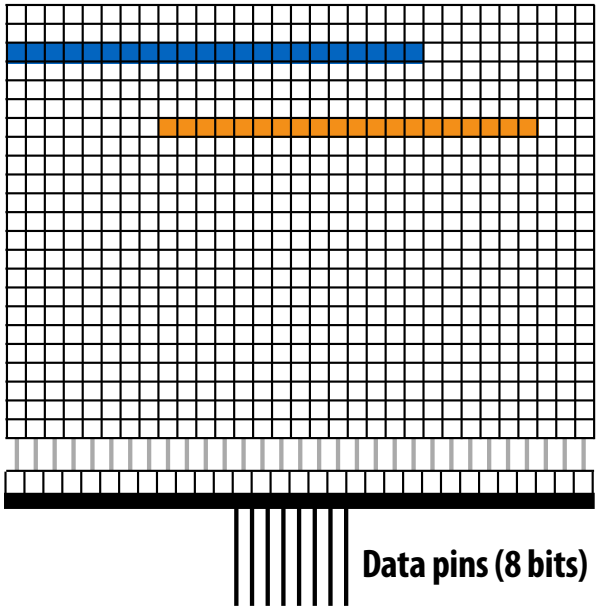




**Data pins in use only a small fraction of time**
**(red = data pins busy)**

**This is bad since they are the scarcest resource!**

# DRAM burst mode




**Idea: amortize latency over larger transfers**

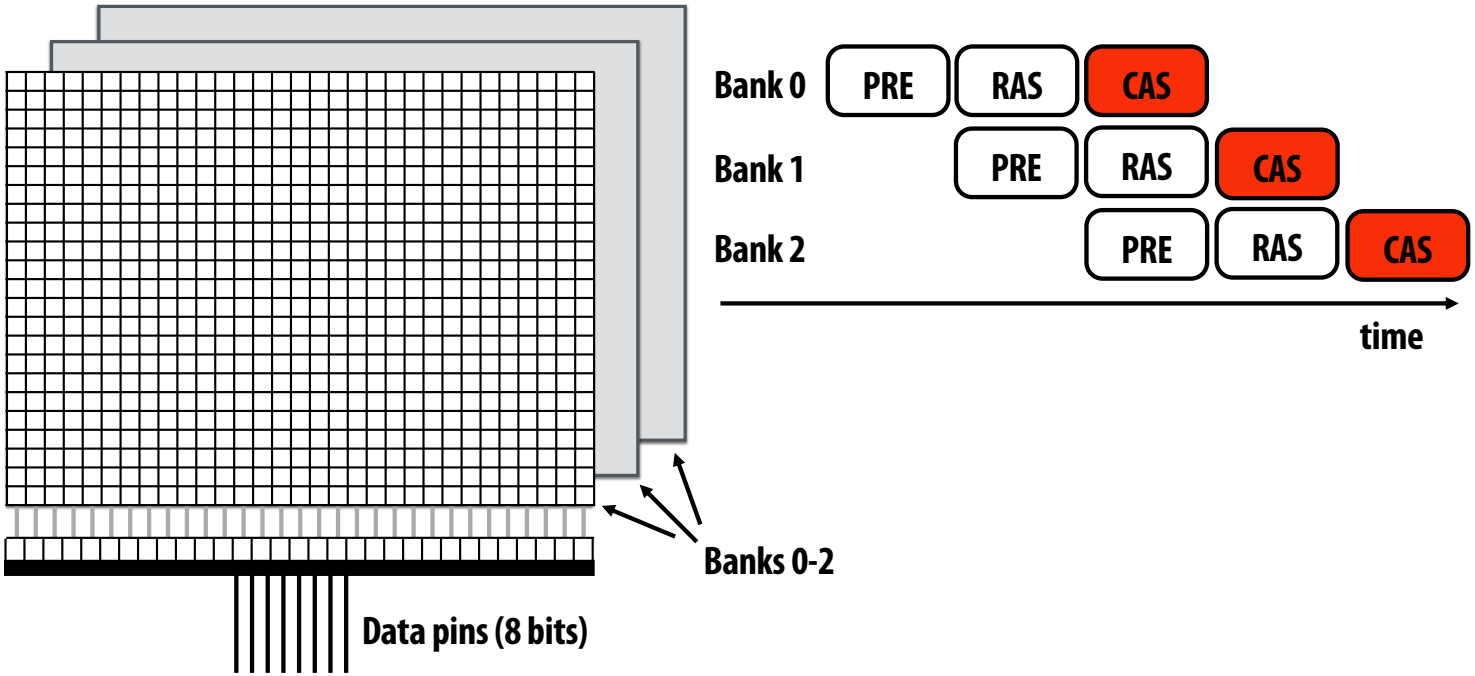
**Each DRAM command describes bulk transfer**
**Bits placed on output pins in consecutive clocks**

# DRAM chip consists of multiple banks

**All banks share same pins (only one transfer at a time)**

**Banks allow for pipelining of memory requests**

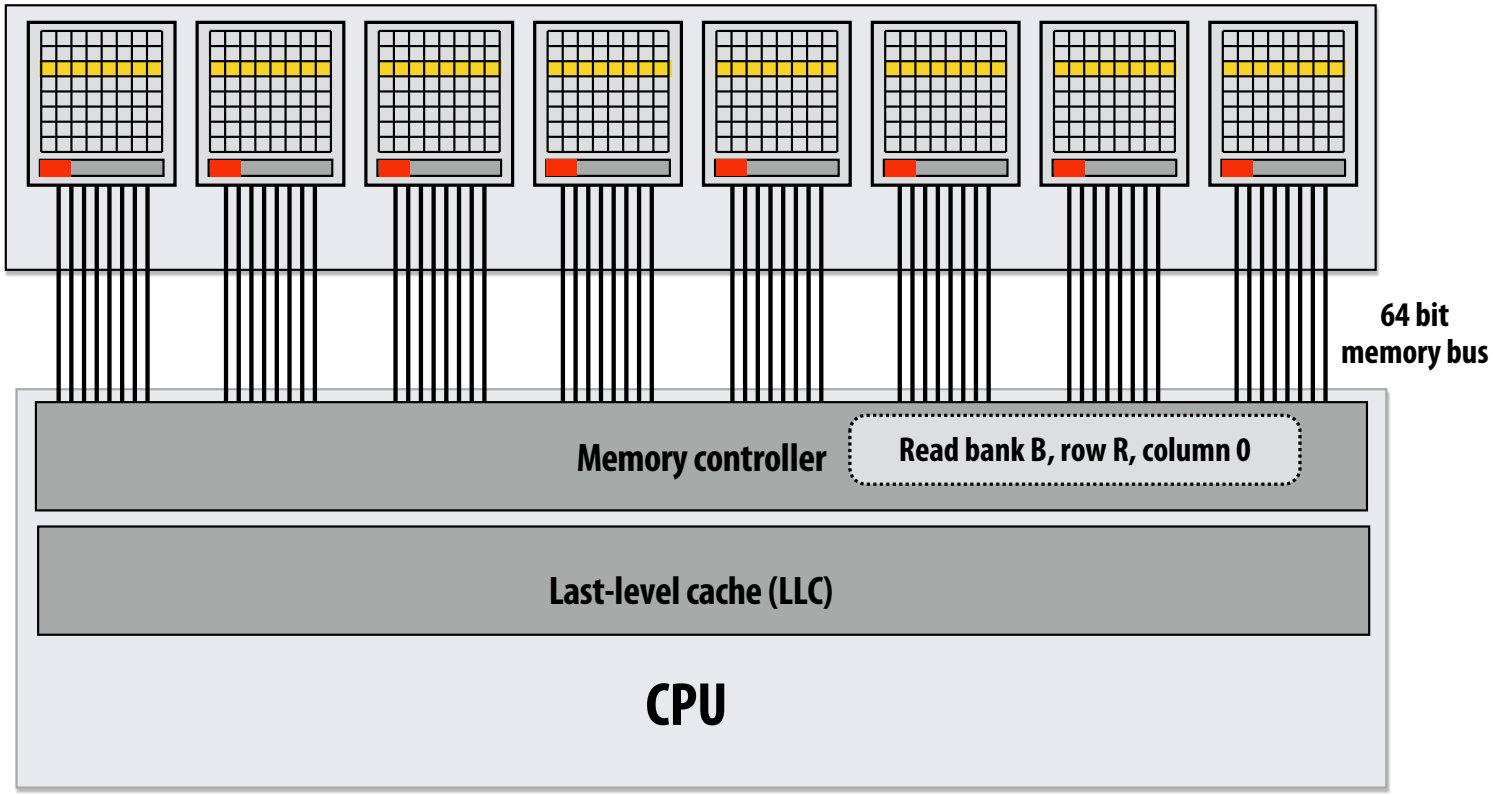
- **Precharge/activate rows/send column address to one bank while transferring data from another**
- **Achieves high data pin utilization**



# Organize multiple chips into a DIMM

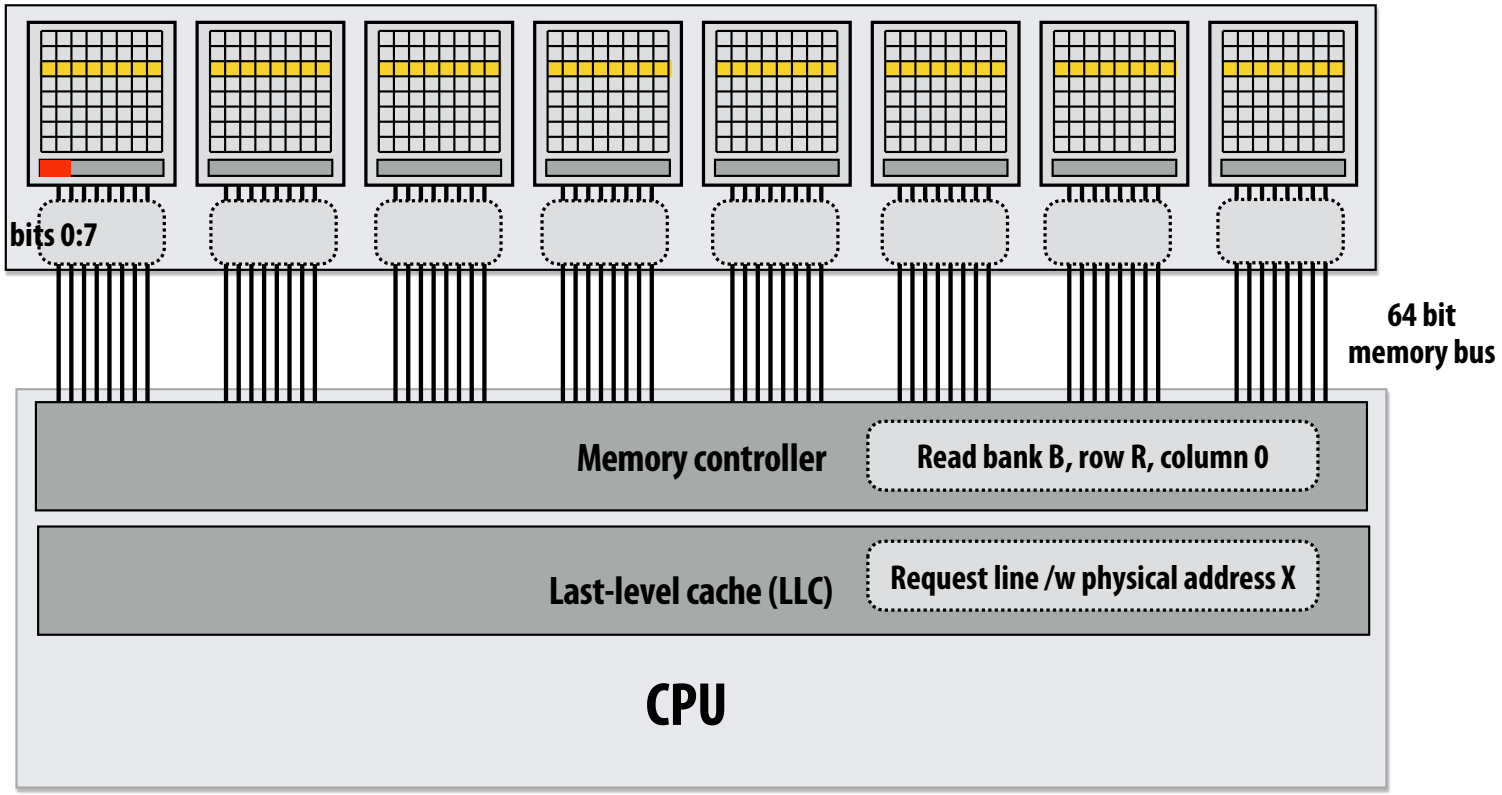
**Example: Eight DRAM chips (64-bit memory bus)**

**Note: DIMM appears as a single, higher capacity, wider interface DRAM module to the memory controller. Higher aggregate bandwidth, but minimum transfer granularity is now 64 bits.**



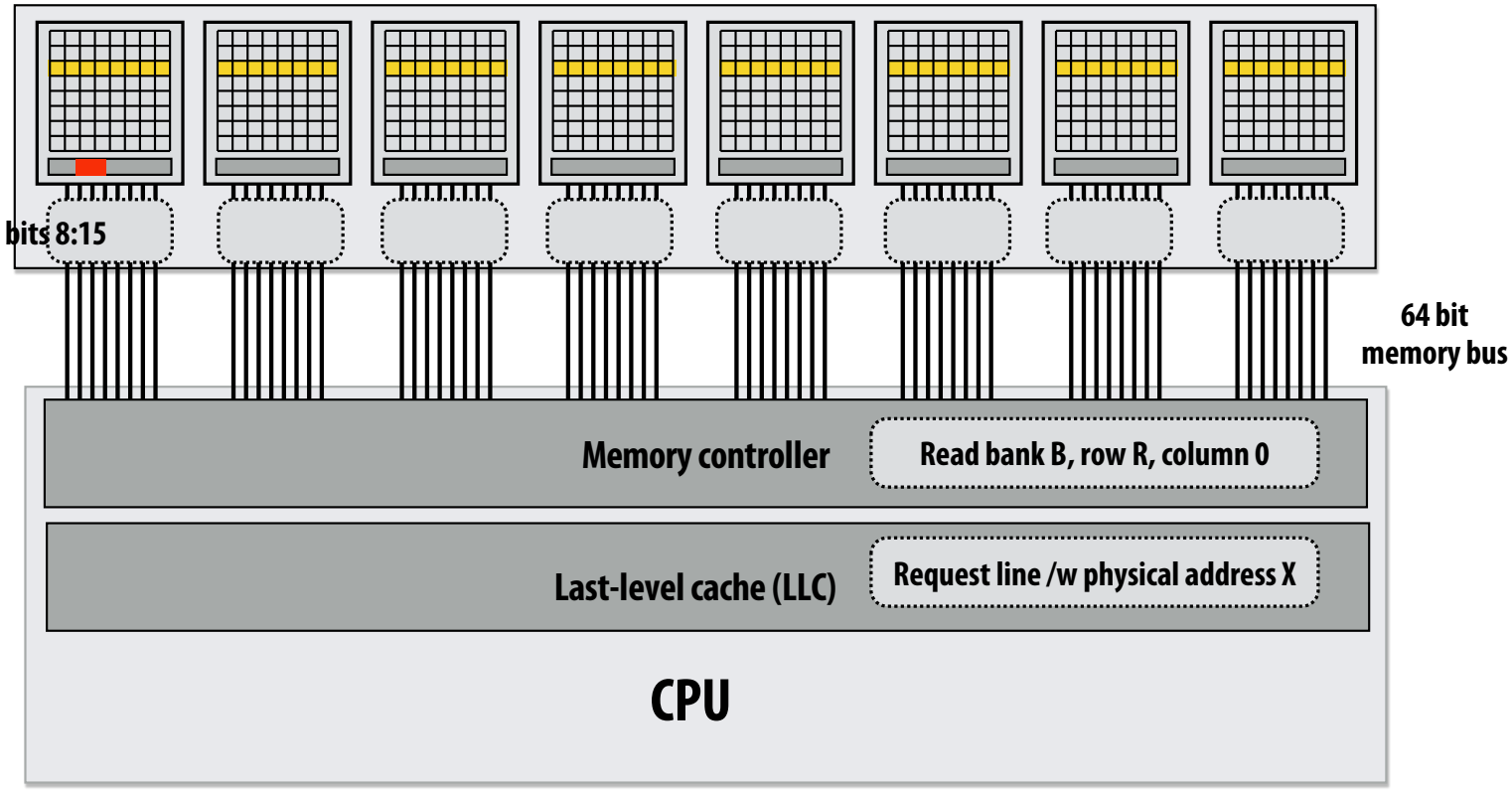
# Reading one 64-byte (512 bit) cache line (the wrong way)

**Assume: consecutive physical addresses mapped to same row of same chip**
**Memory controller converts physical address to DRAM bank, row, column**



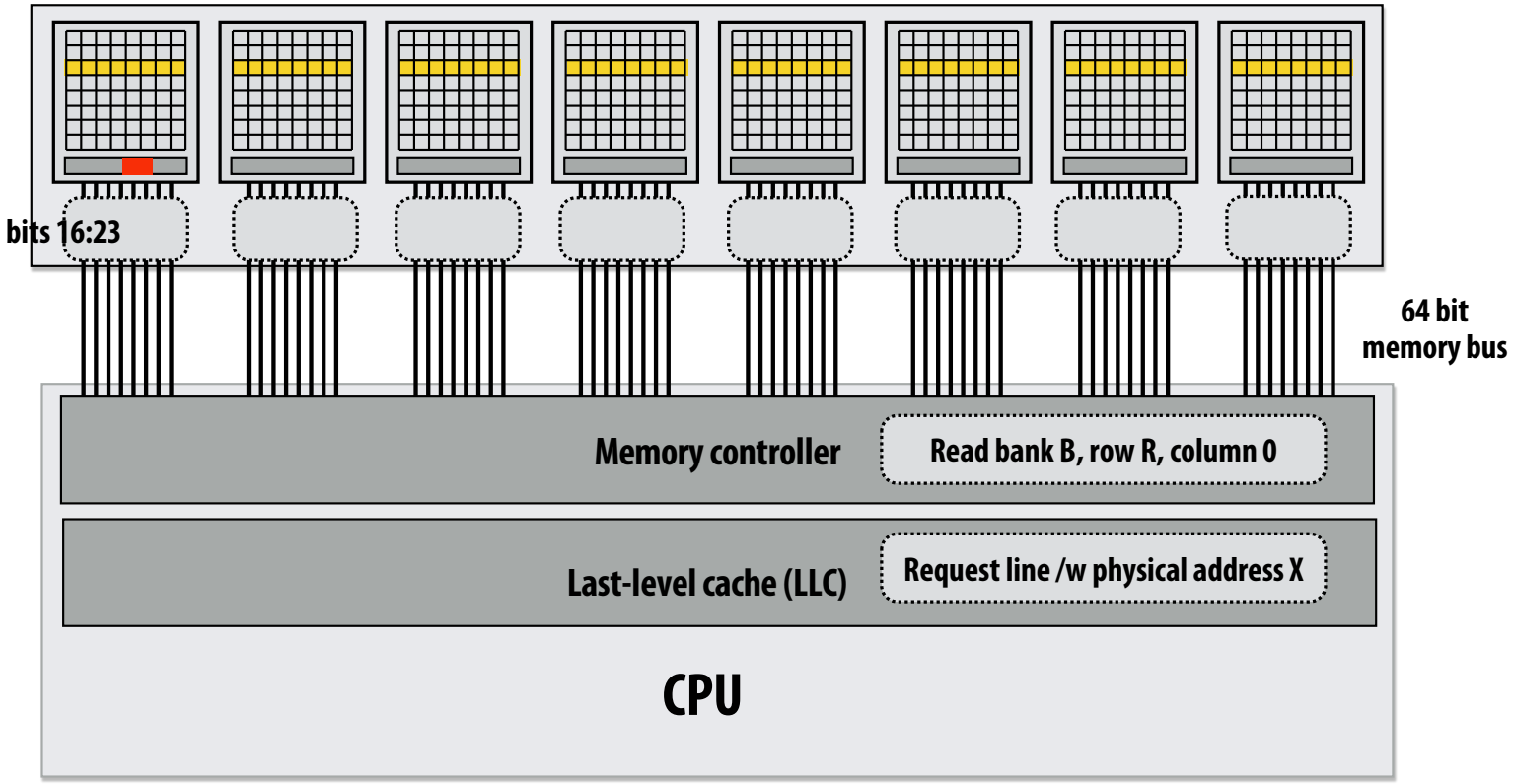
# Reading one 64-byte (512 bit) cache line (the wrong way)

**All data for cache line serviced by the same chip**
**Bytes sent consecutively over same pins**



# Reading one 64-byte (512 bit) cache line (the wrong way)

**All data for cache line serviced by the same chip**
**Bytes sent consecutively over same pins**

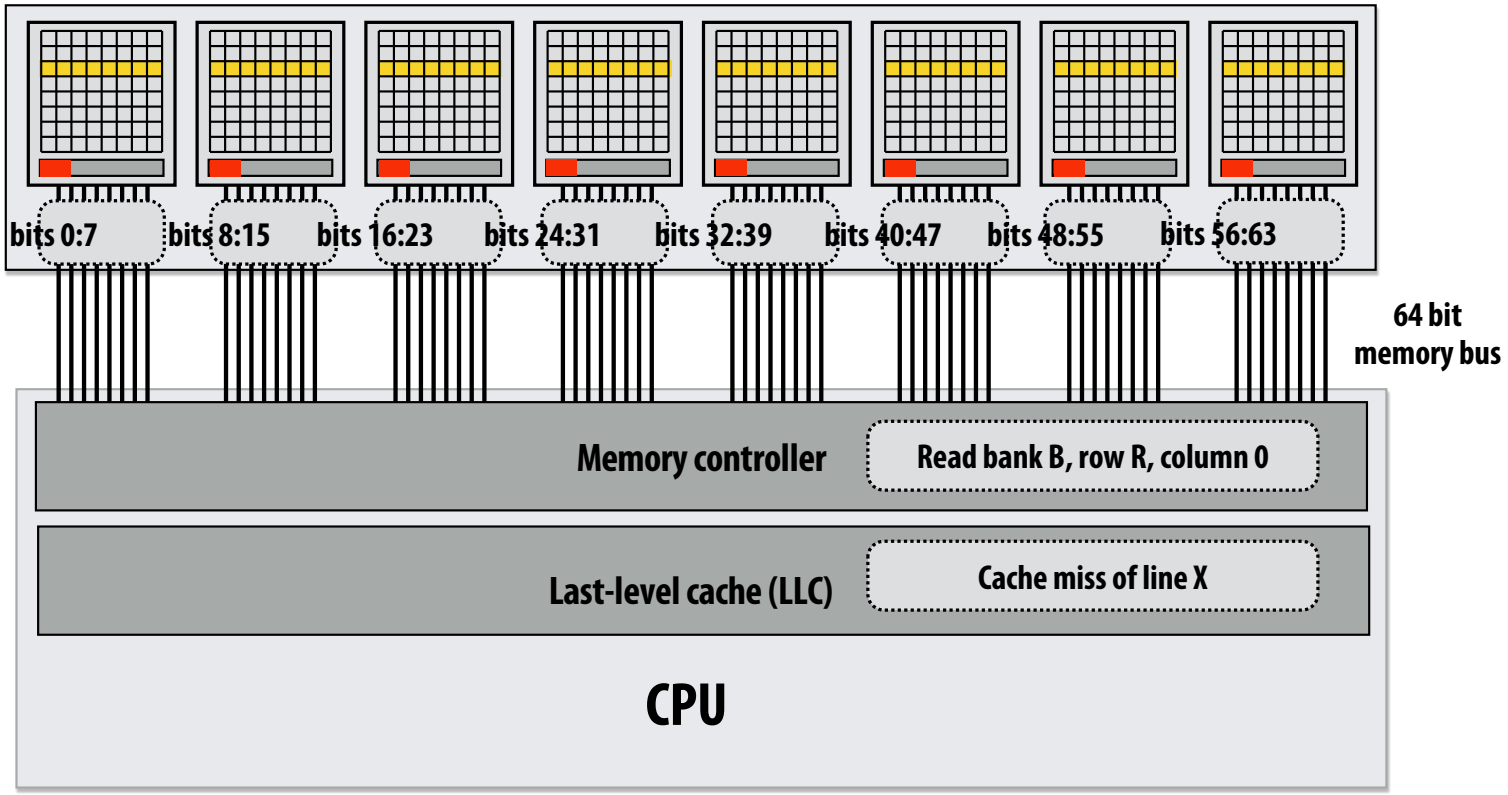


# Reading one 64-byte (512 bit) cache line

Memory controller converts physical address to DRAM bank, row, column

Here: physical addresses are interleaved across DRAM chips at byte granularity

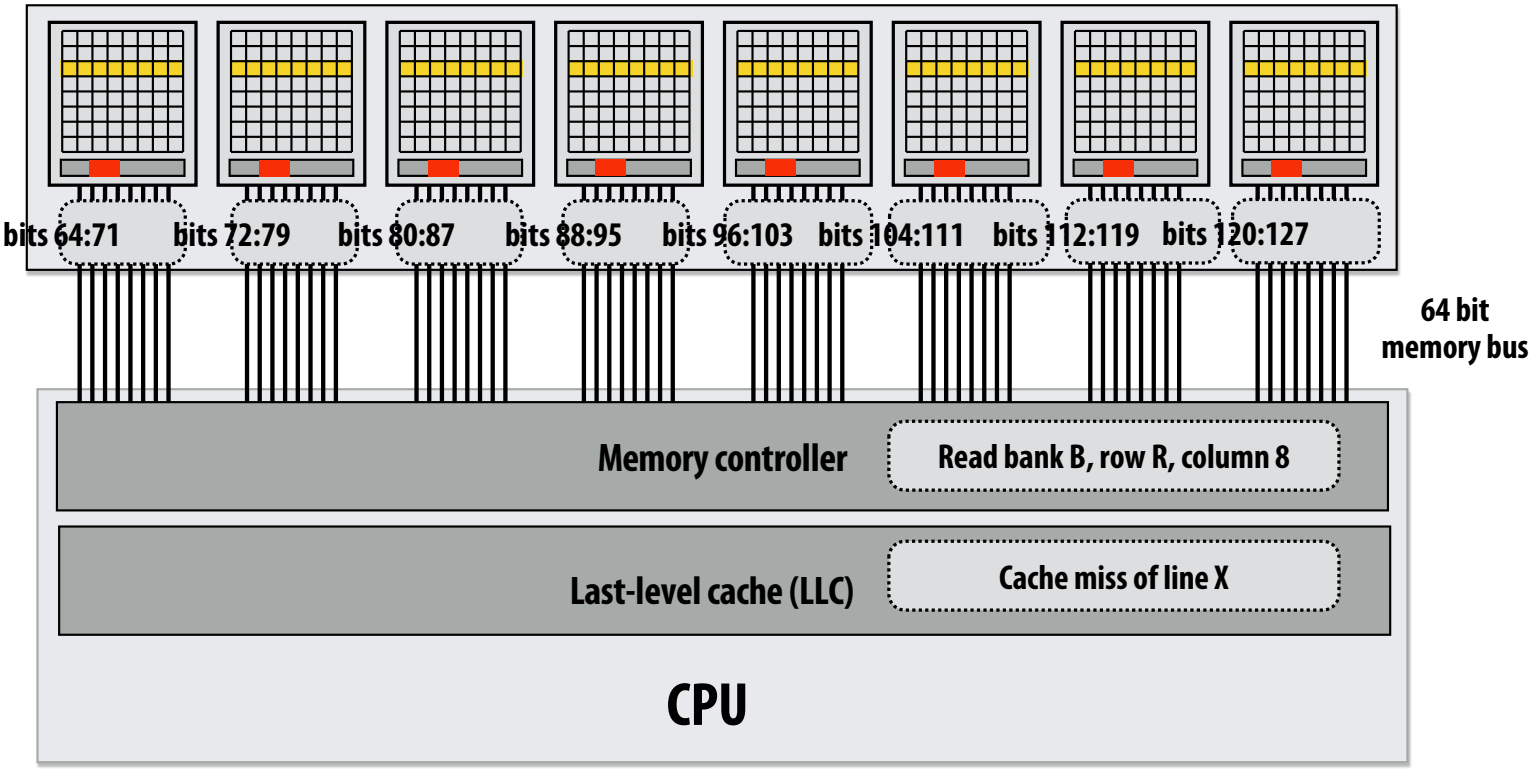
DRAM chips transmit first 64 bits in parallel



# Reading one 64-byte (512 bit) cache line

**DRAM controller requests data from new column ***

**DRAM chips transmit next 64 bits in parallel**



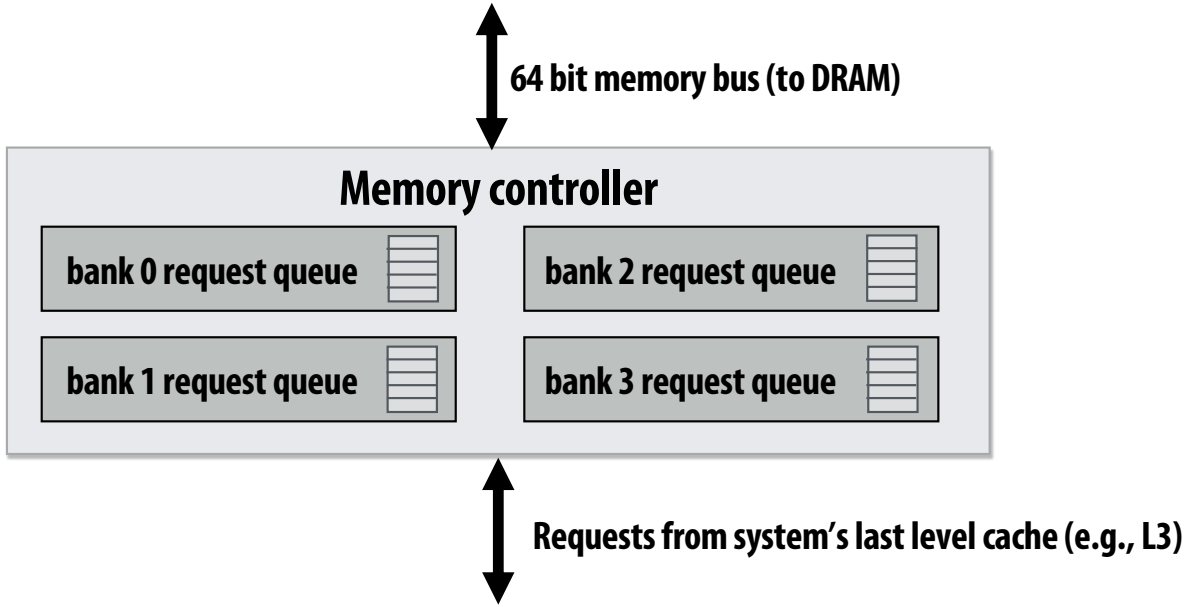

* Recall modern DRAM's support burst mode transfer of multiple consecutive columns, which would be used here

# Memory controller is a memory request scheduler

## Receives load/store requests from LLC

## Conflicting scheduling goals

- Maximize throughput, minimize latency, minimize energy consumption
- Common scheduling policy: FR-FCFS (first-ready, first-come-first-serve)
  - Service requests to currently open row first (maximize row locality)
  - Service requests to other rows in FIFO order
- Controller may coalesce multiple small requests into large contiguous requests (to take advantage of DRAM "burst modes")

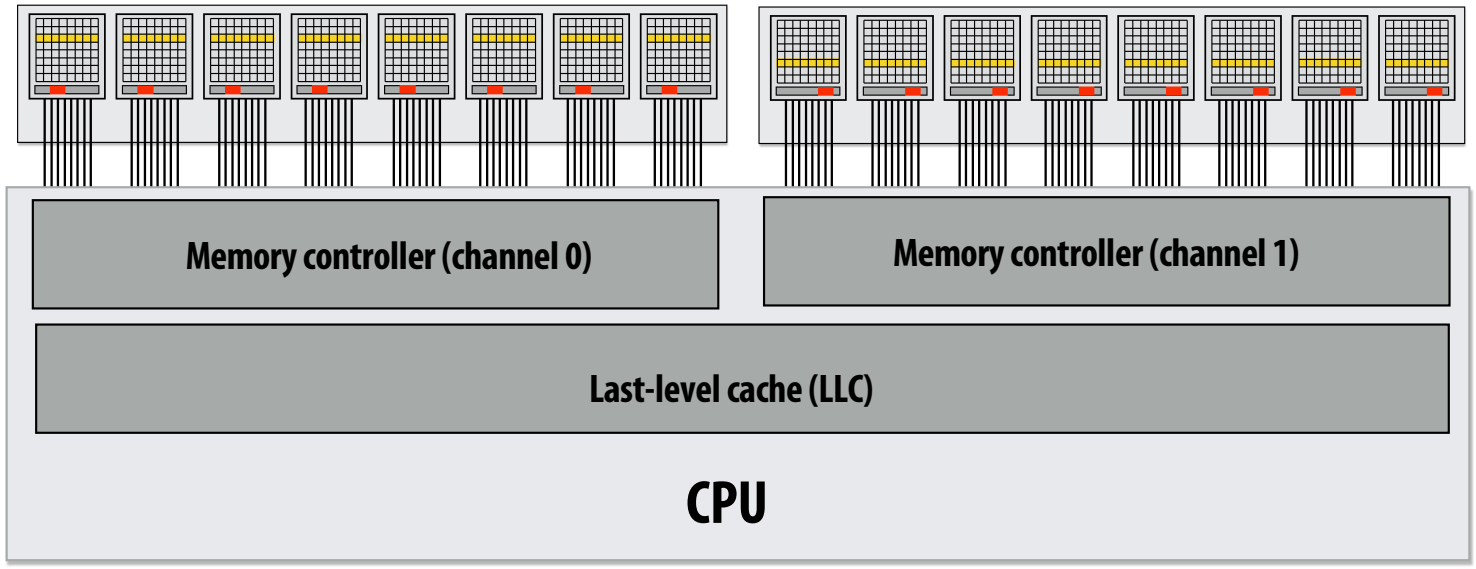


# Dual-channel memory system

Increase throughput by adding memory channels (effectively widen bus)

Below: each channel can issue independent commands

- Different row/column is read in each channel
- Simpler setup: use single controller to drive same command to multiple channels



# Example: DDR4 memory

## DDR4 2400

**Processor: Intel® Core™ i7-7700K Processor (in Myth cluster)**

- 64-bit memory bus x 1.2GHz x 2 transfers per clock* = 19.2GB/s per channel
- 2 channels = 38.4 GB/sec
- ~13 nanosecond CAS

### Memory system details from Intel's site:

**Memory Specifications**

| | |
|---|---|
| Max Memory Size (dependent on memory type) | 64 GB |
| Memory Types | DDR4-2133/2400, DDR3L-1333/1600 @ 1.35V |
| Max # of Memory Channels | 2 |
| ECC Memory Supported ‡ | No |

*** DDR stands for "double data rate"**

https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html

# DRAM summary

## DRAM access latency can depend on many low-level factors

- Discussed today:
  - State of DRAM chip: row hit/miss? is recharge necessary?
  - Buffering/reordering of requests in memory controller

## Significant amount of complexity in a modern multi-core processor has moved into the design of memory controller

- Responsible for scheduling ten's to hundreds of outstanding memory requests
- Responsible for mapping physical addresses to the geometry of DRAMs
- Area of active computer architecture research

**Modern architecture challenge:**
**improving memory performance:**

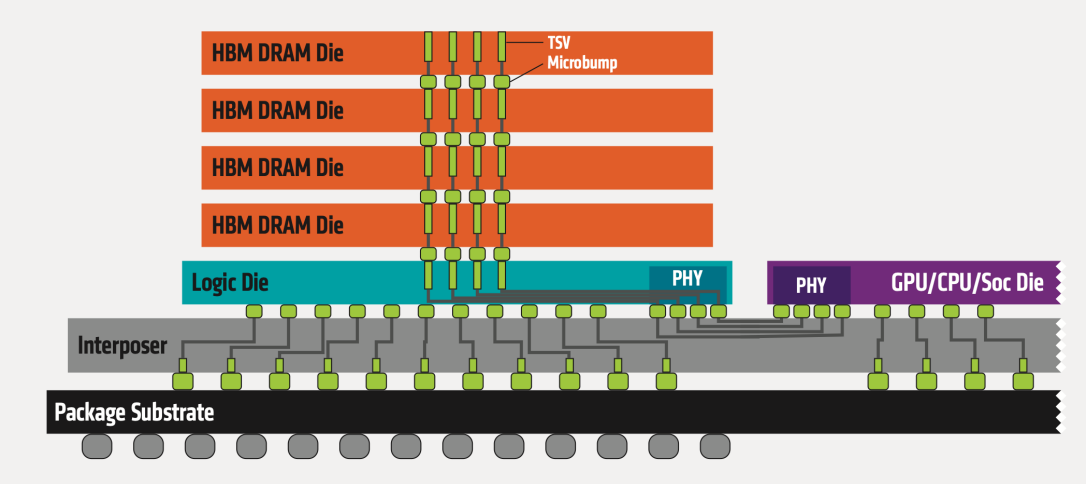
**Decrease distance data must move by**
**locating memory closer to processors**

**(enables shorter, but wider interfaces)**

# Increase bandwidth, reduce power by chip stacking

## Enabling technology: 3D stacking of DRAM chips

- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
- TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack "logic layer" is memory controller, manages requests from processor
- Silicon "interposer" serves as high-bandwidth interconnect between DRAM stack and processor




**Technologies:**
**Micron/Intel Hybrid Memory Cube (HBC)**
**High-bandwidth memory (HBM) - 1024 bit interface to stack**

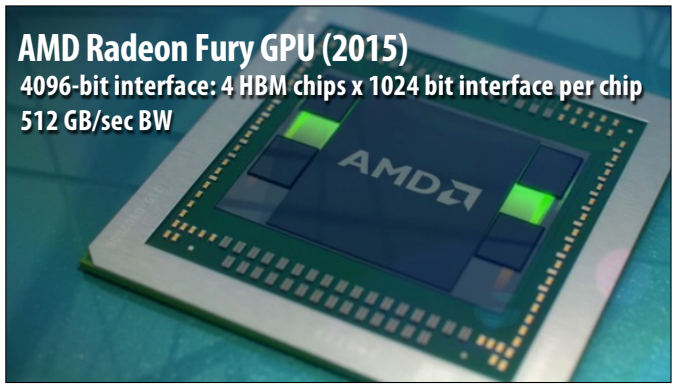
Image credit: AMD

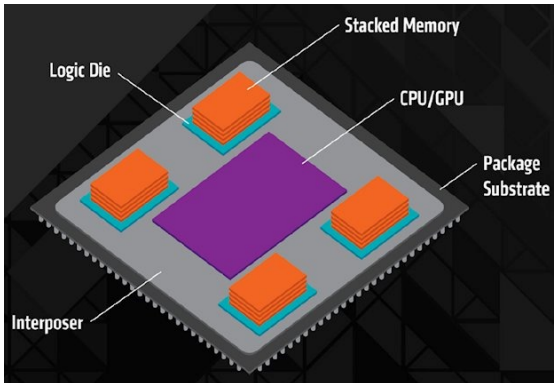
# HBM Advantages

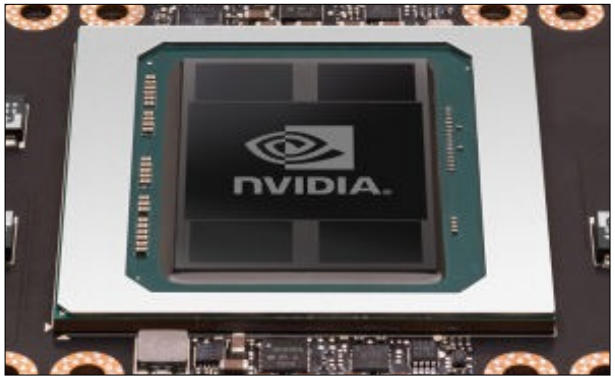
## More Bandwidth
## High Power Efficiency
## Small Form Factor

| | DDR4 | LPDDR4(X) | GDDR6 | HBM2 | HBM2E (JEDEC) | HBM3 (TBD) |
|---|---|---|---|---|---|---|
| Data rate | 3200Mbps | 3200Mbps (up to 4266 Mbps) | 14Gbps (up to 16Gb ps) | 2.4Gbps | 2.8Gbps | >3.2Gbps (TBD) |
| Pin count | x4/x8/x16 | x16/ch (2ch per die) | x16/x32 | x1024 | x1024 | x1024 |
| Bandwidth | 5.4GB/s | 12.8(17)GB/s | 56GB/s | 307GB/s | 358GB/s | >500GB/s |
| Density (per package) | 4Gb/8Gb | 8Gb/16Gb/2 4Gb/32Gb | 8Gb/16Gb | 4GB/8GB | 8GB/16GB | 8GB/16GB/ 24GB (TBD) |

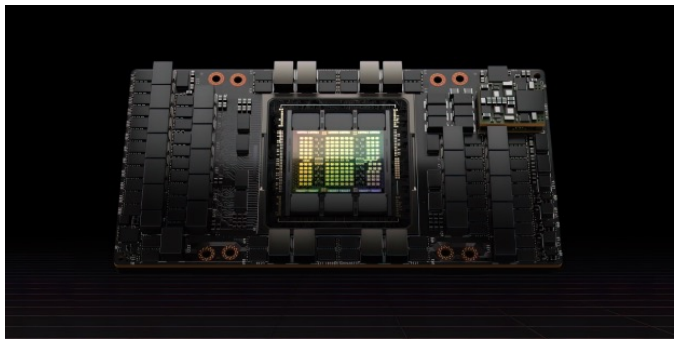
# GPUs are adopting HBM technologies



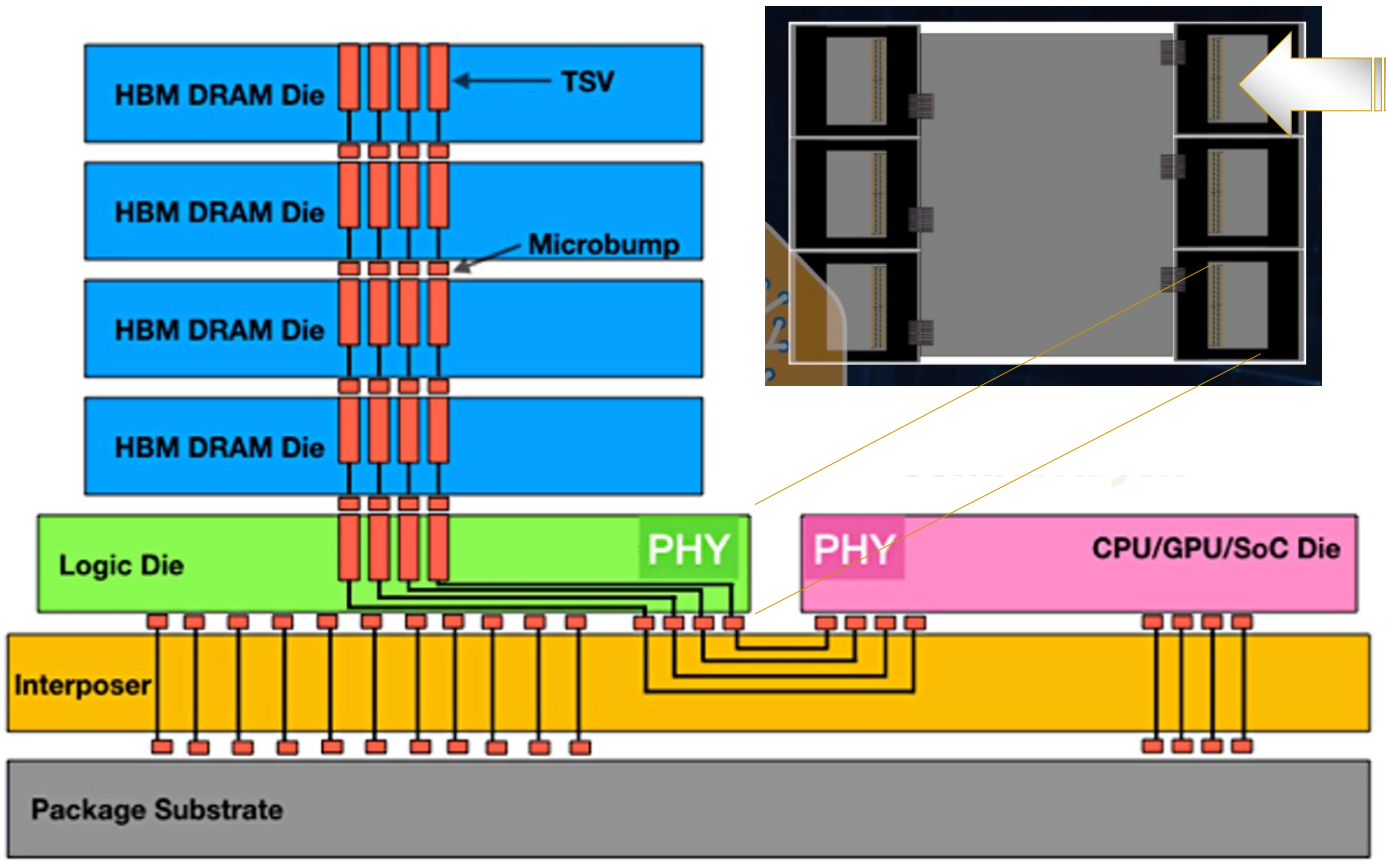




**NVIDIA P100 GPU (2016)**
**4096-bit interface: 4 HBM2 chips x 1024 bit interface per chip**
**720 GB/sec peak BW**
**4 x 4 GB = 16 GB capacity**



**NVIDIA H100 GPU (2022)**
**6144-bit interface: 6 HBM3 stacks x 1024 bit interface per stack**
**3.2 TB/sec peak BW**
**80 GB capacity**

# HBM4 Custom Logic Die






**LPDDR interface**

**I/O interfaces**

- **Ethernet**
- **PCI**

**Compute?**

- **SRAM cache**
- **KV cache compression**

# Summary: the memory bottleneck is being addressed in many ways

## By the application programmer

- Schedule computation to maximize locality (minimize required data movement)

## By new hardware architectures

- Intelligent DRAM request scheduling
- Bringing data closer to processor (deep cache hierarchies, 3D stacking)
- Increase bandwidth (wider memory systems)
- Ongoing research in locating limited forms of computation "in" or near memory
- Ongoing research in hardware accelerated compression (not discussed today)

## General principles

- Locate data storage near processor
- Move computation to data storage
- Data compression (trade-off extra computation for less data transfer)