

Lecture 17:

Programming Specialized Hardware

**Parallel Computing
Stanford CS149, Fall 2024**

Today's Theme

How do you design specialized HW for DNNs?

How do you program specialized hardware?

Google TPU

- **Efficient dense matrix multiply \Rightarrow systolic array**

Nvidia H100

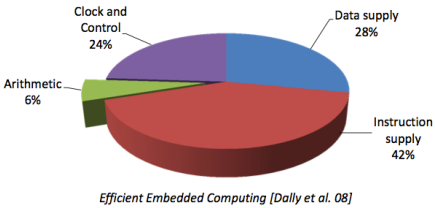
- **Asynchronous compute and memory mechanisms \Rightarrow complex programming**
- **Simplify with Thunderkittens DSL**

SambaNova SN40L

- **Dataflow architecture**
- **Programming model: tiling and streaming with metapipelining**

Recall: Energy Efficiency vs. Programmability

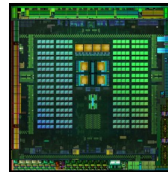
Programmability adds overhead \Rightarrow reduces efficiency



Energy-optimized CPU



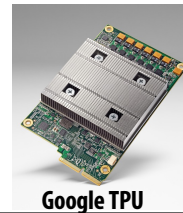
Throughput-oriented processor (GPU)



Programmable DSP



Domain Specific Accelerator



**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

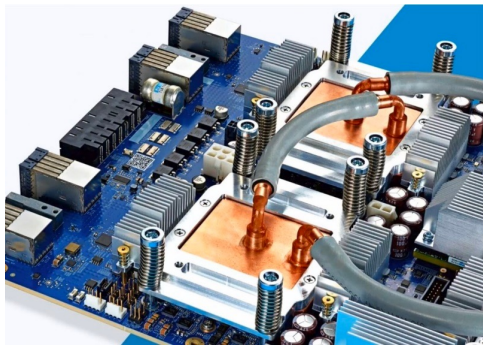
Limited domain of
programmability
with DSLs (e.g. DNN)

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

Hardware acceleration of DNN inference/training



Google TPU3



AWS Trainium 2



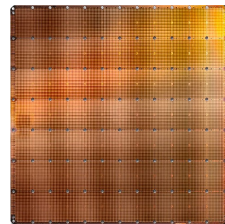
Apple Neural Engine



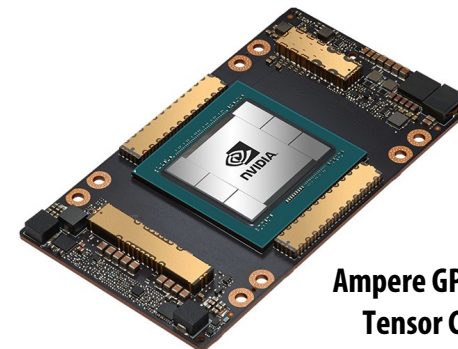
**Intel Deep Learning
Inference Accelerator**



**SambaNova
Cardinal SN10**



Cerebras Wafer Scale Engine



**Ampere GPU with
Tensor Cores**

Investment in AI hardware

SambaNova Systems Raises \$676M in Series D, Surpasses \$5B Valuation and Becomes World's Best-Funded AI Startup

SoftBank Vision Fund 2 leads round backing breakthrough platform that delivers unprecedented AI capability and accessibility to customers worldwide

April 13, 2021 09:00 AM Eastern Daylight Time

PALO ALTO, Calif. --(BUSINESS WIRE)--SambaNova Systems, the company building the industry's most advanced software, hardware and services to run AI applications, today announced a \$676 million Series D funding round led by SoftBank Vision Fund 2*. The round includes additional new investors Temasek and GIC, plus existing backers including funds and accounts managed by BlackRock, Intel Capital, GV (formerly Google V

"We're here to revolutionize the AI market, and this round greatly accelerates that mission"

[Tweet this](#)

"We're here to revolutionize the AI market, and this round gre founder and CEO. "Traditional CPU and GPU architectures hi to solve humanity's greatest technology challenges, a new ap to see a wealth of prudent investors validate that."

SambaNova's flagship offering is Dataflow-as-a-Service (Daa to jump-start enterprise-level AI initiatives, augmenting organ centers, allowing the organization to focus on its business objectives instead of infrastructure.

Artificial intelligence chip startup Cerebras Systems claims it has the "world's fastest AI supercomputer," thanks to its large Wafer Scale Engine processor that comes with 400,000 compute cores.

The Los Altos, Calif.-based startup introduced its CS-1 system at the **Supercomputing conference in Denver** last week after raising more than \$200 million in funding from investors, most recently with an \$88 million Series D round that was raised in November 2018, according to Andrew Feldman, the founder and CEO of Cerebras who was previously an executive at AMD.

AI chipmaker Graphcore raises \$222M at a \$2.77B valuation and puts an IPO in its sights


Ingrid Lunden @ingridlunden / 10:59 PM PST • December 28, 2020

Groq Closes \$300 Million Fundraise

Wed, April 14, 2021, 6:00 AM - 4 min read

With Investment Co-Led by Tiger Global Management and D1 Capital, Groq Is Well Capitalized for Accelerated Growth

MOUNTAIN VIEW, Calif., April 14, 2021 /PRNewswire/ -- Groq Inc., a leading innovator in compute accelerators for artificial intelligence (AI), machine learning (ML) and high performance computing, today announced that it has closed its Series C fundraising. Groq closed \$300 million in new funding, co-led by Tiger Global Management and D1 Capital, with participation from The Spruce House Partnership and Addition, the venture firm founded by Lee Fixel. This round brings Groq's total funding to \$367 million, of which \$300 million has been raised since the second-half of 2020, a direct result of strong customer endorsement since the company launched its first product.



Applications based on artificial intelligence — whether they are systems running autonomous services, platforms being used in drug development or to predict the spread of a virus, traffic management for 5G networks or something else altogether — require an unprecedented amount of computing power to run. And today, one of the big names in the world of designing and



Intel Acquires Artificial Intelligence Chipmaker Habana Labs

Combination Advances Intel's AI Strategy, Strengthens Portfolio of AI Accelerators for the Data Center

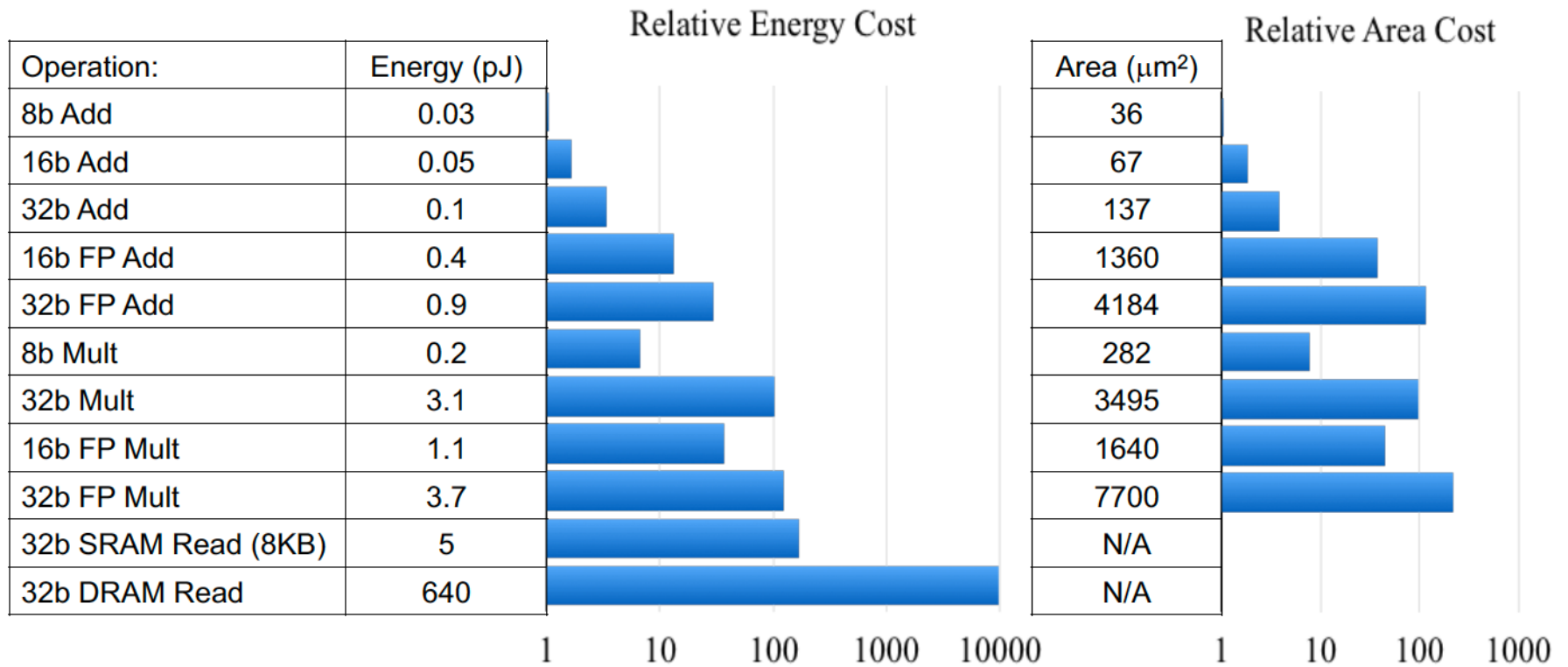
SANTA CLARA Calif., Dec. 16, 2019 – Intel Corporation today announced that it has acquired Habana Labs, an Israel-based developer of programmable deep learning accelerators for the data center for approximately \$2 billion. The combination strengthens Intel's artificial intelligence (AI) portfolio and accelerates its efforts in the nascent, fast-growing AI silicon market, which Intel expects to be greater than \$25 billion by 2024¹.

"This acquisition advances our AI strategy, which is to provide customers with solutions to fit every performance need – from the intelligent edge to the data center," said Navin Shenoy, executive vice president and general manager of the Data Platforms Group at Intel. "More specifically, Habana turbo-charges our AI offerings for the data center with a high-performance training processor family and a standards-based programming environment to address evolving AI workloads."

Numerical data formats

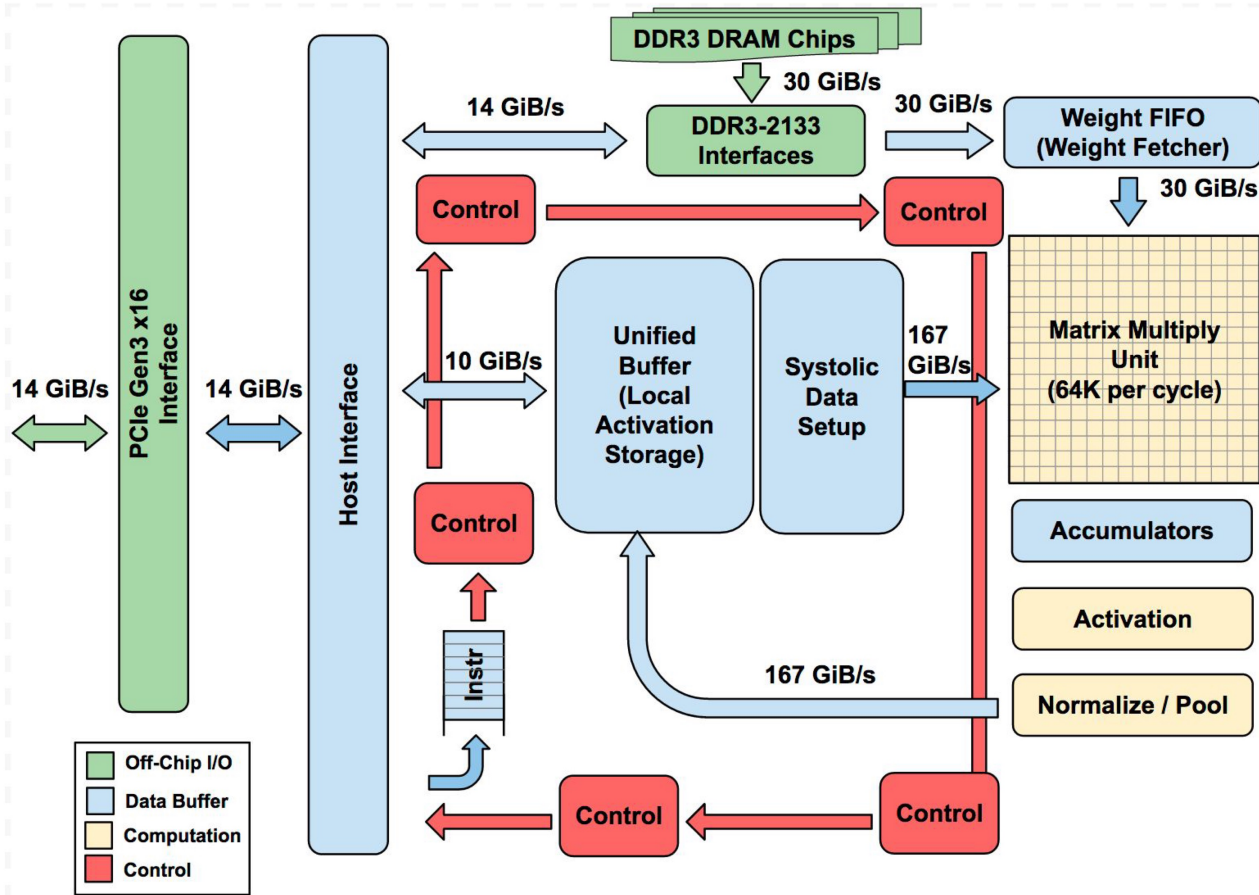
		Range	Accuracy	Reminder:
FP32		$10^{-38} - 10^{38}$.000006%	$-1^S \times (1 + (M \times 2^{-23})) \times 2^{(E-127)}$
FP16		$6 \times 10^{-5} - 6 \times 10^4$.05%	
Int32		$0 - 2 \times 10^9$	Exact	
Int16		$0 - 6 \times 10^4$	Exact	
Int8		$0 - 127$	Exact	
BF16		BF16: Same range as FP32, but lower accuracy		
BF8 E4M3		$0 - 448$		
BF8 E5M2		$0 - 57344$		

Energy and Area Cost of Compute



Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
 Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Google's TPU (v1)



Key instructions:

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

Figure credit: Jouppi et al. 2017

TPU area proportionality

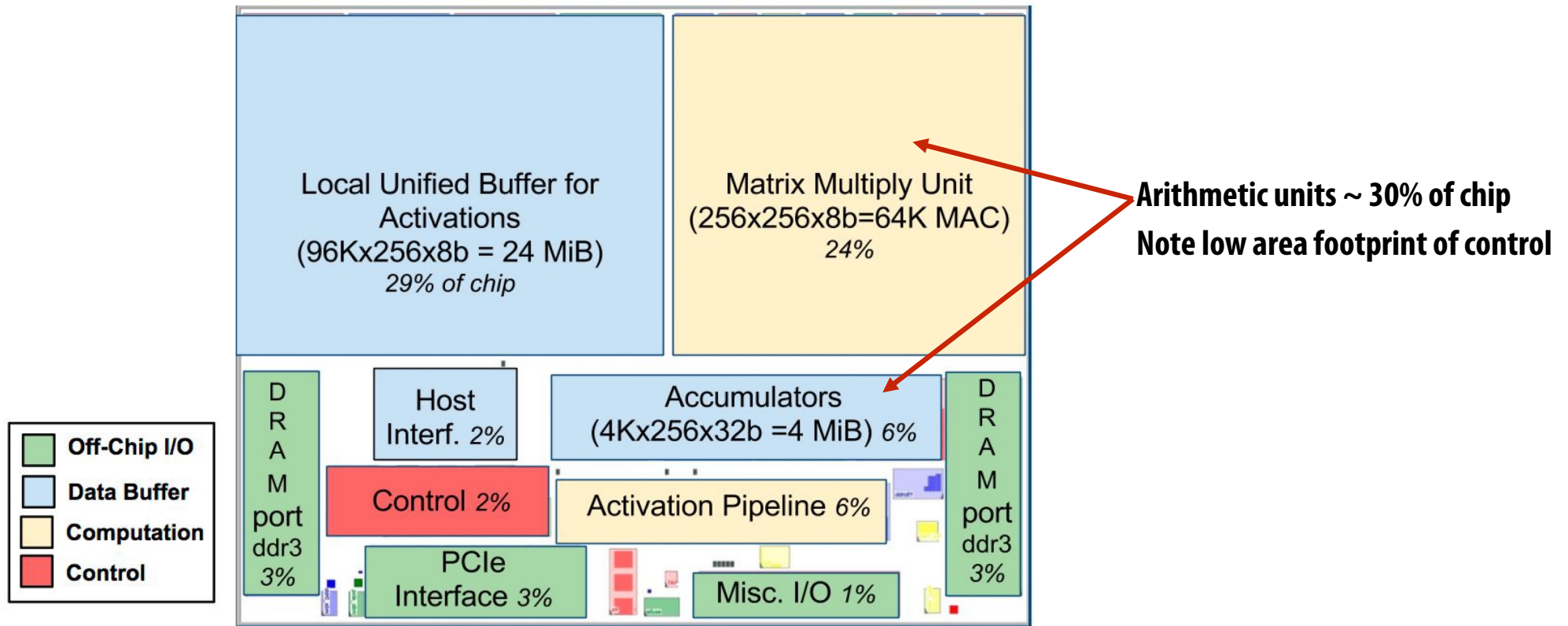
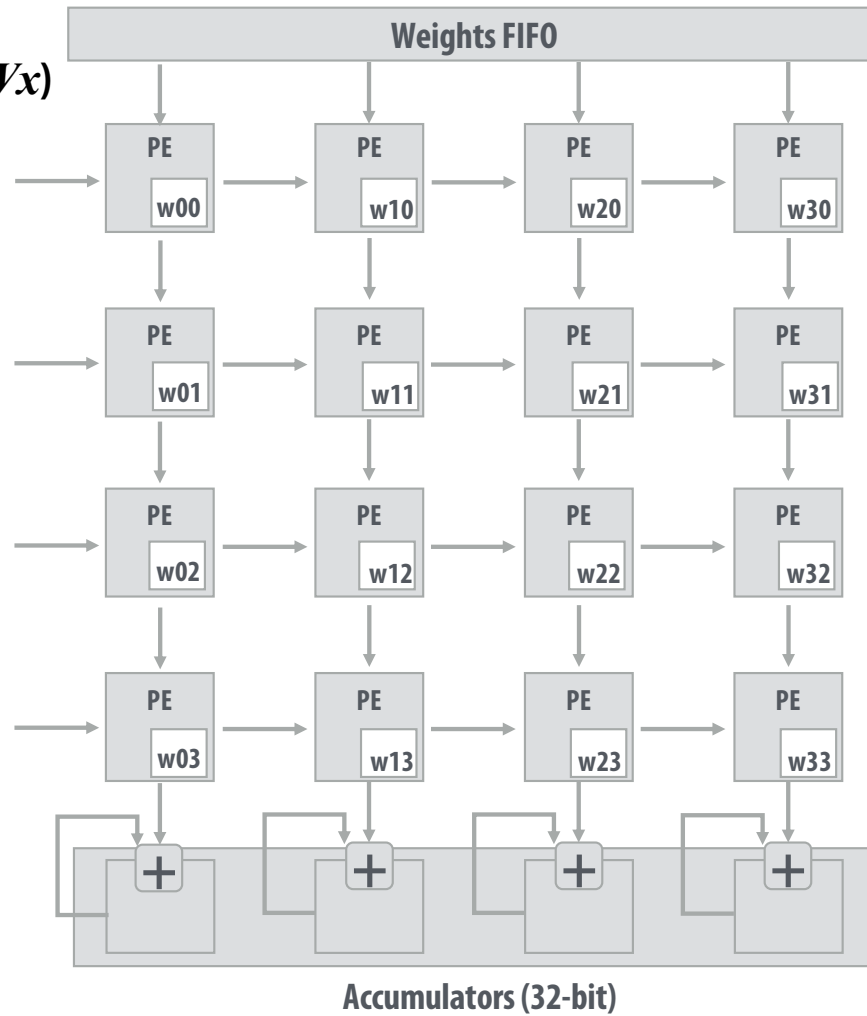


Figure credit: Jouppi et al. 2017

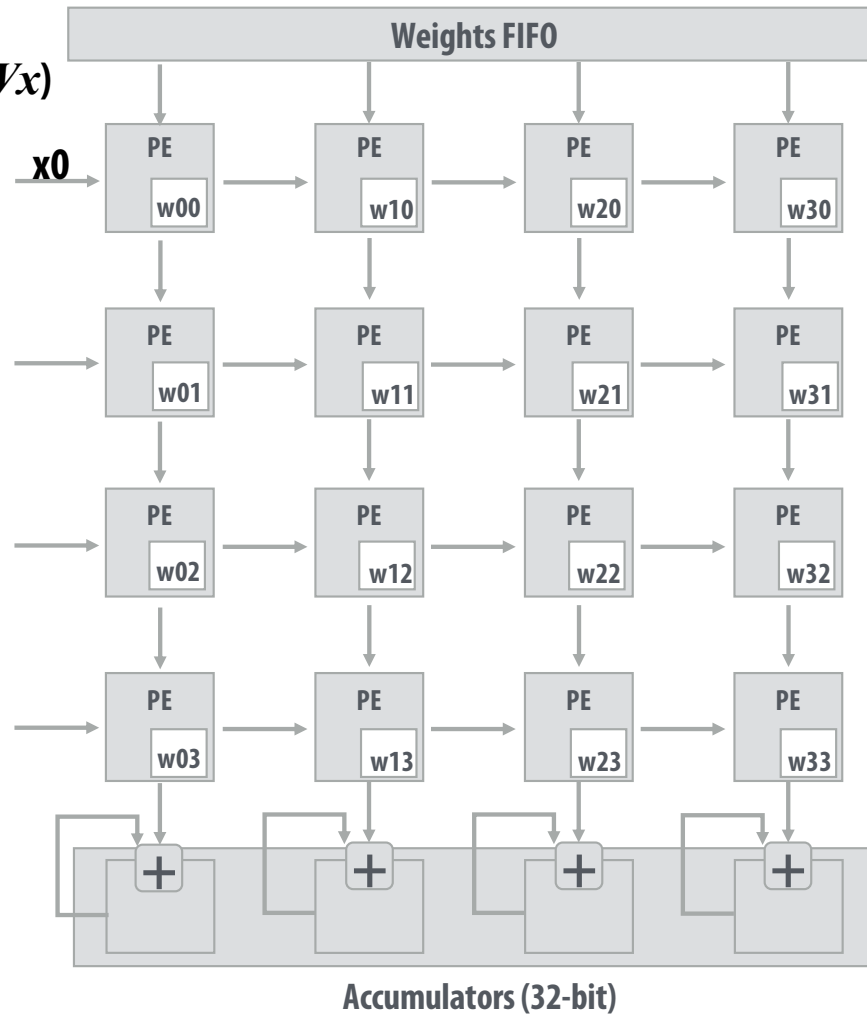
Systolic array

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



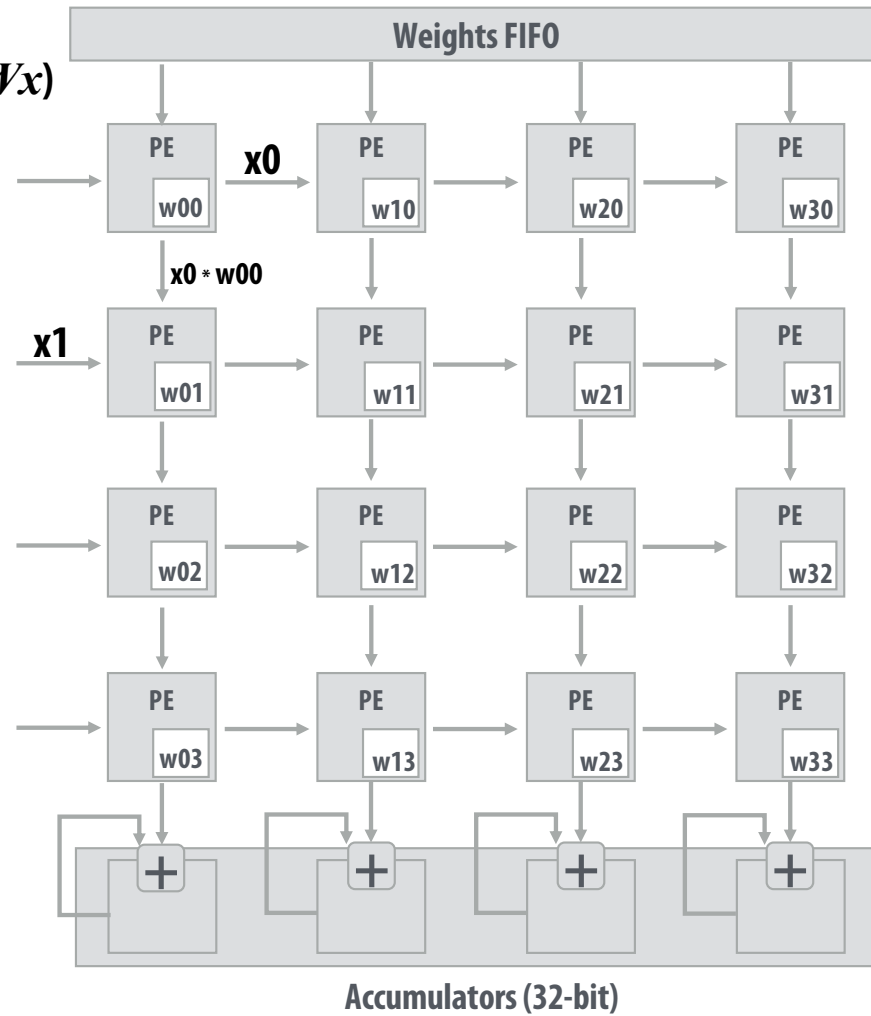
Systolic array

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



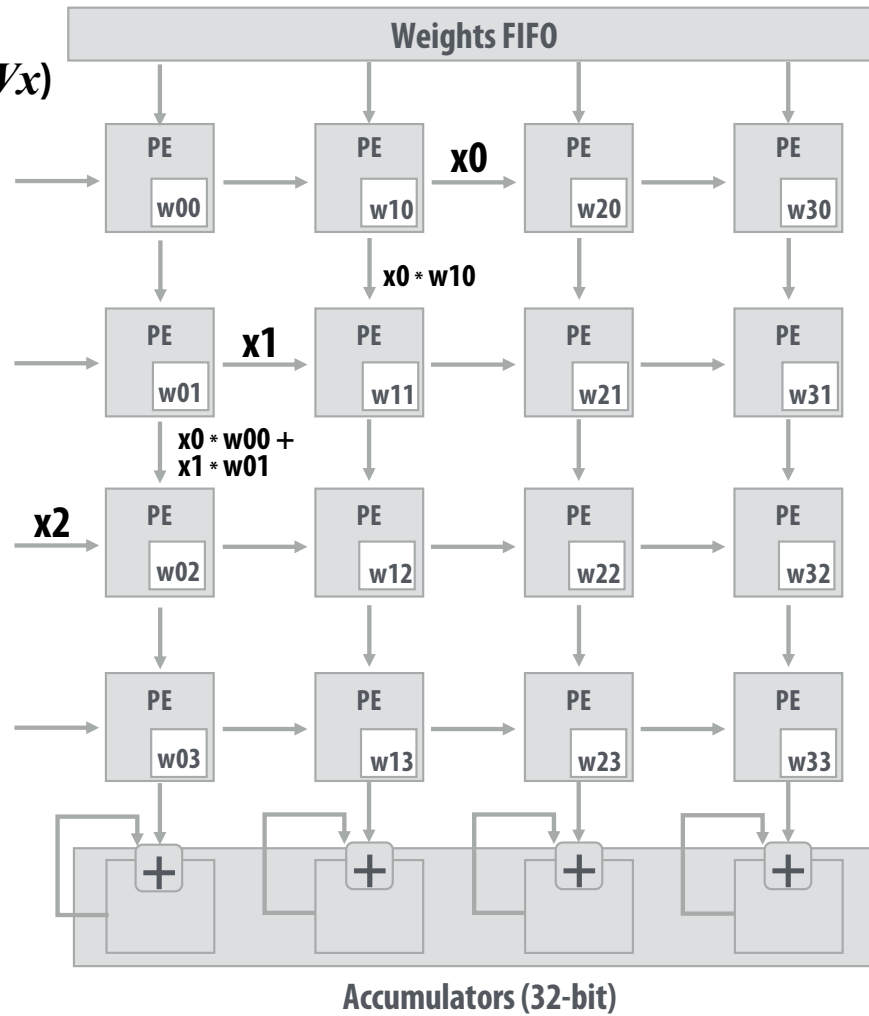
Systolic array

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



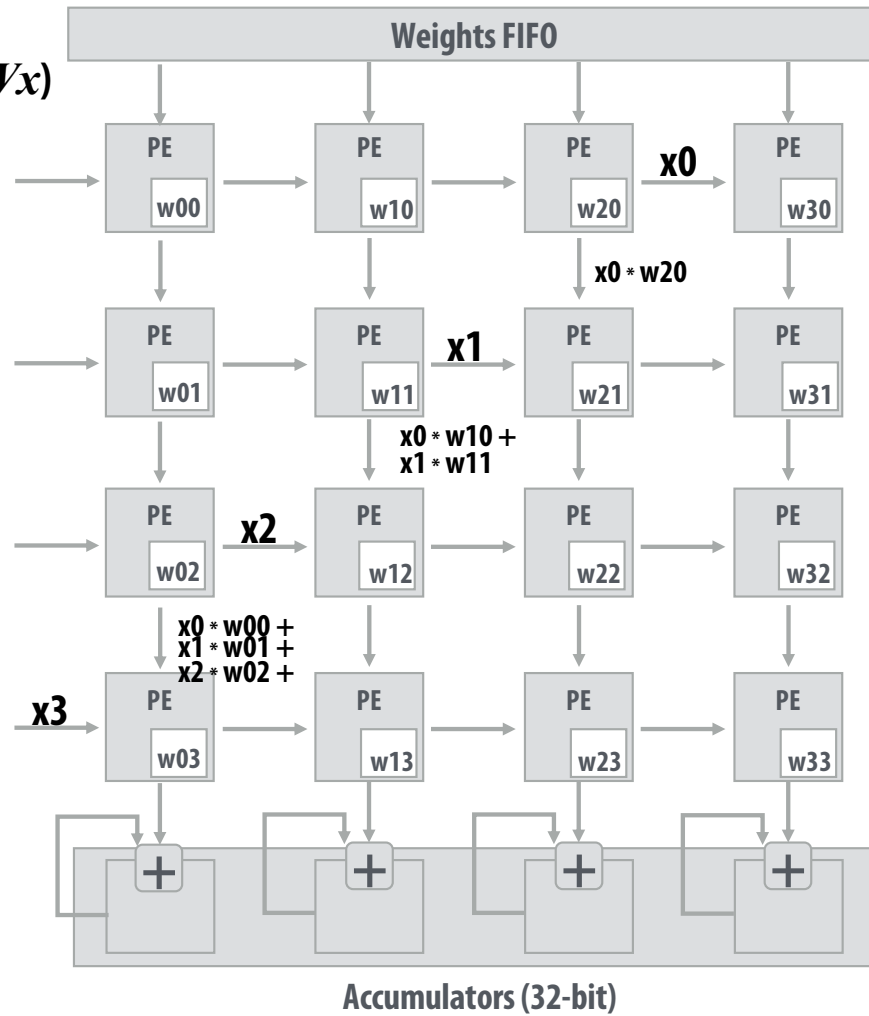
Systolic array

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



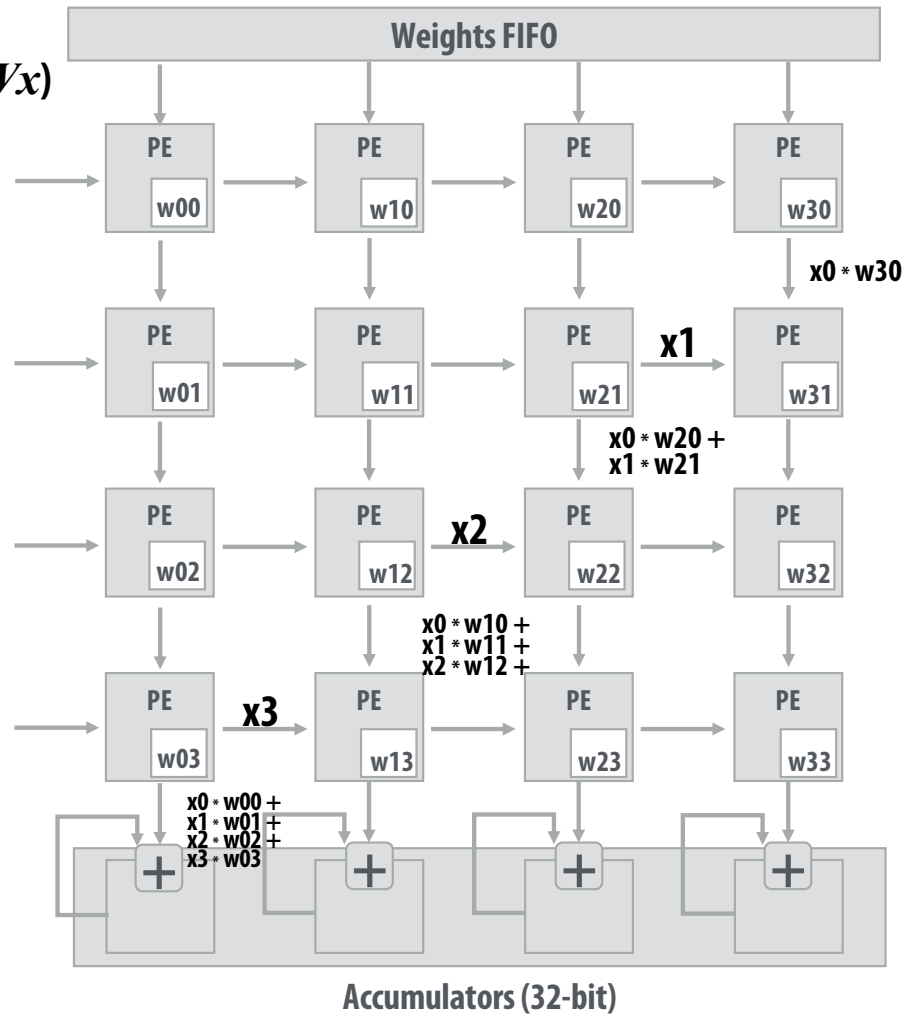
Systolic array

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



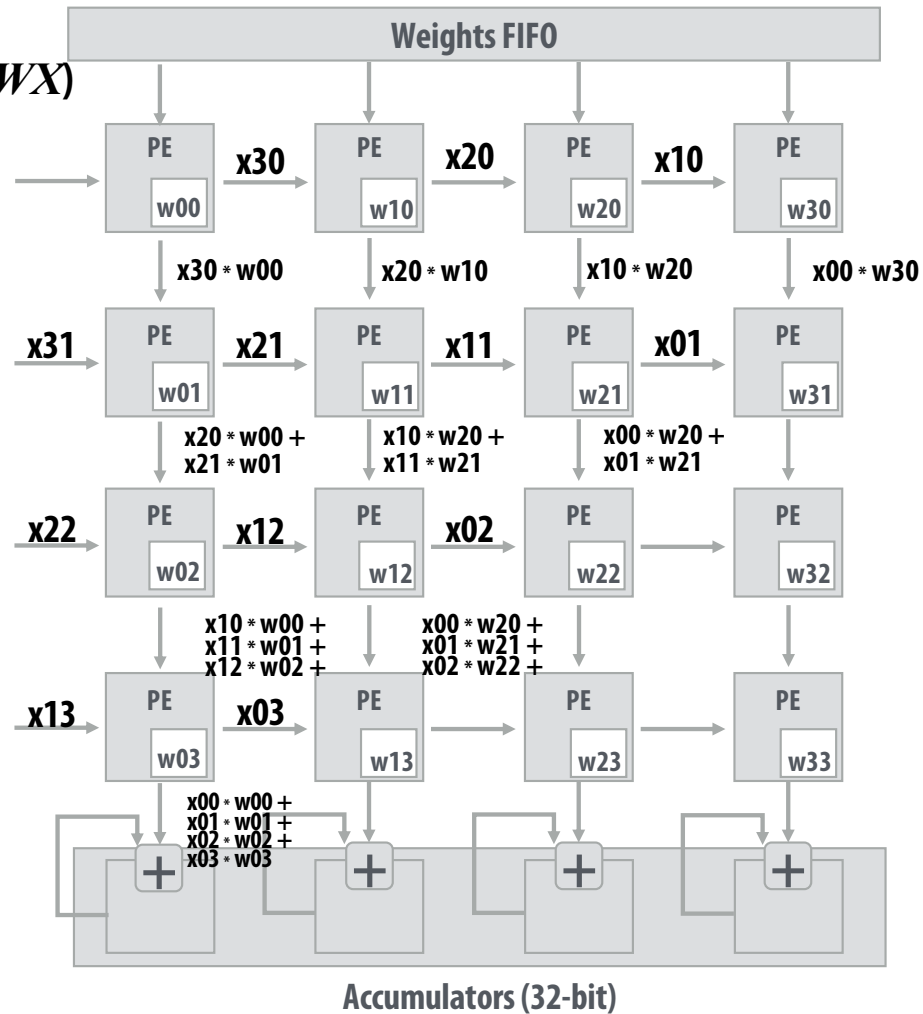
Systolic array

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



Systolic array

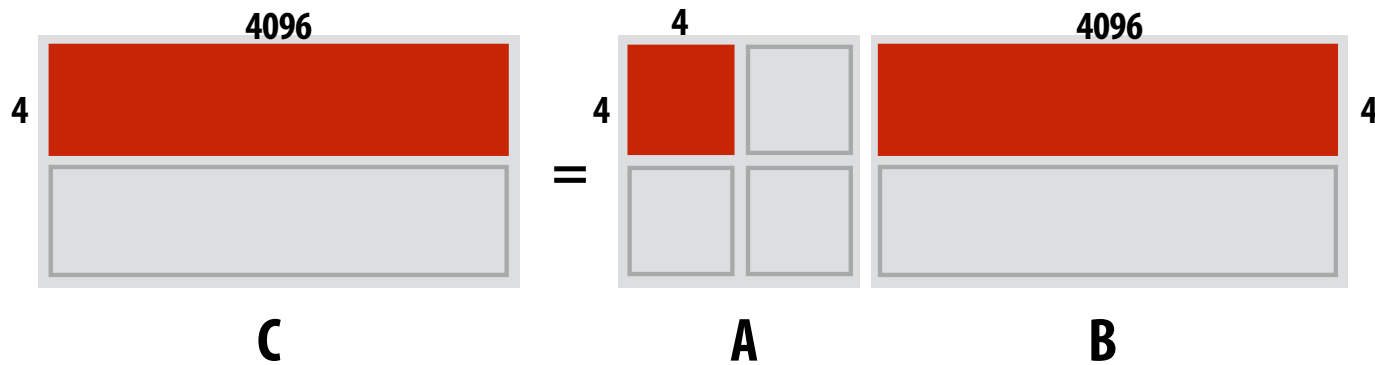
(matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

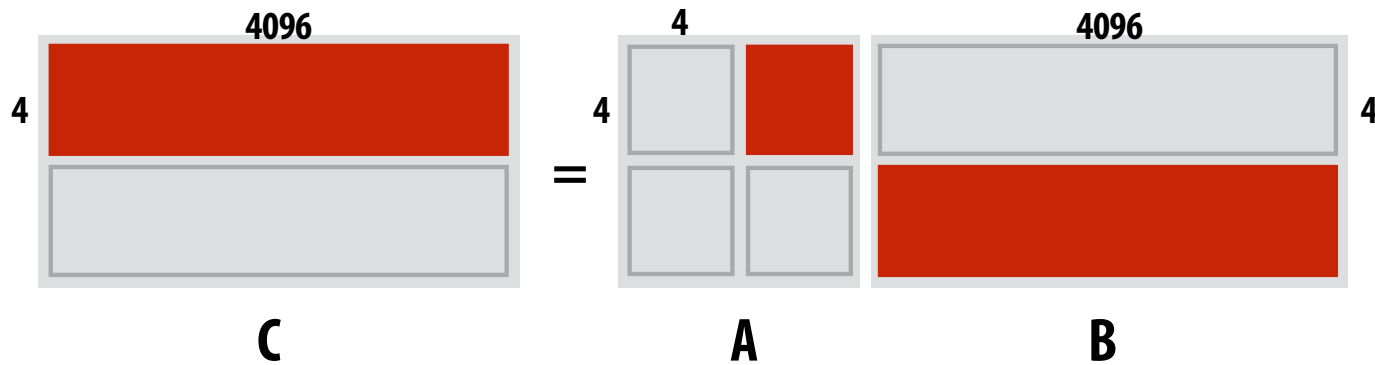
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



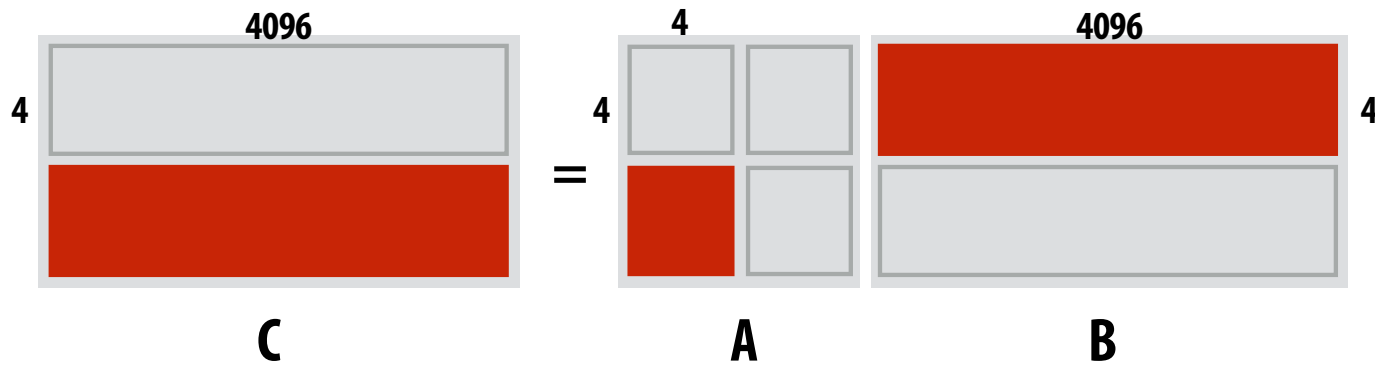
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



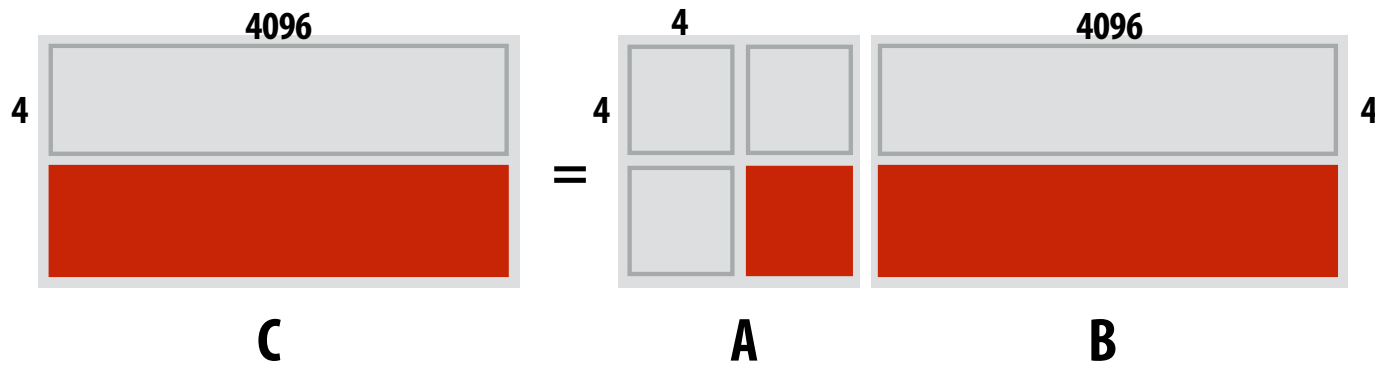
Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$

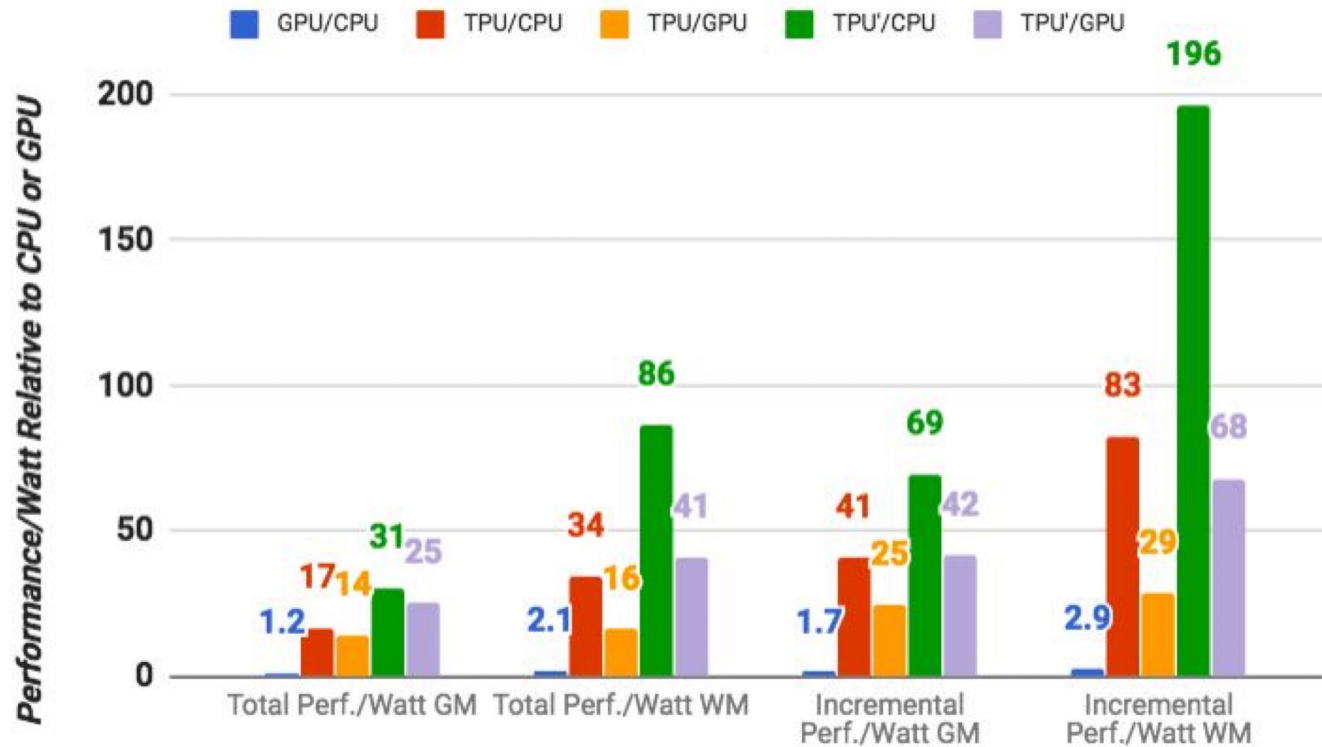


Building larger matrix-matrix multiplies

Example: $A = 8 \times 8$, $B = 8 \times 4096$, $C = 8 \times 4096$



TPU Performance/Watt



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

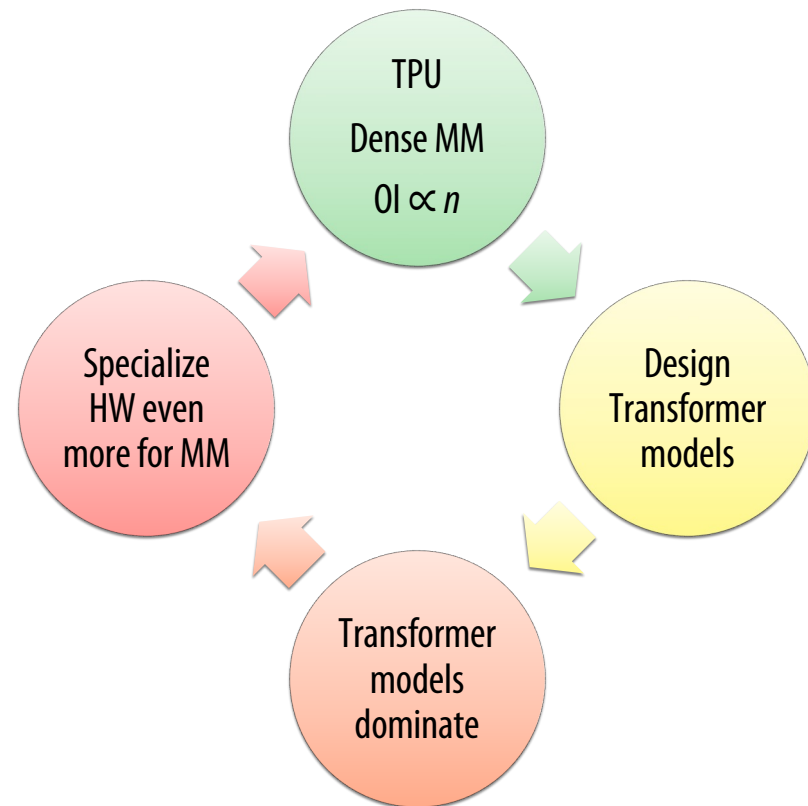
total = cost of host machine + CPU
incremental = only cost of TPU

Hardware Lottery



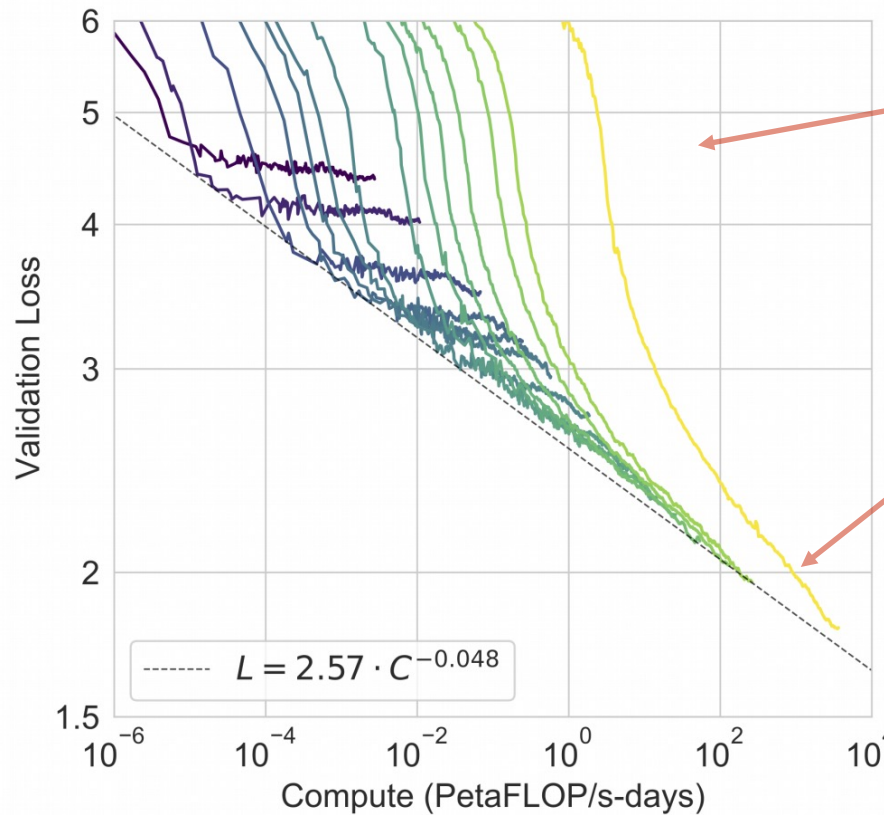
When a research idea wins because it is suited to the available software and hardware and not because the idea is universally superior to alternative research directions.

Sara Hooker



Scaling up (for training big models)

Example: GPT-3 language model



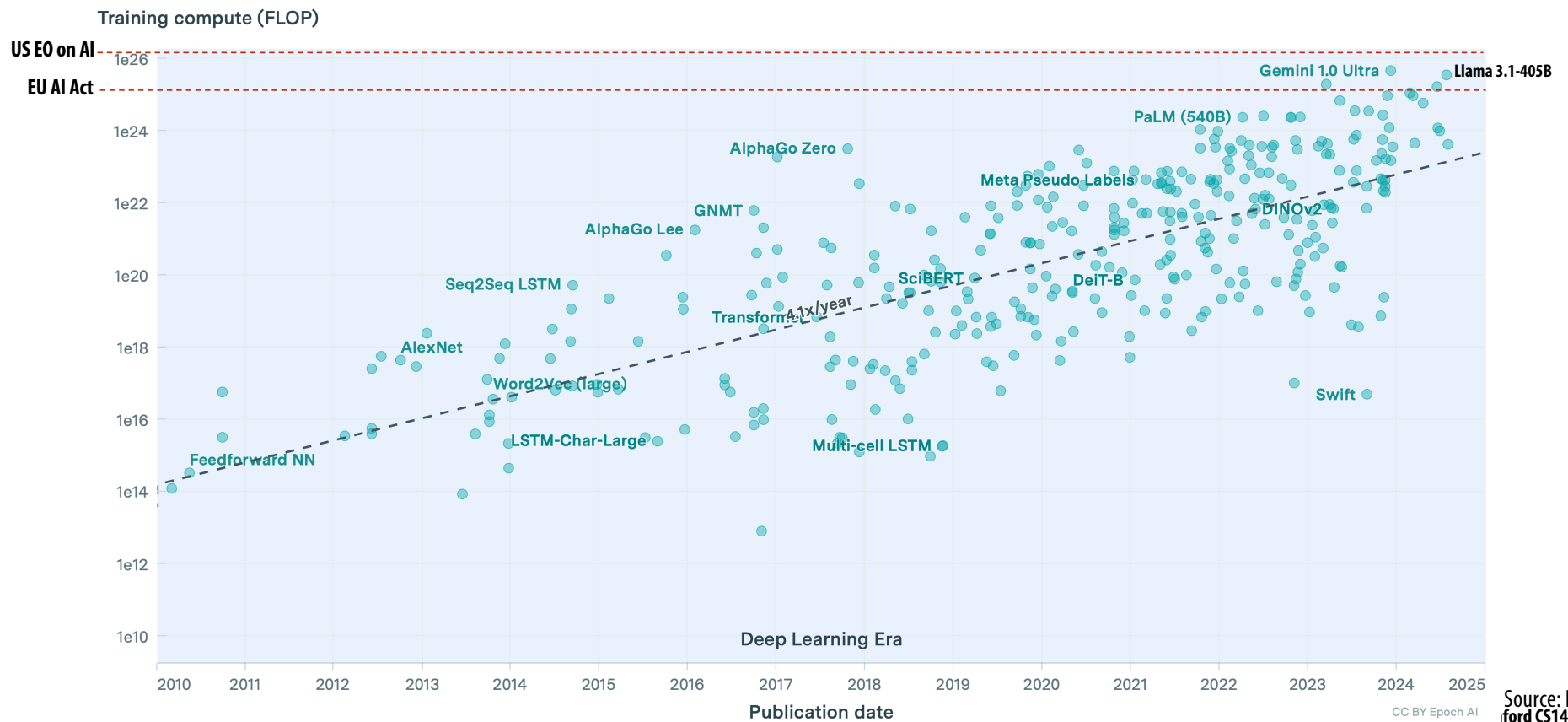
(Amount of training — note this is log scale)

Very big models +
More training
=
Better accuracy

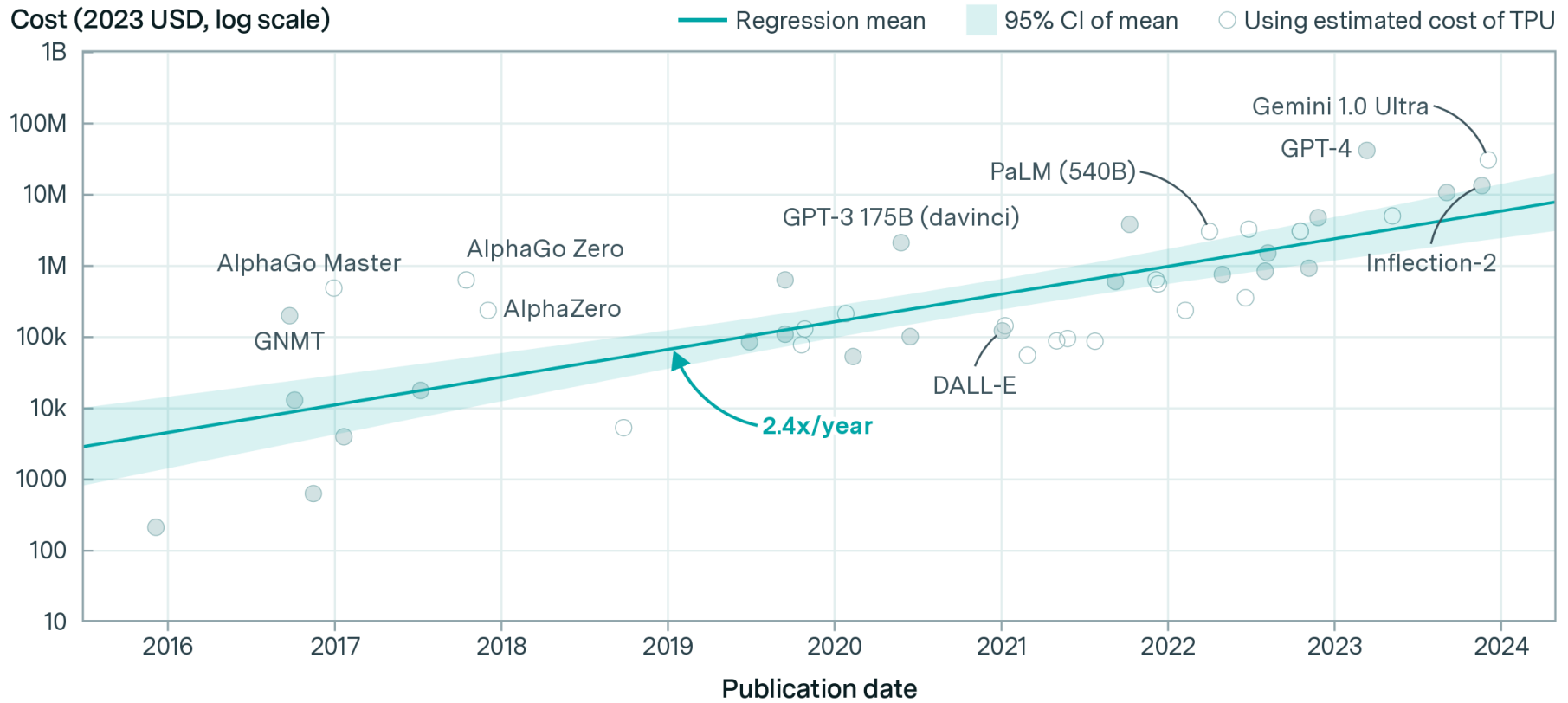
Power law effect:
exponentially more compute to take
constant step in accuracy

Large Model Training Compute

$$\text{Compute} = \text{Training time} \times \# \text{ of accelerator chips} \times \text{Peak FLOP/s} \times \text{Utilization rate}$$

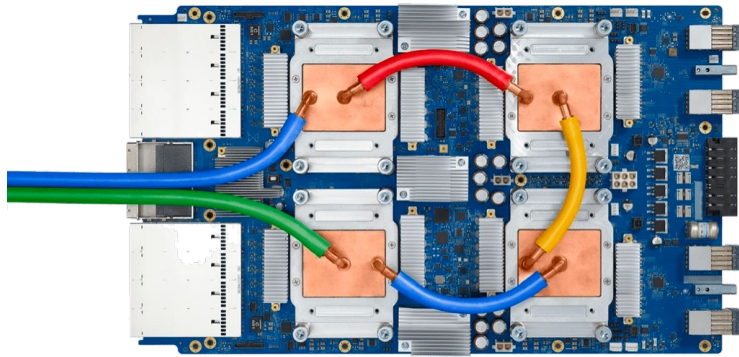


Hardware and Energy Costs of Training



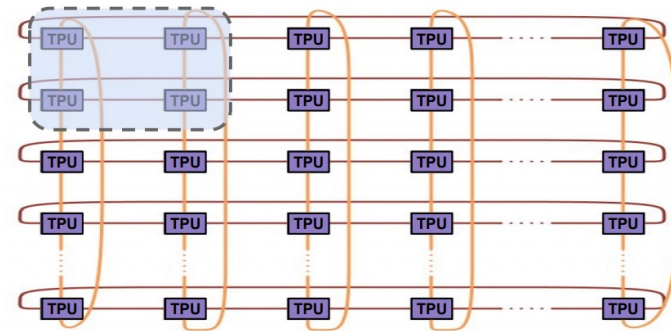
TPU v3 supercomputer

TPU v3 board
4 TPU3 chips

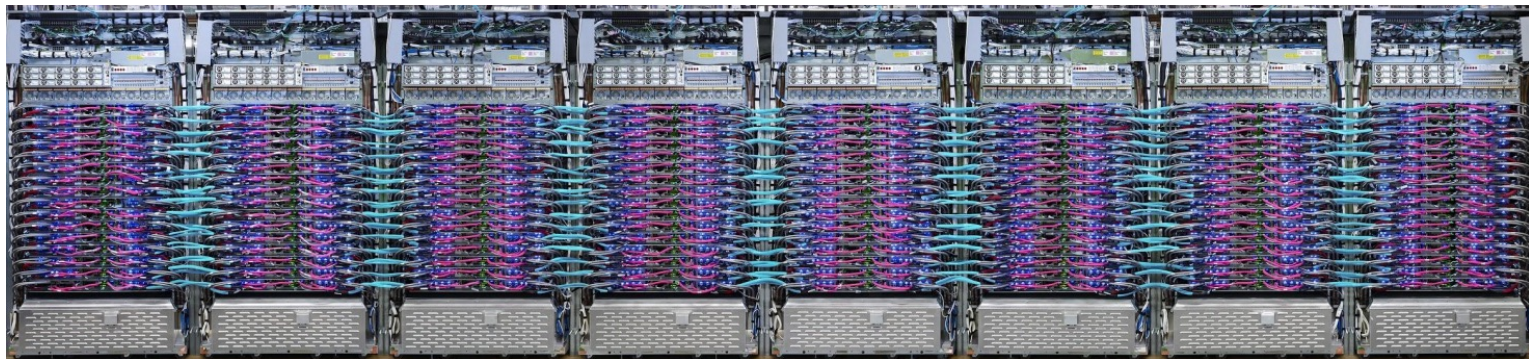


One TPU v3 board

TPUs connected by
2D Torus interconnect



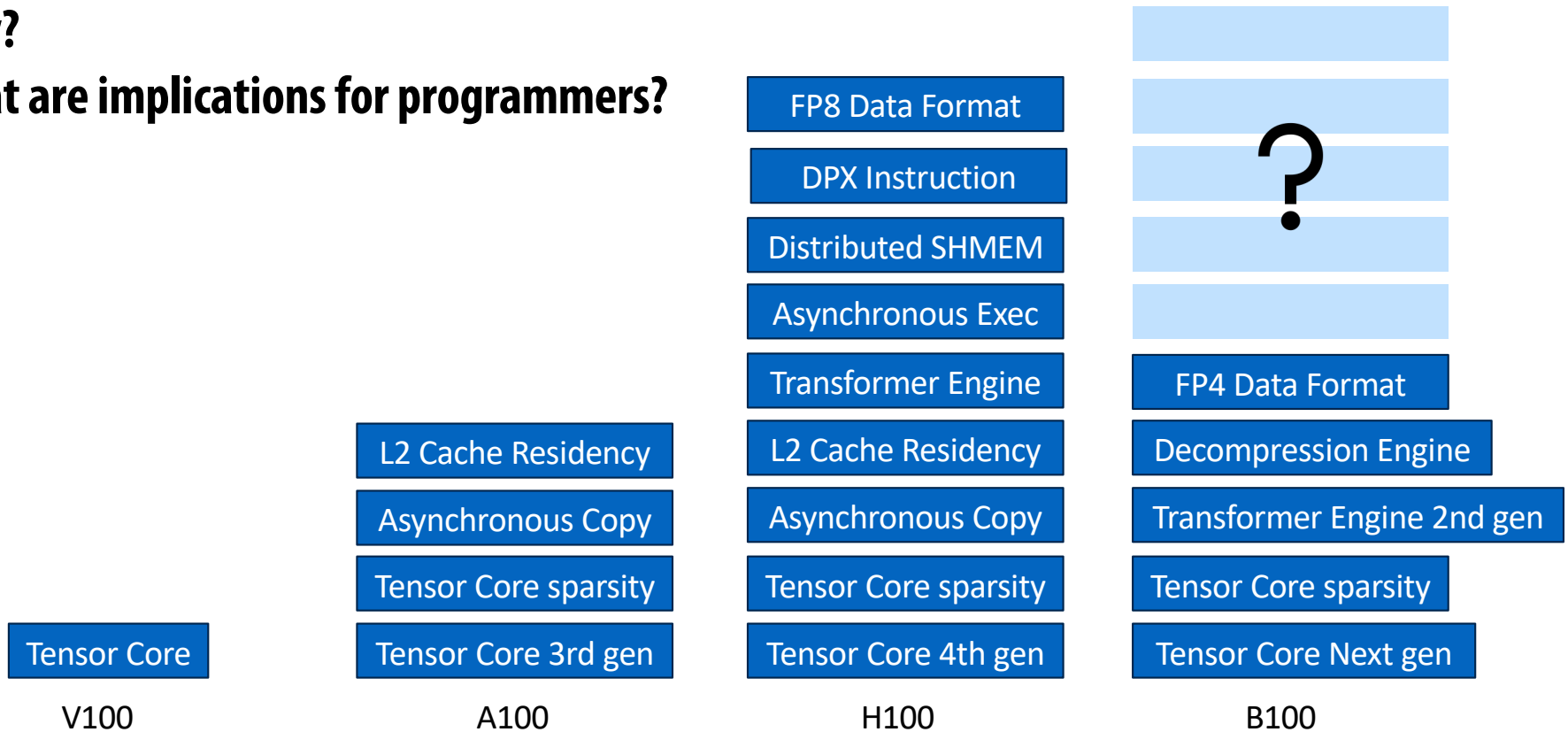
TPU supercomputer (1024 TPU v3 chips)



Nvidia Chips Becoming More Specialized

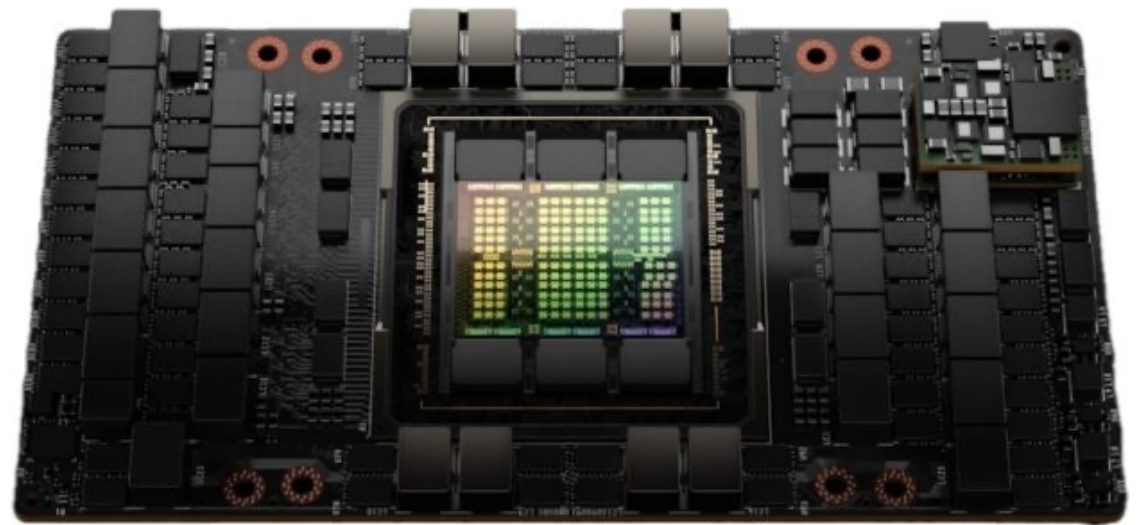
Why?

What are implications for programmers?

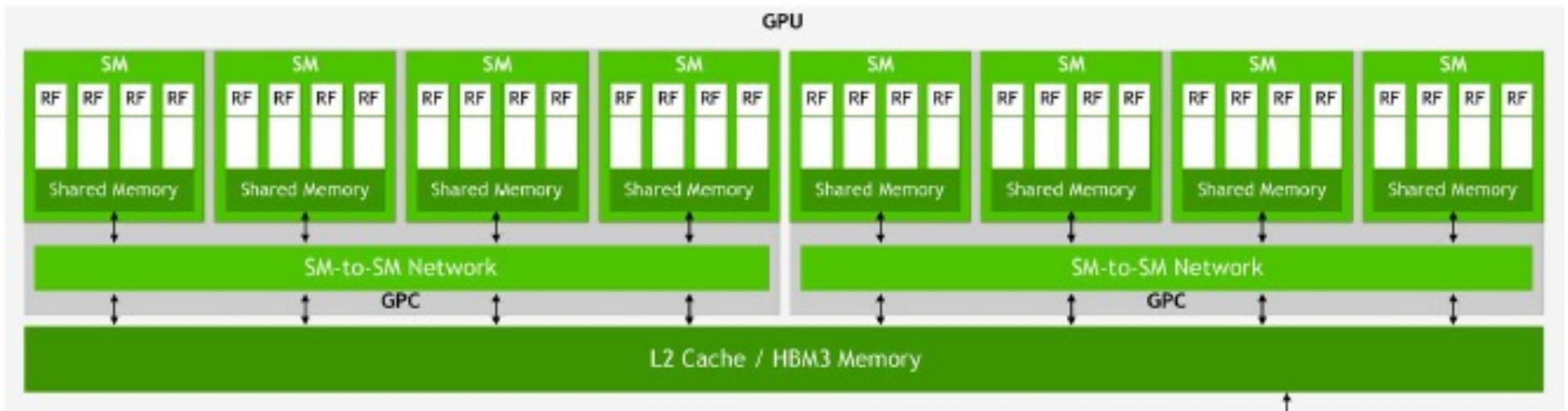


Nvidia H100 GPU (2022)

Fourth-generation Tensor Core
Tensor Memory Accelerator (TMA) unit
CUDA cluster capability
HBM3 with up to 80 GB
TSMC 4nm
80 Billion transistors



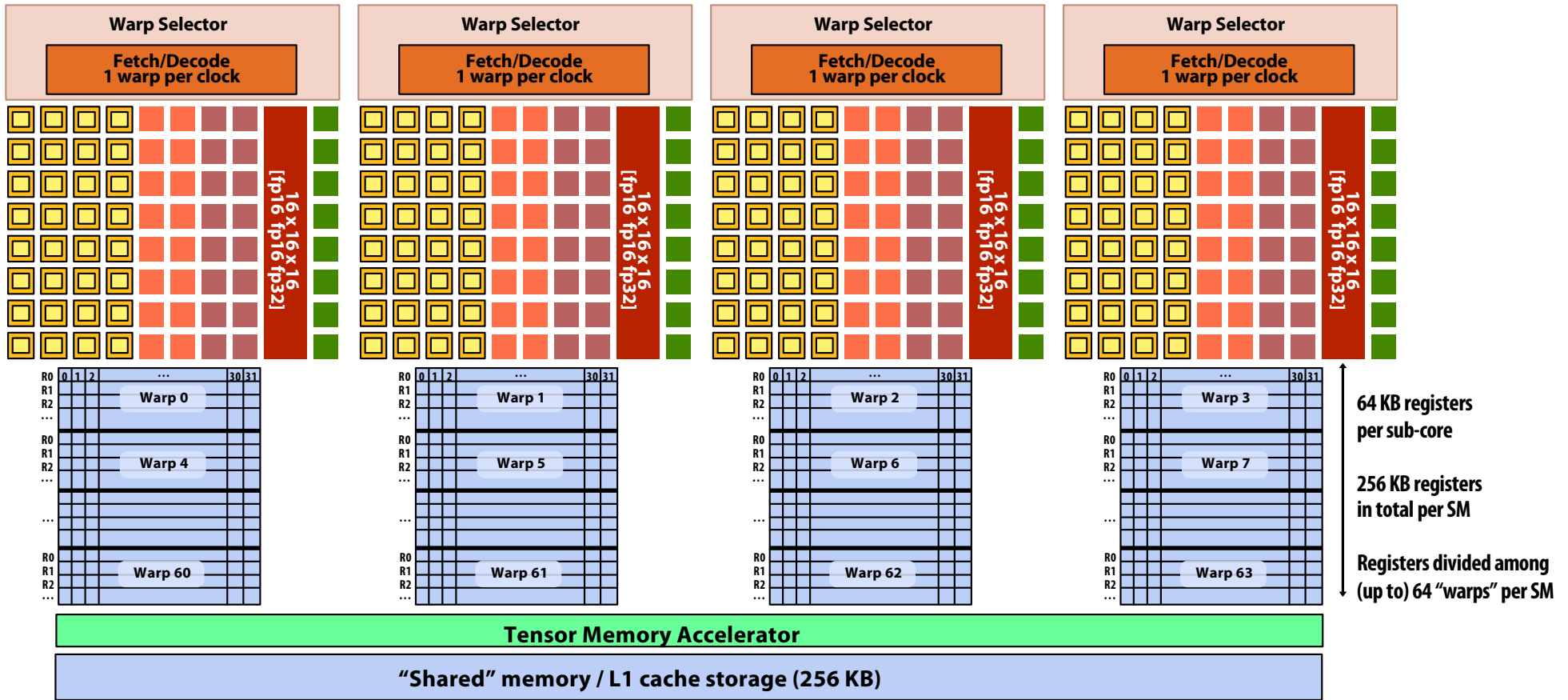
H100 CUDA, Compute and Memory Hierarchies




CUDA Hierarchy	Compute Hierarchy	Memory Hierarchy
Grid	GPU	80 GB HBM/ 50 MB L2
Cluster	CPC	256 KB shared memory per SM
Thread Block	SM	256 KB shared memory
Threads	SIMD Lanes	1 KB RF per thread, 64KB per SM partition

- Thread block cluster is a collective of up to 16 thread blocks
- Each thread block is guaranteed to execute on a separate SM and to run at the same time


H100 GPU Streaming Multi-processor (SM)





64 KB registers per sub-core
 256 KB registers in total per SM
 Registers divided among (up to) 64 "warps" per SM

 = SIMD fp32 functional unit, control shared across 16 units (32 x MUL-ADD per clock *)

 = SIMD int functional unit, control shared across 16 units (16 x MUL/ADD per clock **)

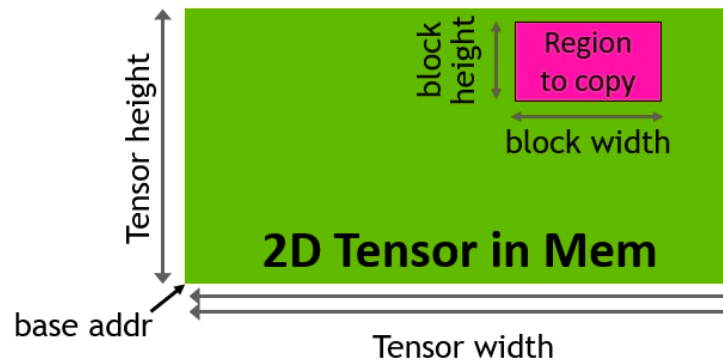
 = SIMD fp64 functional unit, control shared across 8 units (16 x MUL/ADD per clock **)

 = Tensor core unit
 = Load/store unit

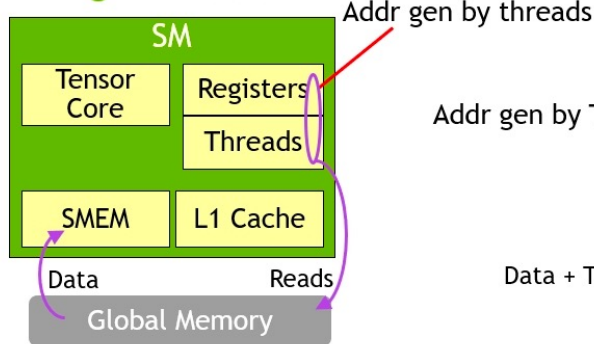
* one 32-wide SIMD operation every clock
 ** one 32-wide SIMD operation every 2 clocks

Tensor Memory Accelerator

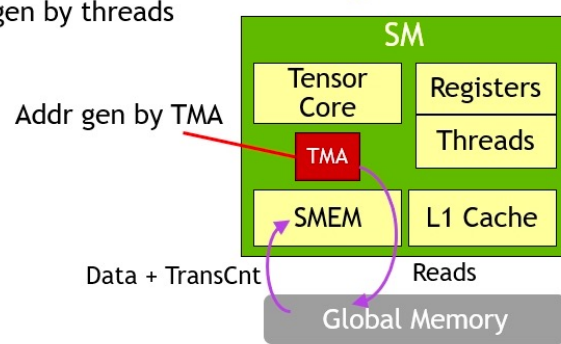
Copy Descriptor



A100 Using LDGSTS instr



H100 Using TMA Unit



Special purpose instructions for efficient data movement

Asynchronously load/store a region of a tensor from global to shared memory

Copy descriptor describes region

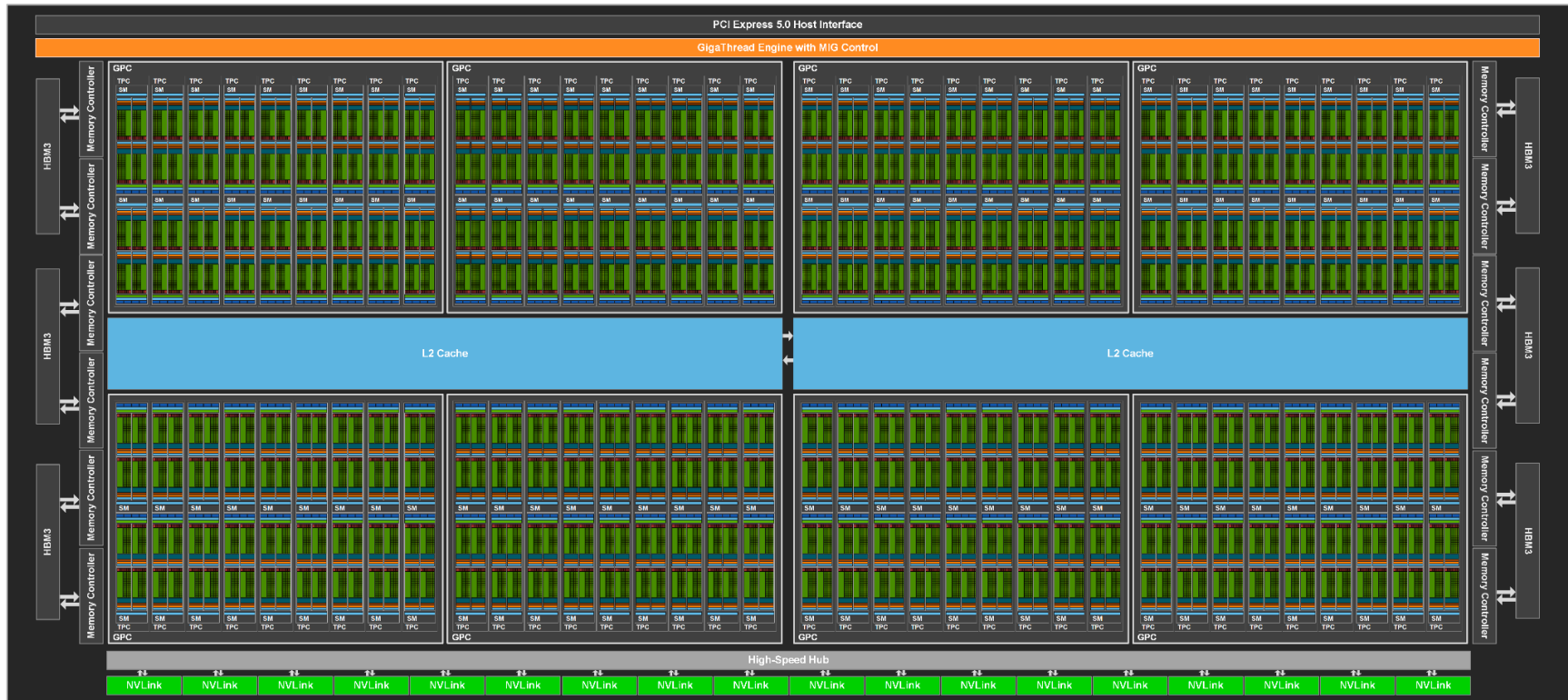
Single thread issue TMA operation

cuda : memcpy_async

Signal barrier when copy is complete

Hardware address generation and data movement

The Whole H100



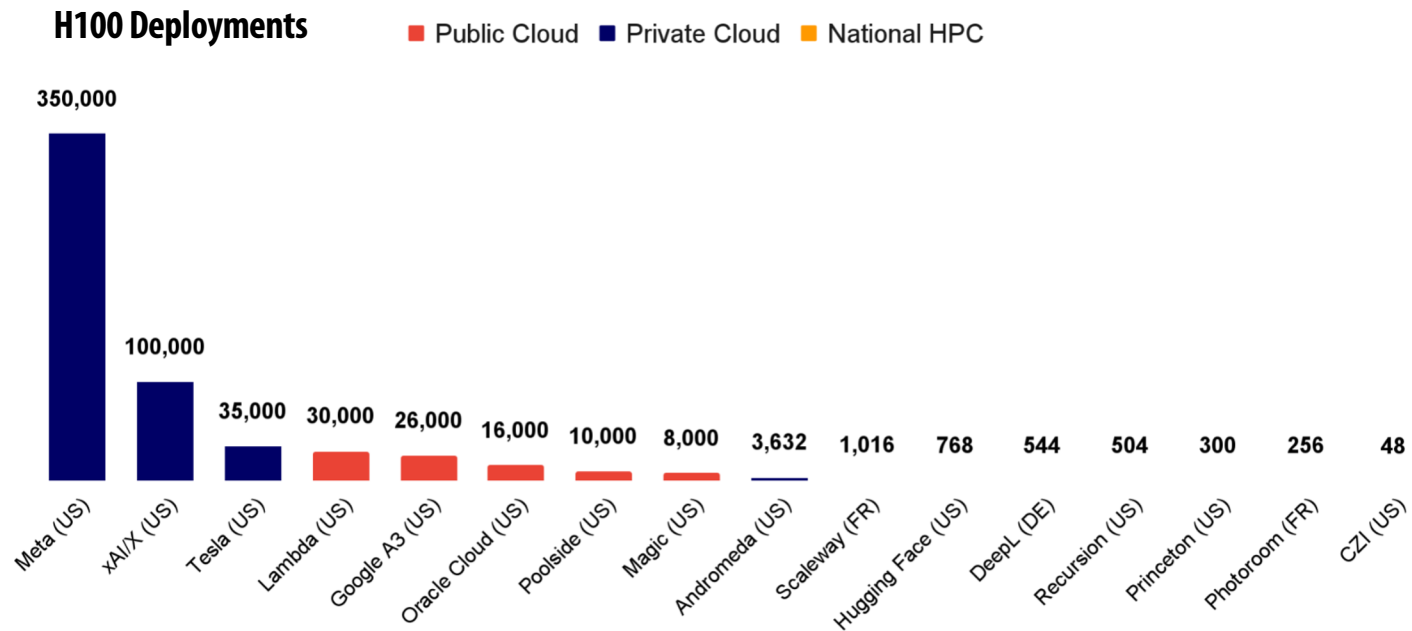
144 SMs

Tensor cores (systolic array MMA): 989 TFLOPS (fp16) ⇒ ~90% of TFLOPS

SIMD: 134 TFLOPS (fp16), 67 TFLOPS (fp32) ⇒ ~10% of TFLOPS

GPU Kernels are Important

- 2024 GPU market is enormous \Rightarrow NVIDIA 2024 quarterly revenue of $> \$30B$
- 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 $\sim 70\%$ on A100s to $\sim 35\%$ on H100s. Took 2 years to come back up to $\sim 65\%$ with FlashAttention-3
- Poor kernels can cost billions of dollars worth of compute

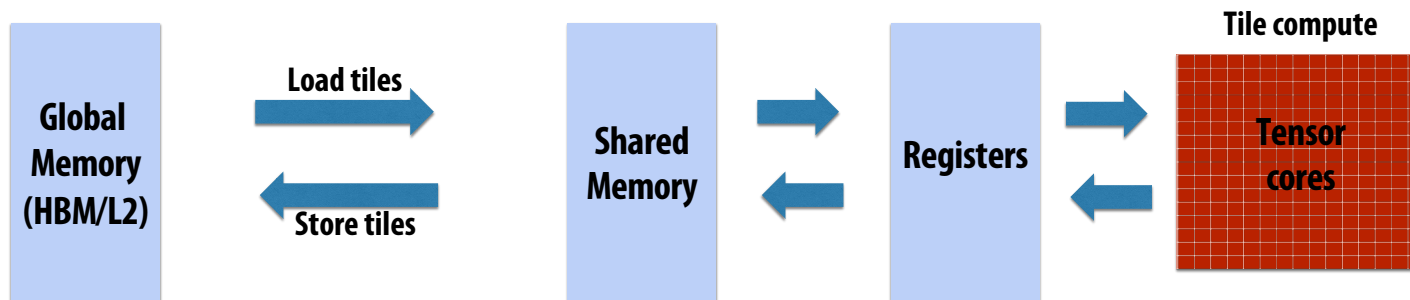


Extracting Peak Performance from the H100

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

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

A tile processing pipeline



ThunderKittens

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

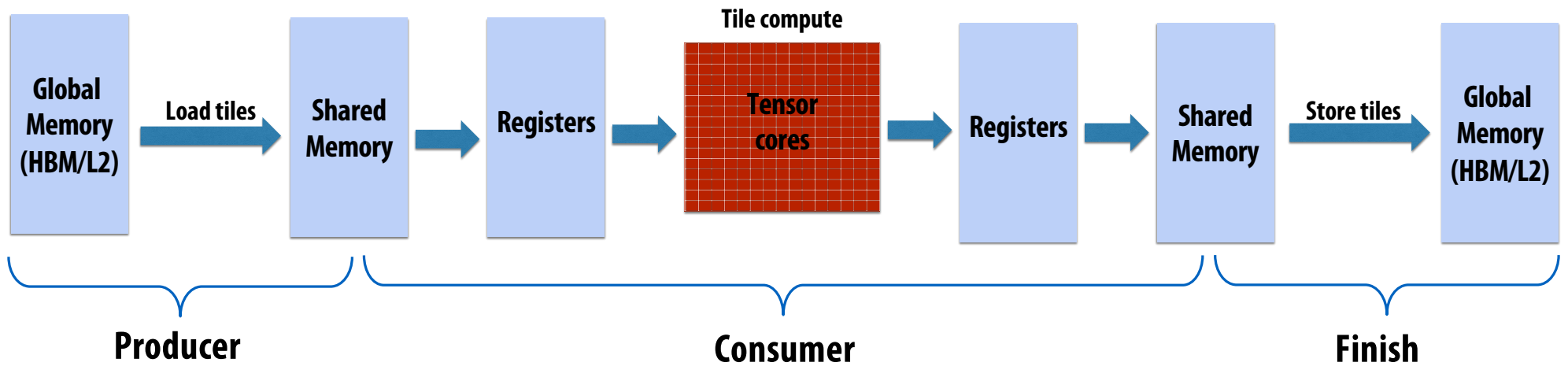
- Initializers -- zero out a shared vector, for example.
- Unary ops, like exp
- Binary ops, like mul
- Row / column ops, like a row_sum



A Simple Embedded DSL for AI kernels

Ben Spector et. al.

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

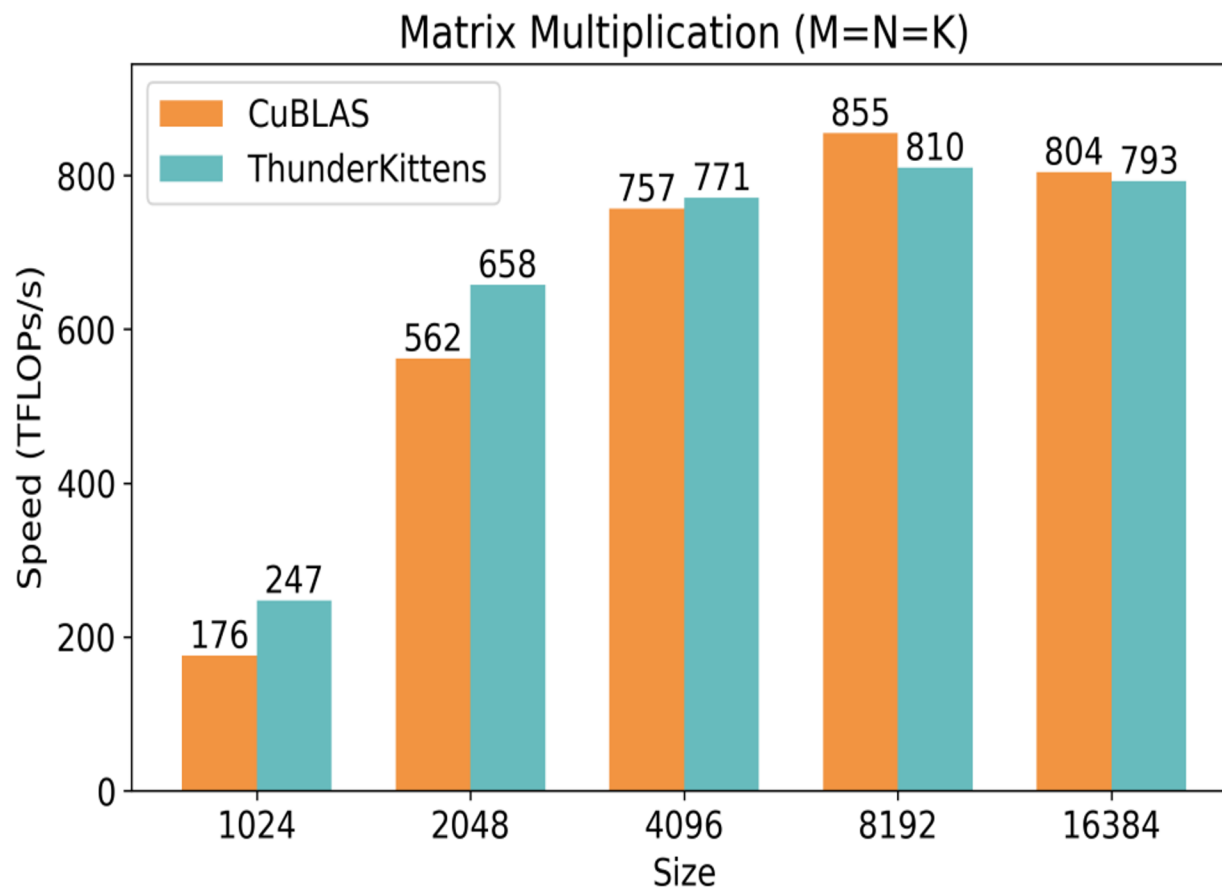
```
struct matmul_template {
    using layout = matmul_layout;
    static constexpr int NUM_CONSUMER_WARPS=8, INPUT_PIPE_STAGES=4; // 8 active consumer warps, 4 active producer warps (default), and a 4-stage pipeline
    static constexpr int PRODUCER_BARRIER_ARRIVALS=1, CONSUMER_BARRIER_ARRIVALS=2; // 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 semaphore 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});
    }
};
};
```

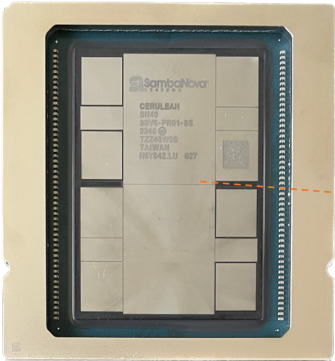
TK Matmul Performance



Can we have efficiency and a simpler programming model?

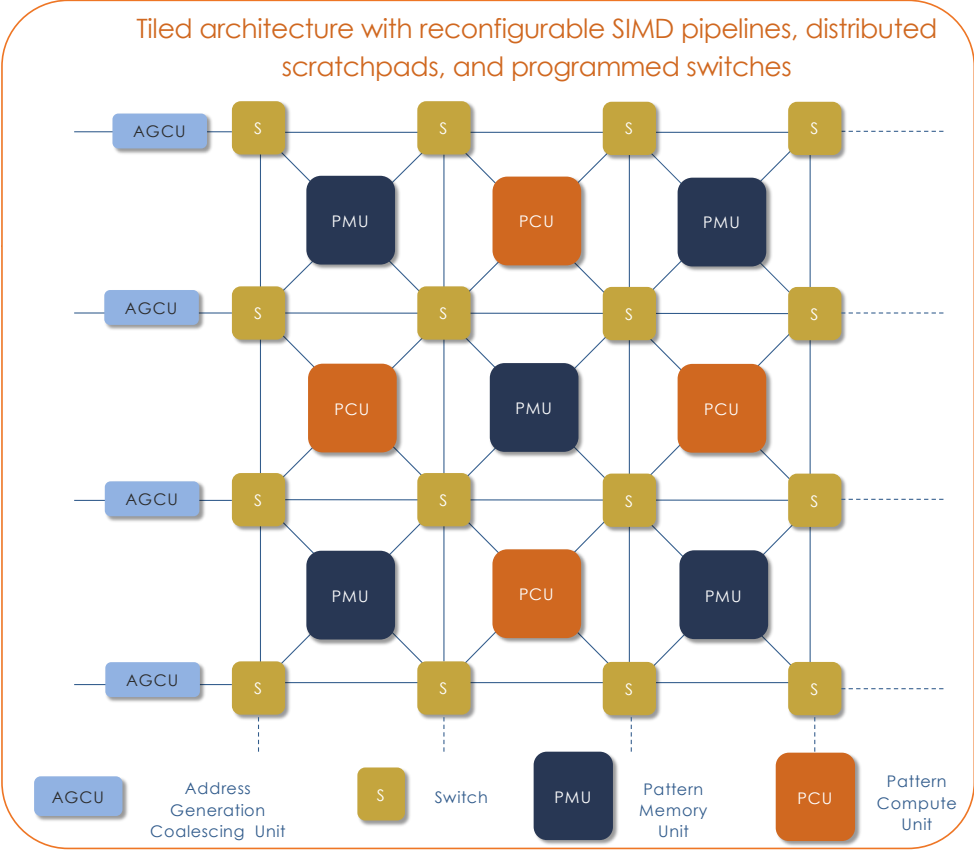
(Hint: Take a data-centric view)

Reconfigurable Dataflow



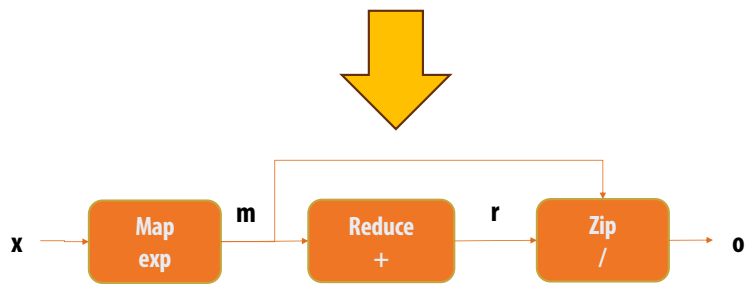
SambaNova SN40L RDU

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

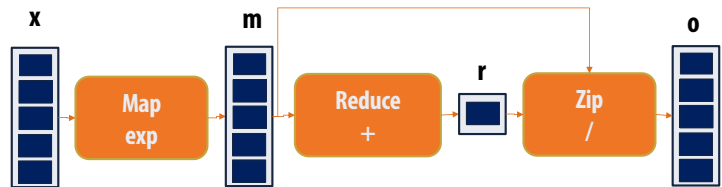


Dataflow Programming with Data Parallel Patterns

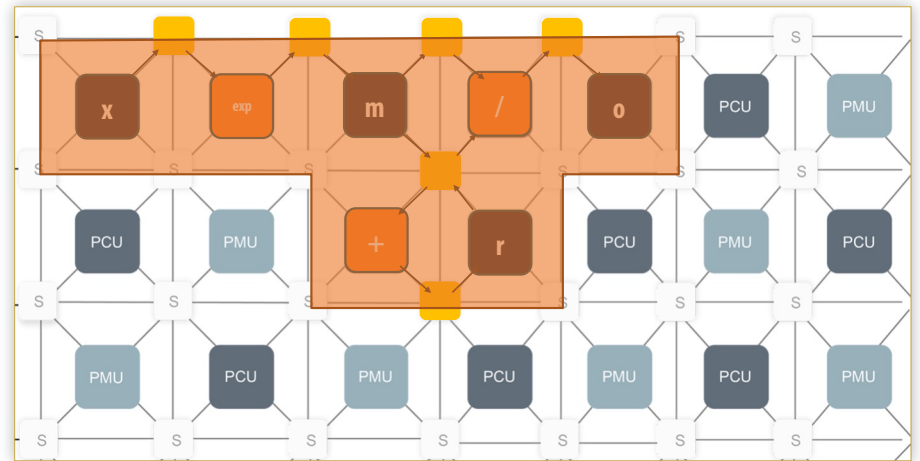
SOFTMAX:
$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$



Tiling
Parallelization
Parallel Pattern fusion



Place & Route
Codegen



- **PCU:** systolic and streaming compute
- **PMU:** High address generation flexibility and bandwidth
- **ICN:** High on-chip interconnect flexibility and bandwidth

- **Composable Compute Primitives:** MM, Map, Zip, Reduce, Gather, Scatter ...
- **Flexible scheduling in space and time**

Metapipelining

Hierarchical coarse-grained pipeline: A “pipeline of pipelines”

- Exploits nested parallelism

Convert parallel pattern (loop) to a streaming pipeline

- Insert pipe stages in the body of the loop
- Overlap execution of loop iterations

Intermediate data between stages stored in double buffers

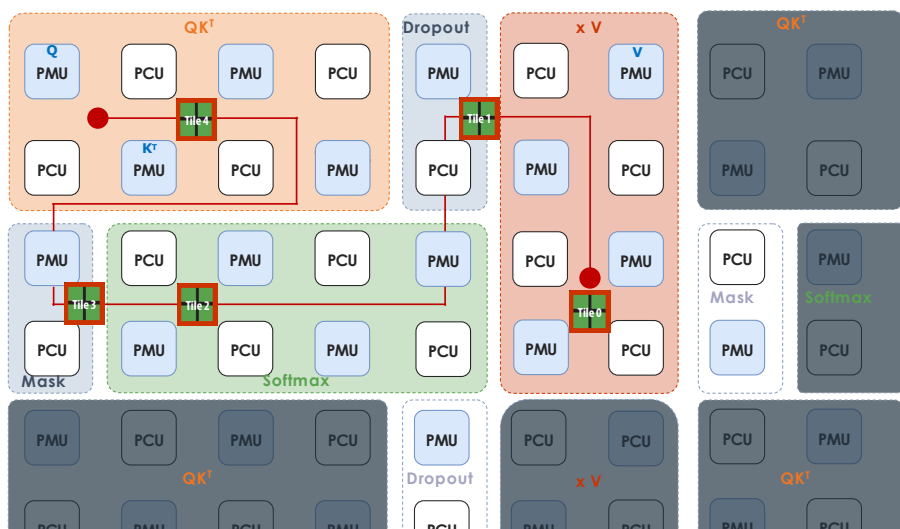
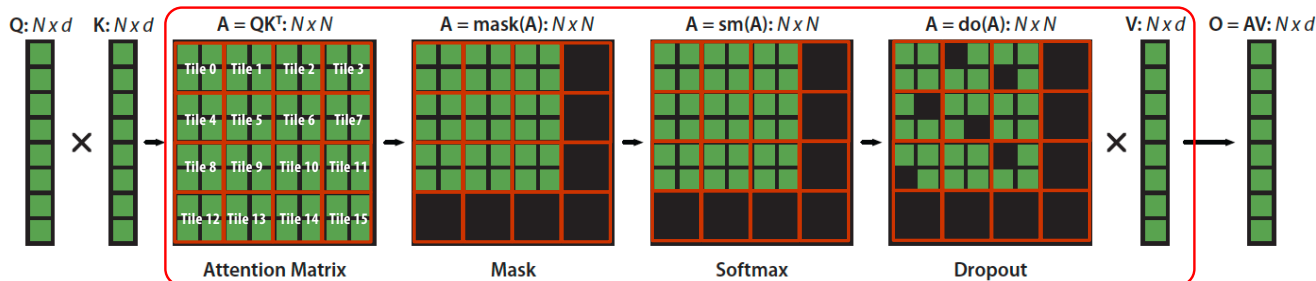
- Handles imbalanced stages with varying execution times

Tiling and fusion

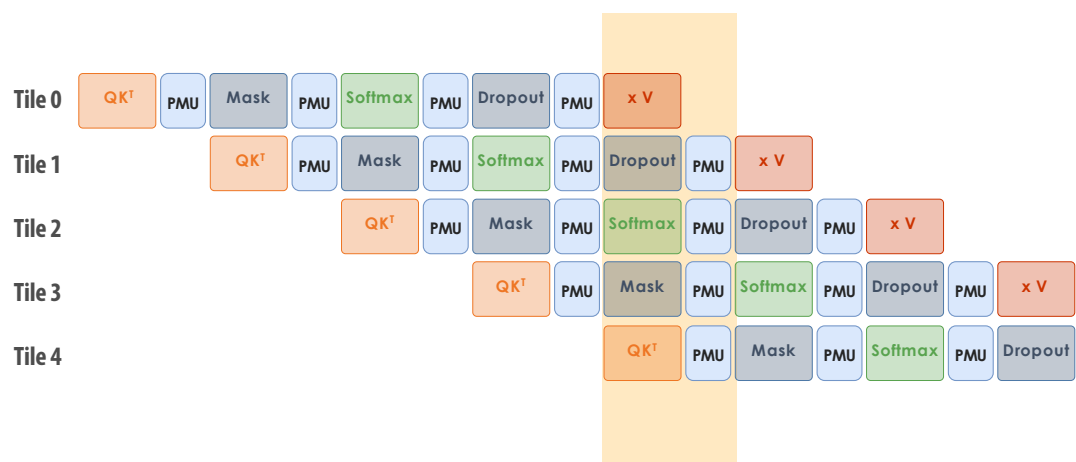
- Optimized for tiling
- Metapipelining can work when fusion does not
- Buffers can be used to change access pattern (e.g. transpose data)

FlashAttention Metapipeline

FlashAttention



Dataflow execution with token control \Rightarrow no overhead synchronization



MetaPipeline = Streaming Dataflow

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);

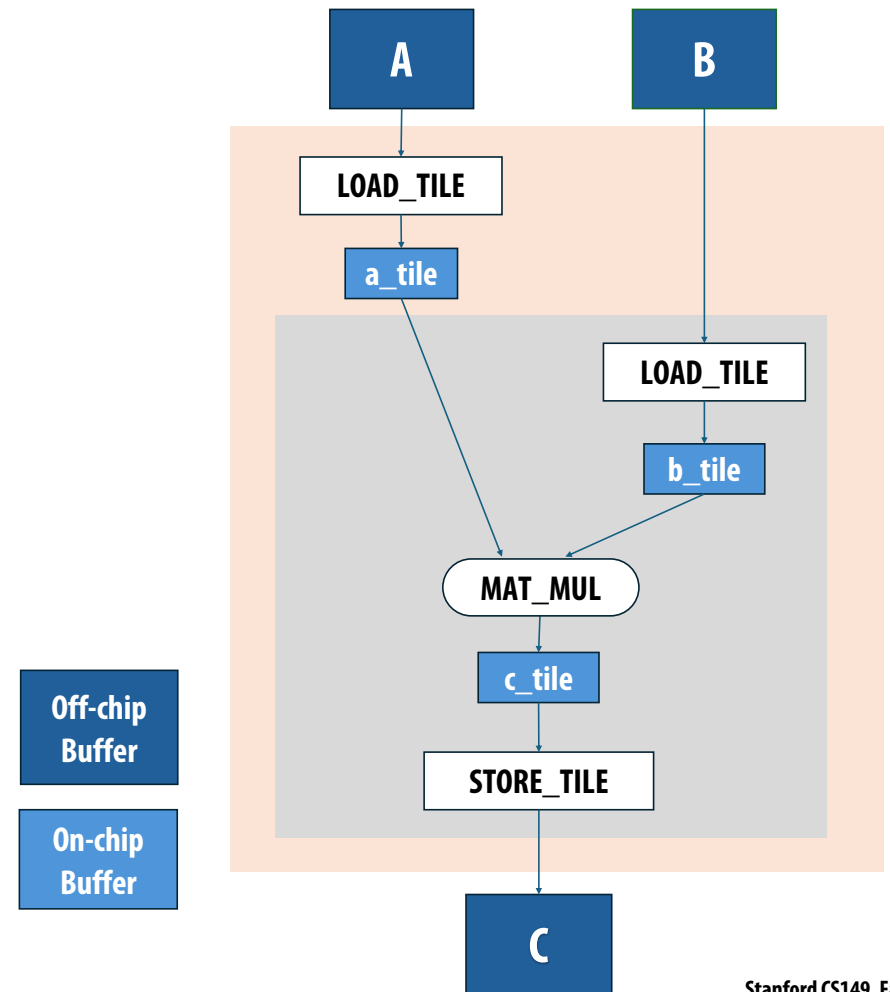
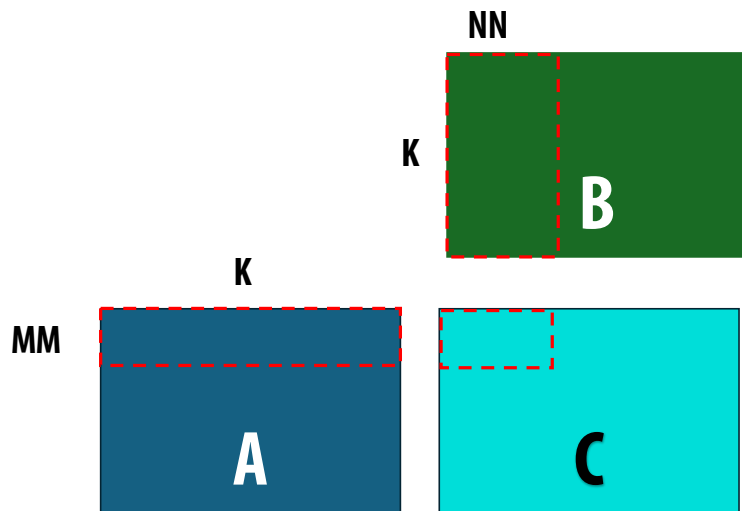
auto MM = 256; // Tile size along M, assumes to evenly divide M
auto NN = 64; // Tile size along N, assumes to evenly divide N

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);
        auto c = MAT_MUL(a_tile, b_tile);
        auto c_tile = BUFFER(c);
        STORE_TILE(C, c_tile);
    });
});
```

Matmul Metapipe

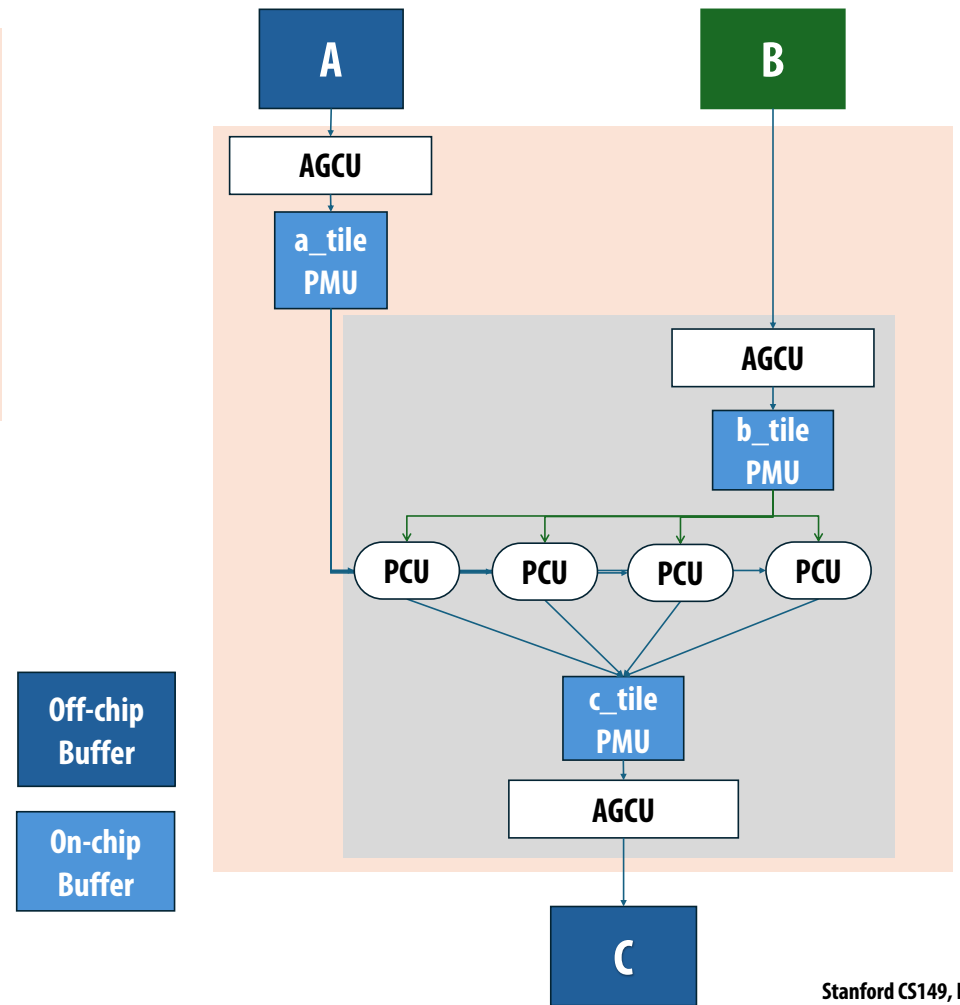
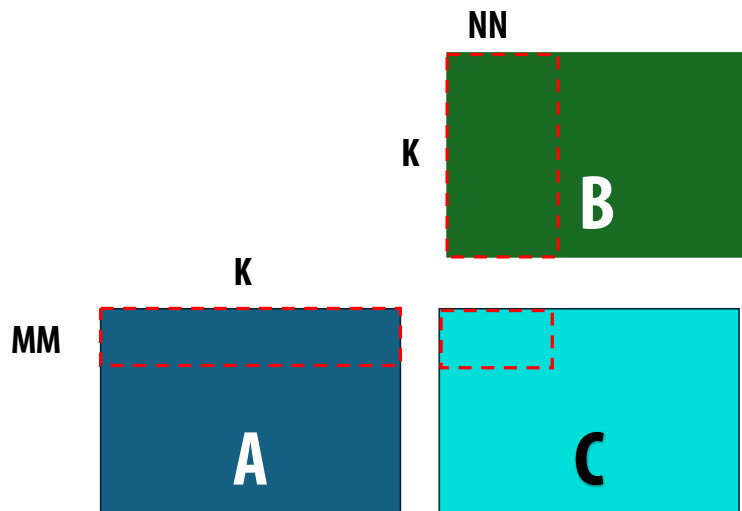
```
METAPIPE(M, MM) {  
  a_tile = LOAD_TILE(A, a_tile.shape)  
  METAPIPE(N, NN) {  
    b_tile = LOAD_TILE(B, b_tile.shape)  
    c = MAT_MUL(a_tile, b_tile)  
    c_tile = BUFFER(c)  
    STORE_TILE(c_tile)  
  }  
}
```



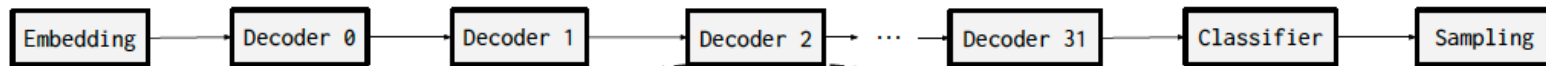
Matmul Metapipe Mapping

```

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)
    c_tile = BUFFER(c)
    STORE_TILE(c_tile)
  }
}
    
```



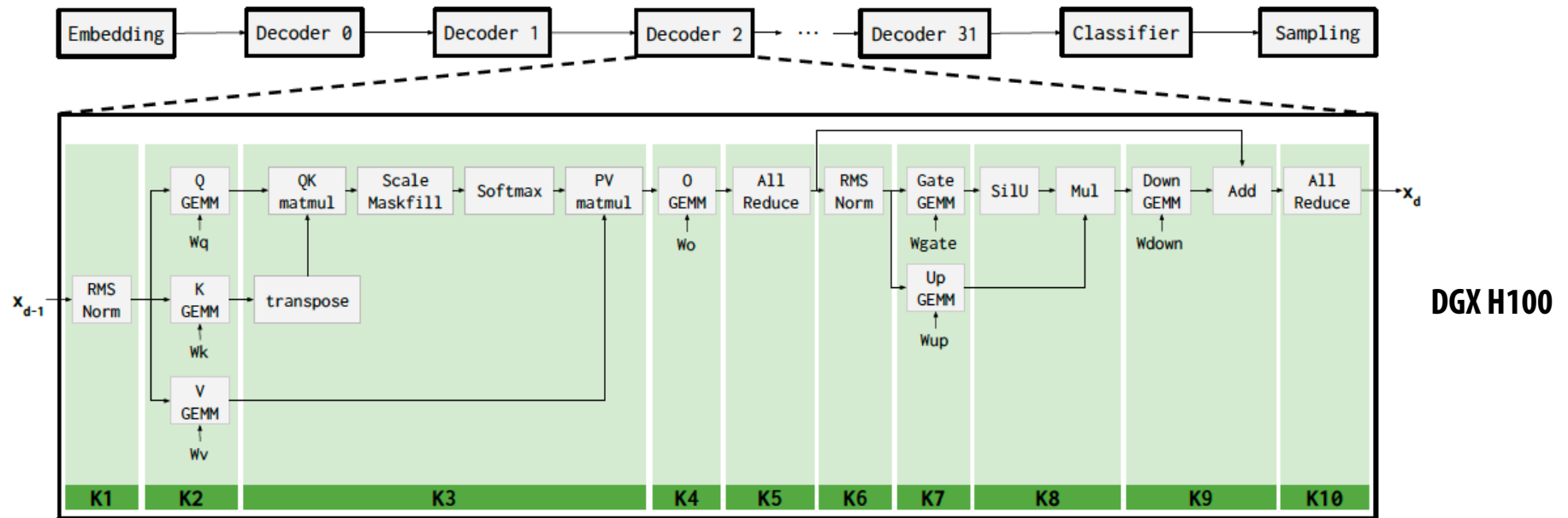
Llama 3.1 8B



Llama3.1-8B with 32 decoder layers

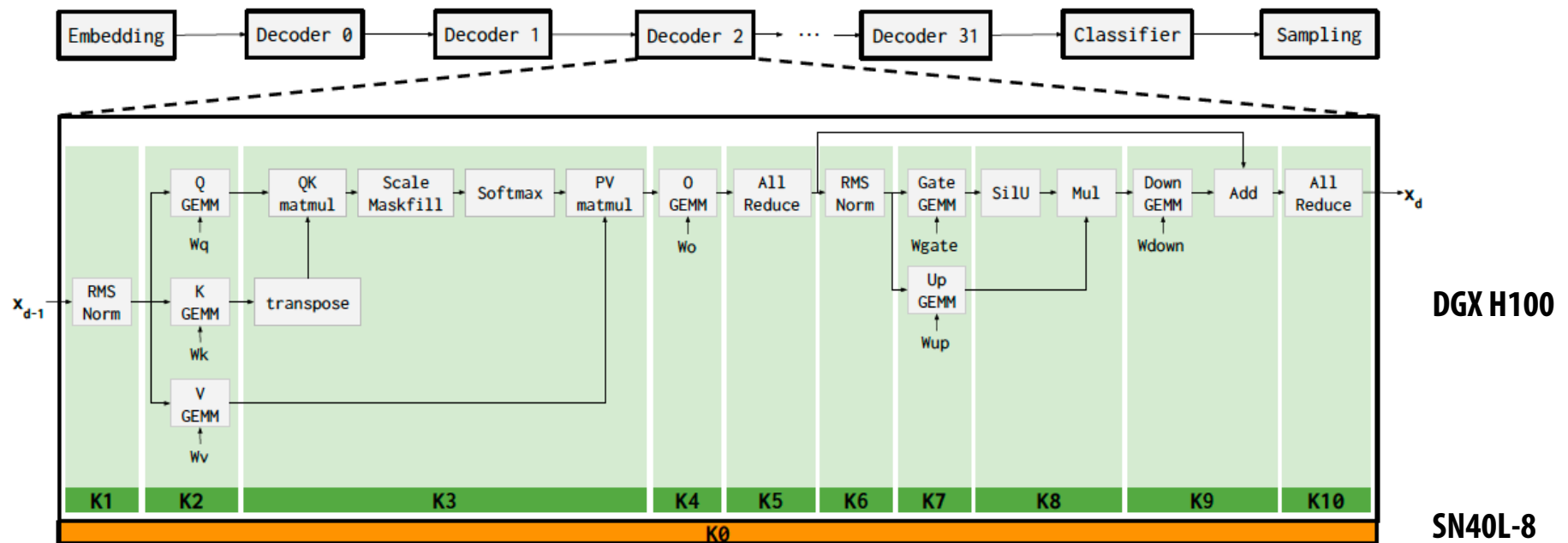
Tensor Parallel Llama 3.1 8B

Parallelize across 8 chips



Tensor Parallel Llama 3.1 8B

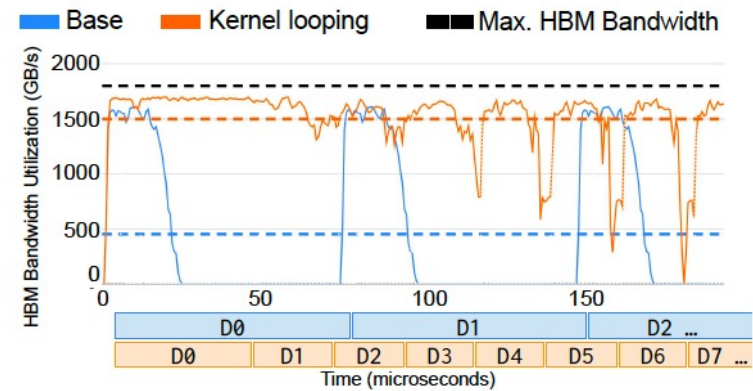
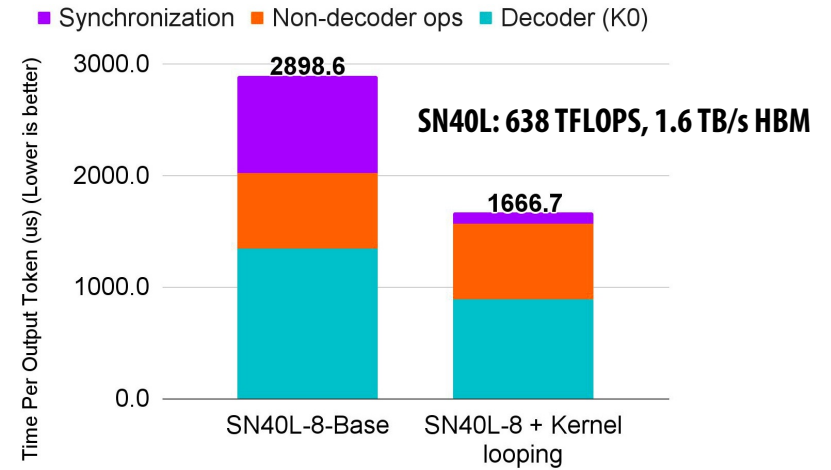
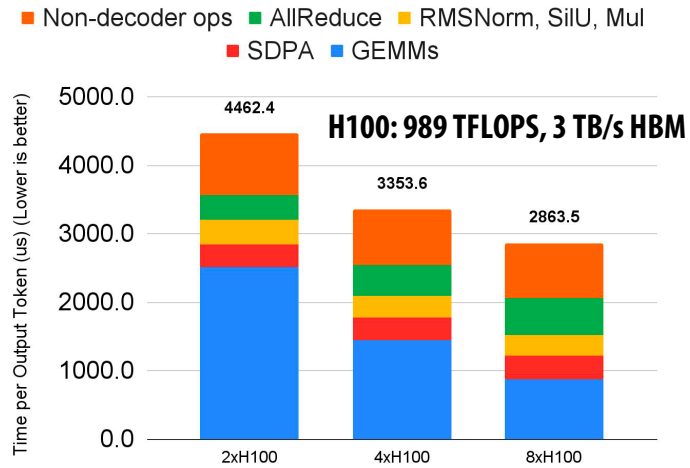
Parallelize across 8 chips



SN40L: Single kernel

- Allreduce is asynchronous and pipelined with other operators
- Kernel looping further reduces overheads

DGX H100 vs. SN40L-8



Summary: Specialized Hardware and Programming for DNN Processing

Specialized hardware for executing key DNN computations efficiently

Feature large/many matrix multiply units

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

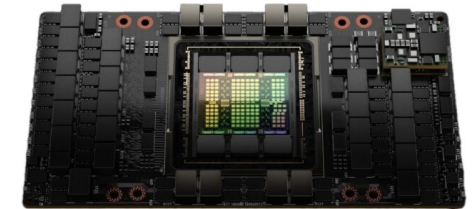
H100: Asynchronous compute and memory mechanisms \Rightarrow complex programming

- Need ThunderKittens to manage complexity

SN40L: Dataflow model with metapipelining \Rightarrow simpler programming model

- Sophisticated compiler to optimize and map to dataflow hardware

Minimizing synchronization overheads required for high performance

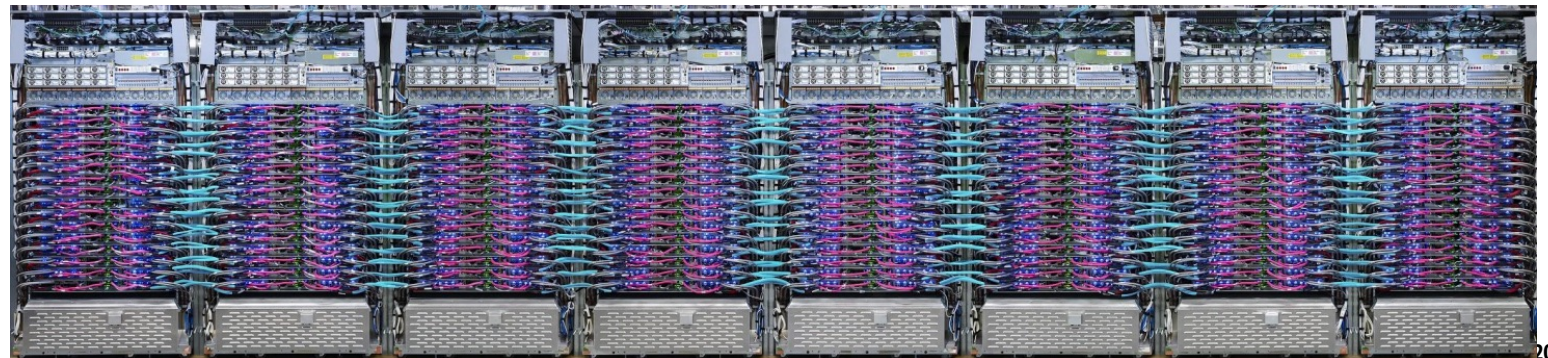


H100



SN40L

TPU supercomputer
(1024 TPU v3 chips)



Reducing energy consumption idea 1:
use specialized processing
(use the right processor for the job)

Reducing energy consumption idea 2:
move less data

Data movement 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!

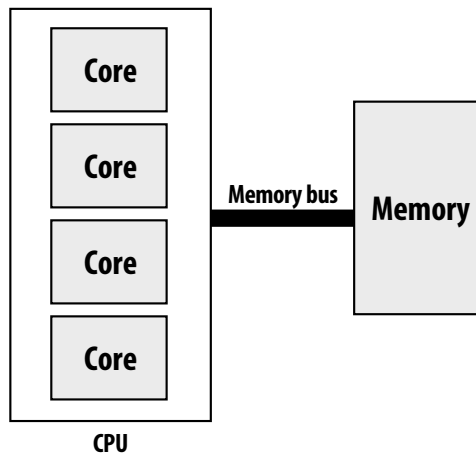
Data movement limits performance

Many processing elements...

= higher overall rate of memory requests

= need for more memory bandwidth

(result: bandwidth-limited execution)

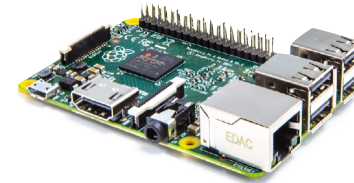


Data movement has high energy cost

~ 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

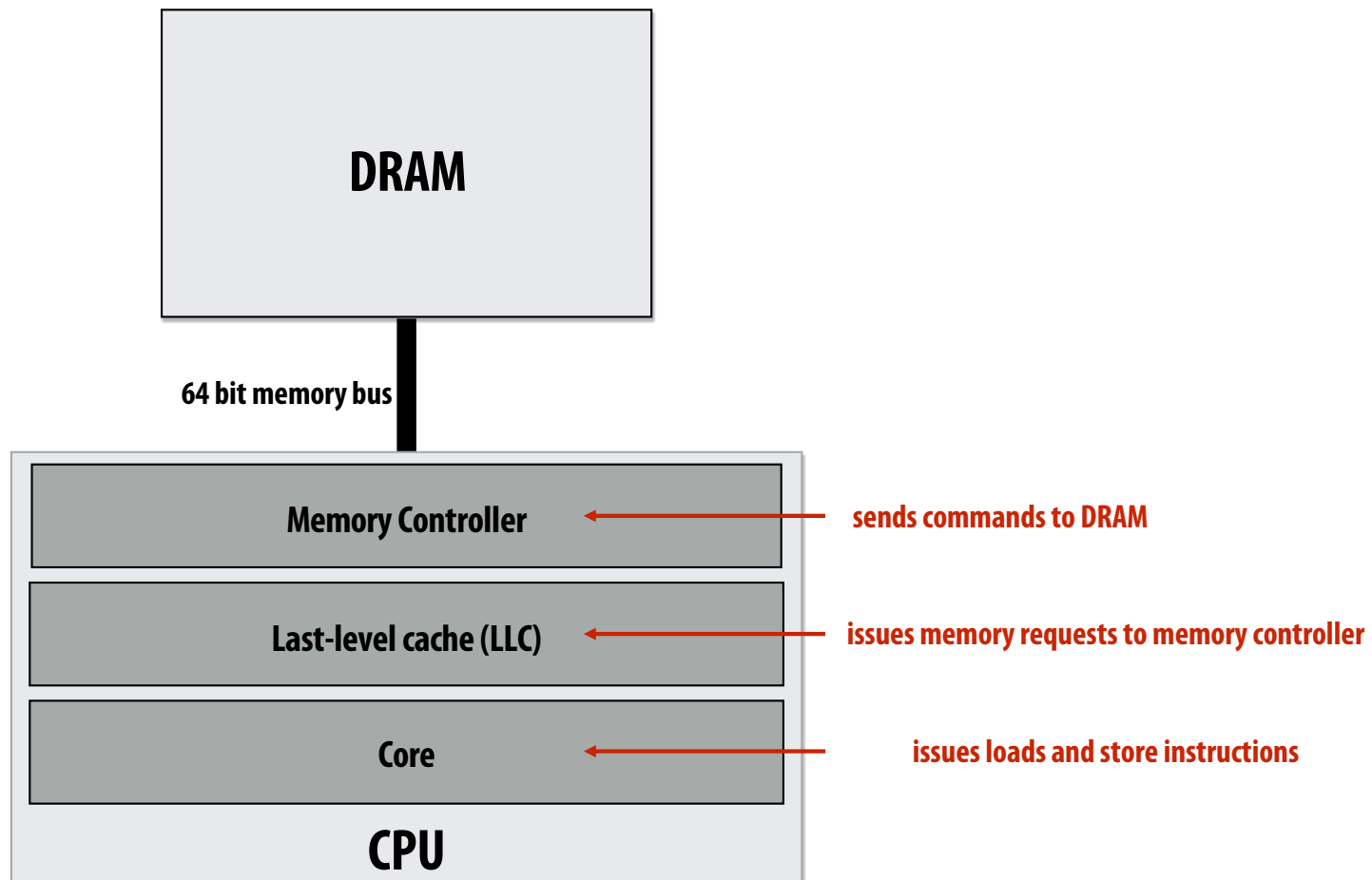


* Source: [Han, ICLR 2016], 45 nm CMOS assumption

Accessing DRAM

(a basic tutorial on how DRAM works)

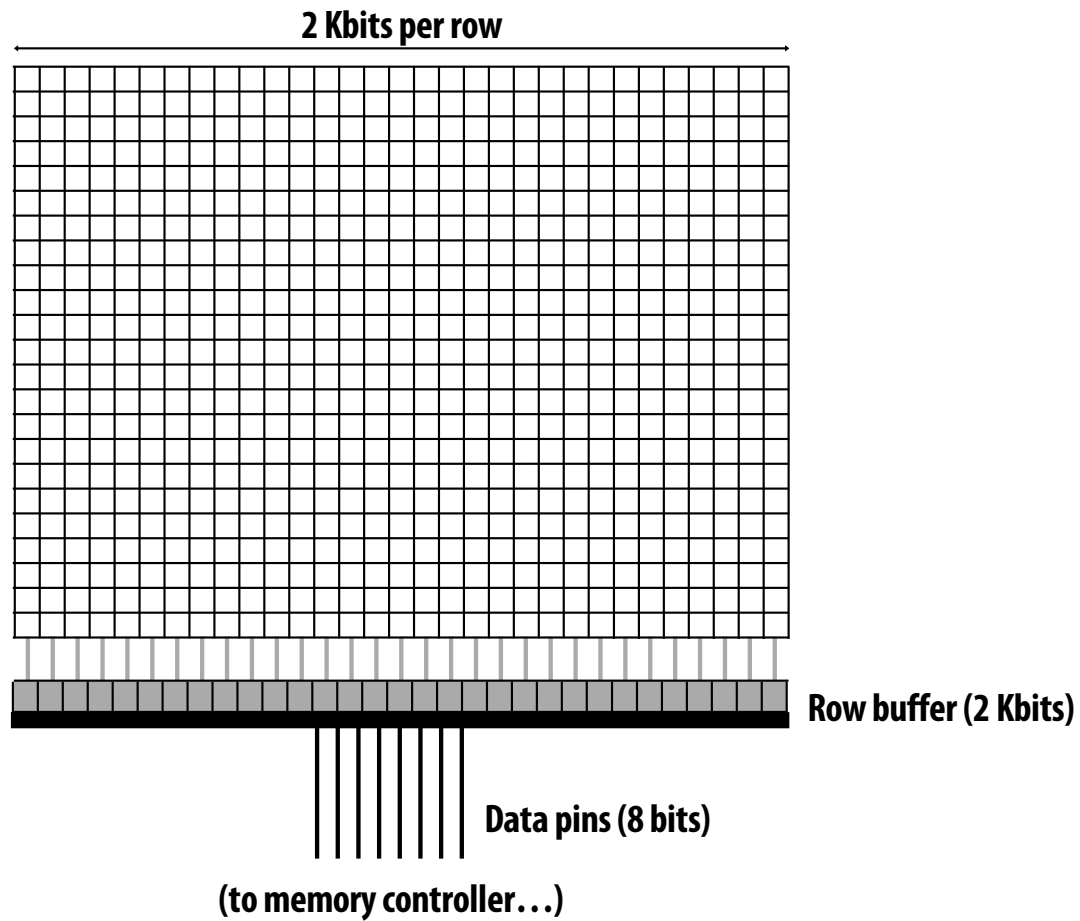
The memory system



DRAM array

1 transistor + capacitor per "bit"

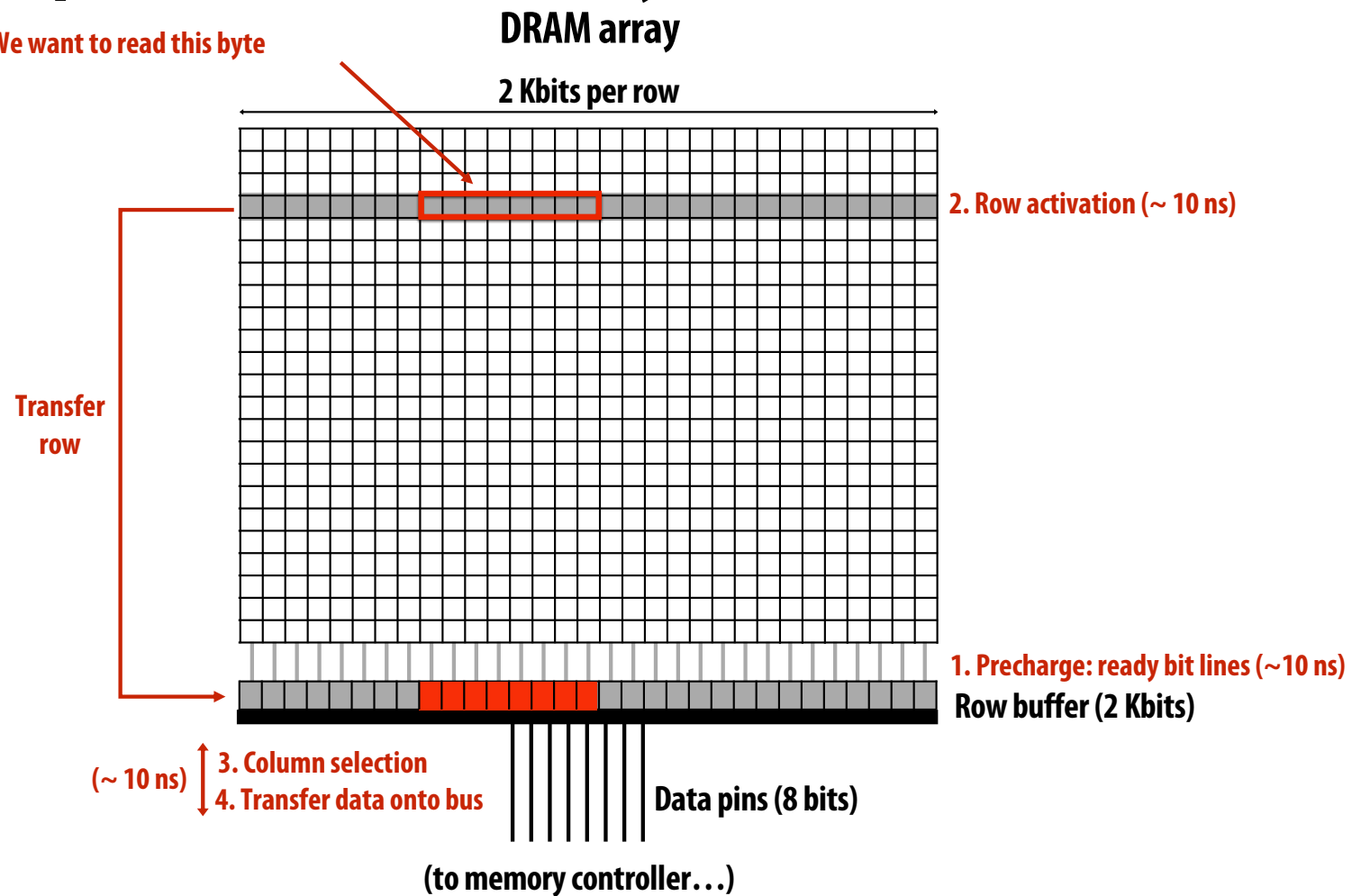
(Recall: a capacitor stores charge)



DRAM operation (load one byte)

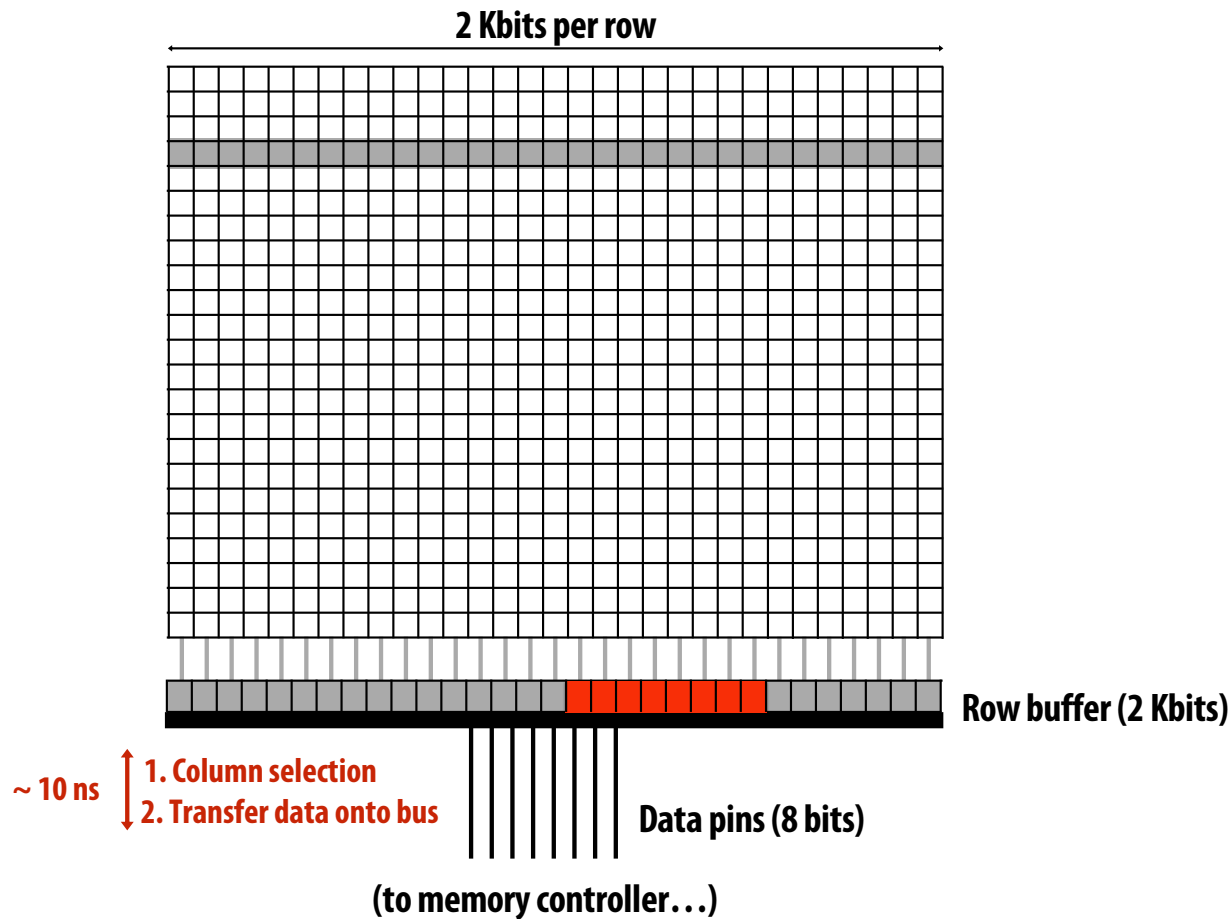
Estimated latencies are in units of
memory clocks: DDR3-1600

We want to read this byte



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)



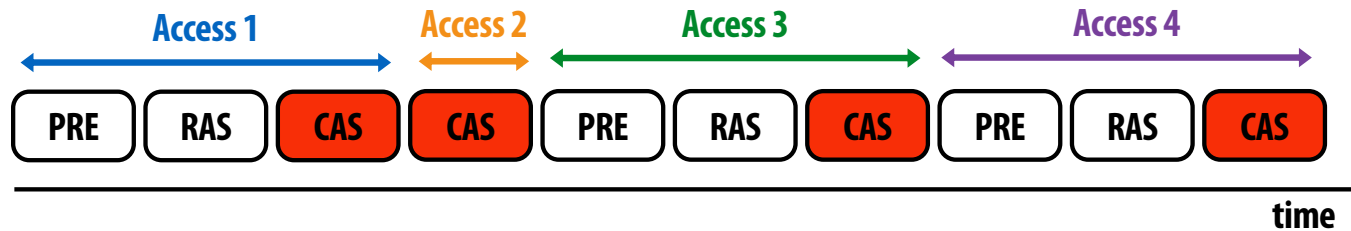
Question 1: when to execute precharge?

After each column access?

Only when new row is accessed?

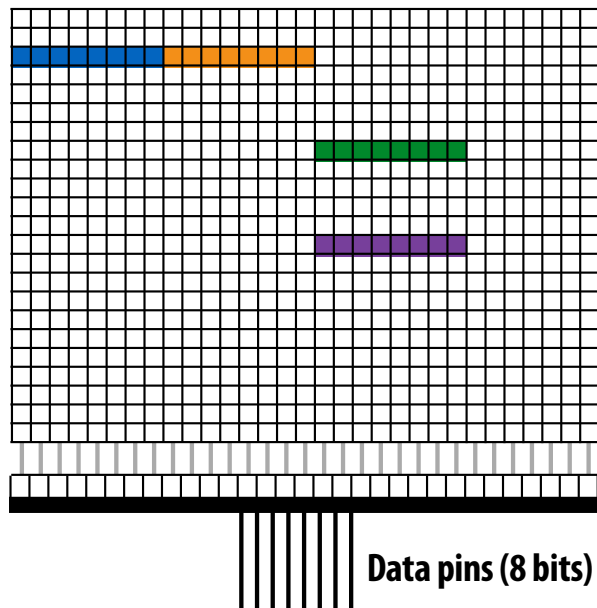
Question 2: how to handle latency of DRAM access?

Problem: low pin utilization due to latency of access

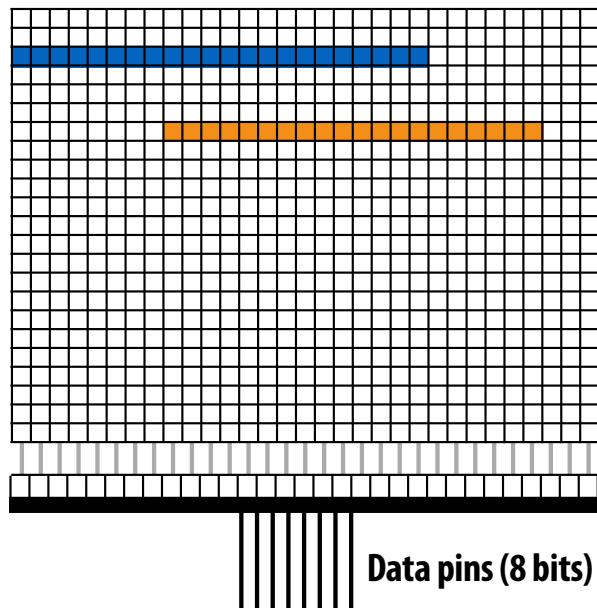
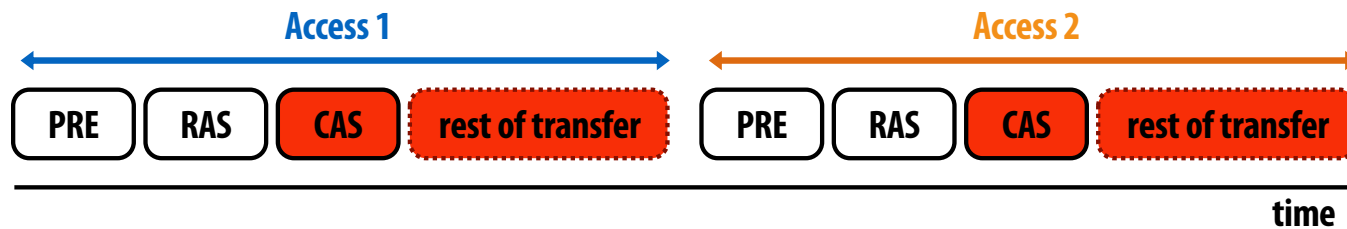


Data pins in use only a small fraction of time
(red = data pins busy)

This is bad since they are the scarcest resource!



DRAM burst mode



Idea: amortize latency over larger transfers

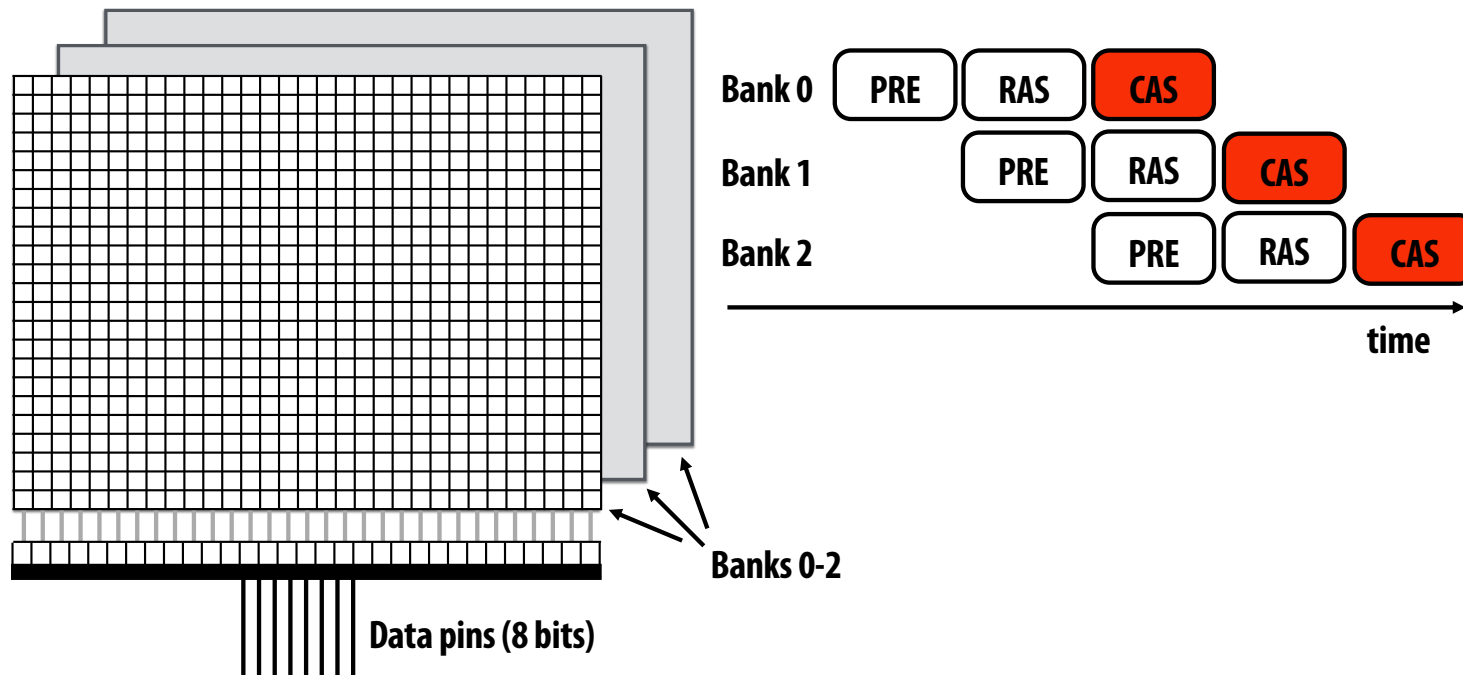
**Each DRAM command describes bulk transfer
Bits placed on output pins in consecutive clocks**

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

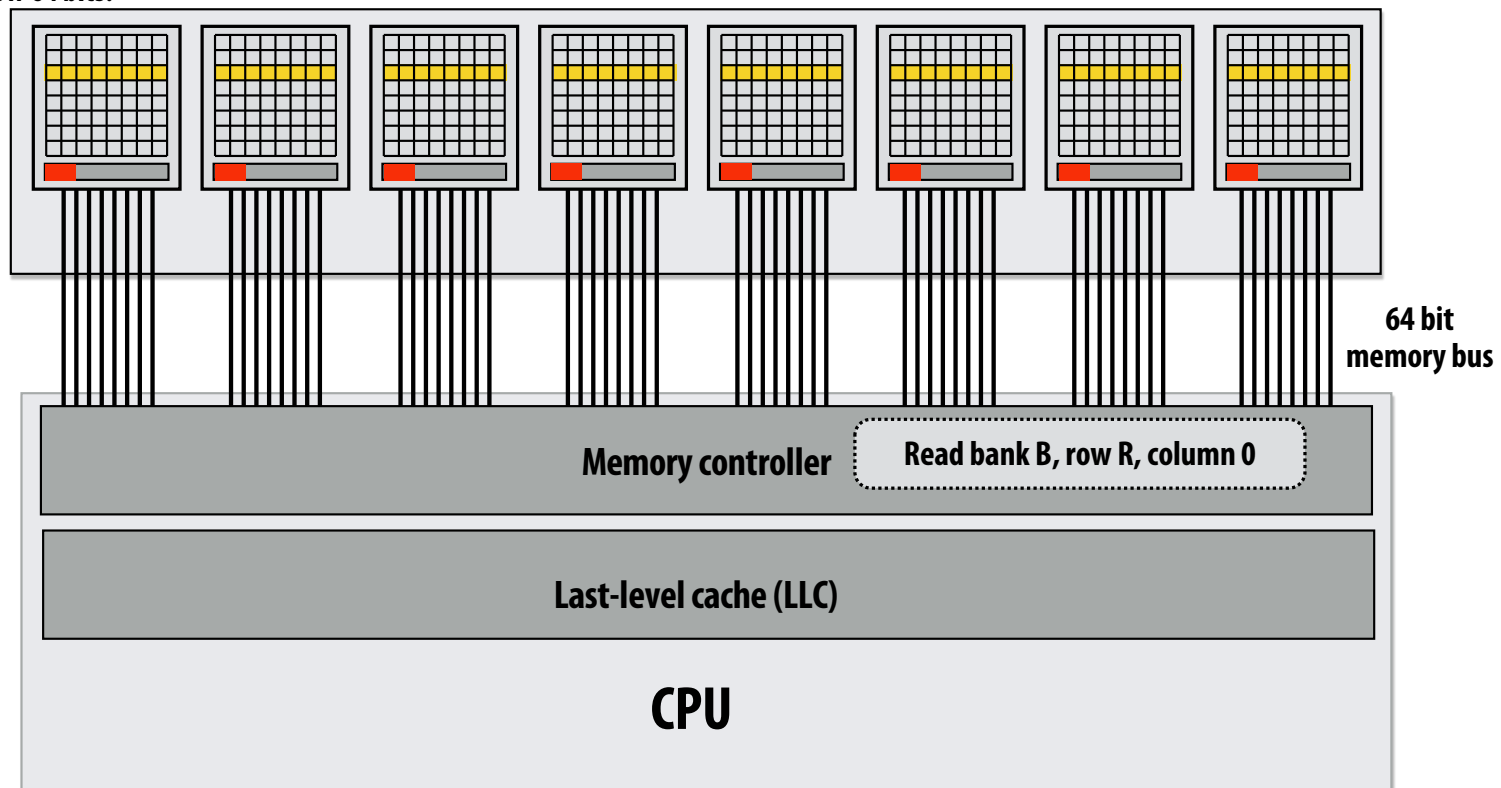


Organize multiple chips into a DIMM



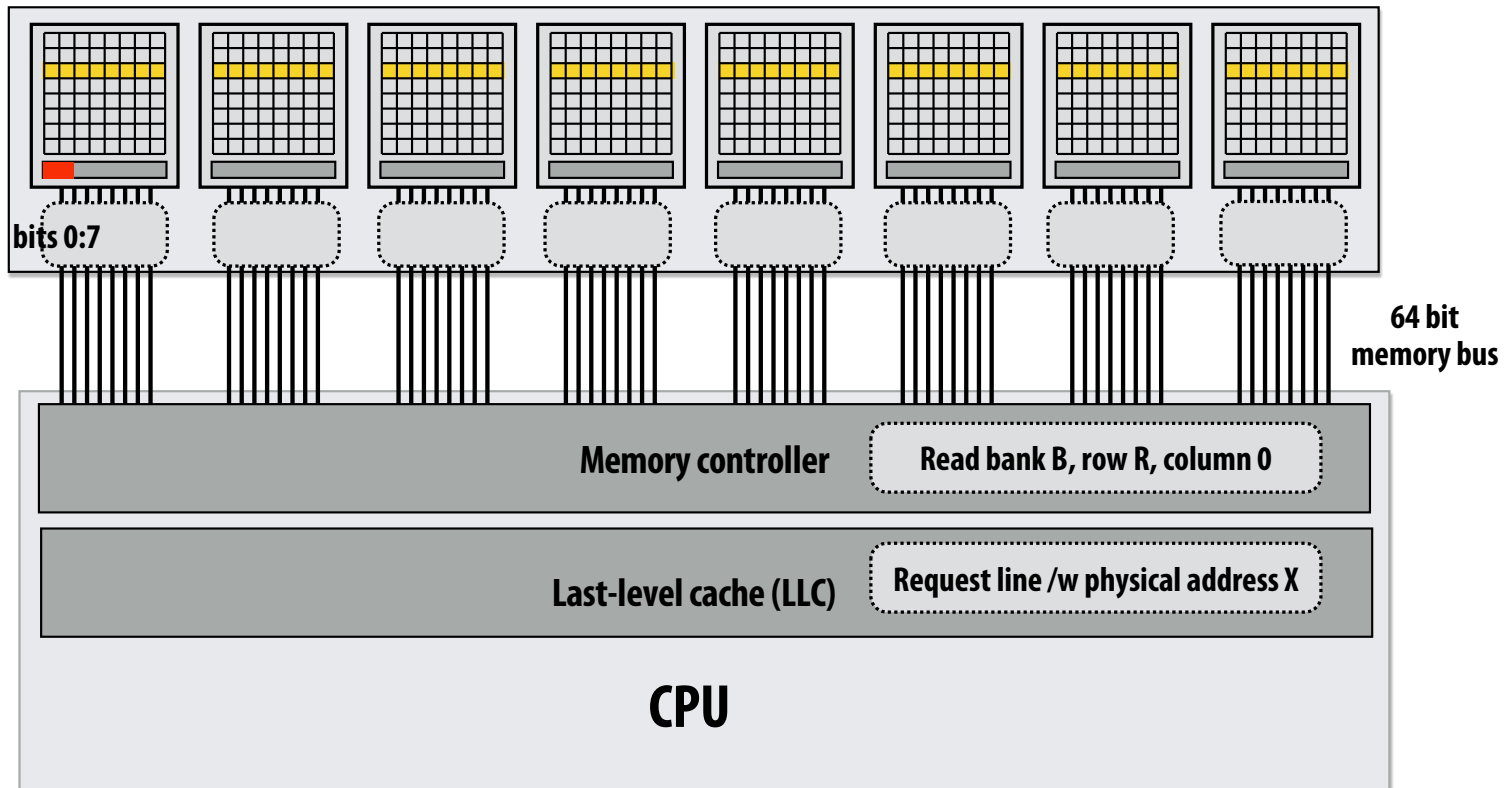
Example: Eight DRAM chips (64-bit memory bus)

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.



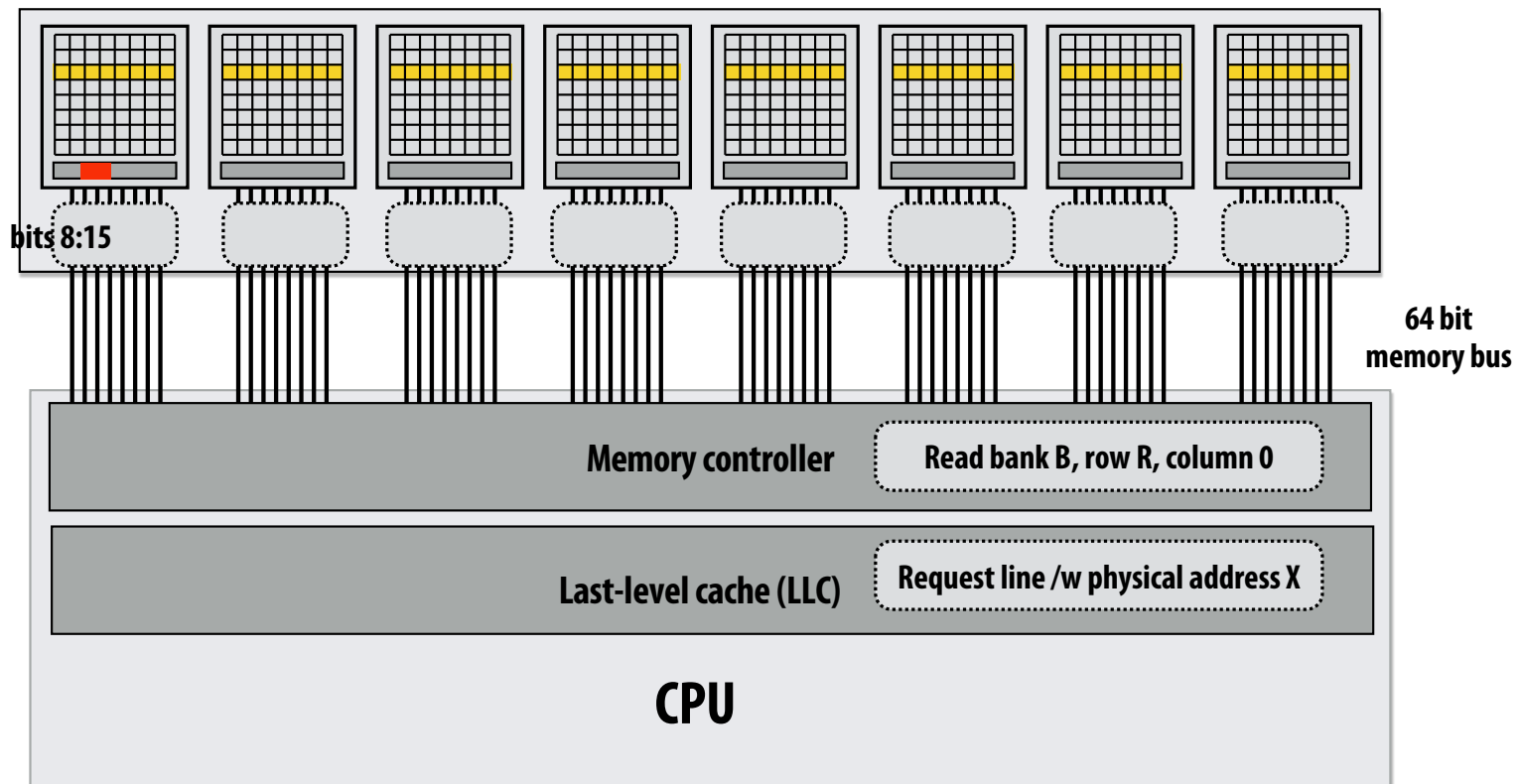
Reading one 64-byte (512 bit) cache line (the wrong way)

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



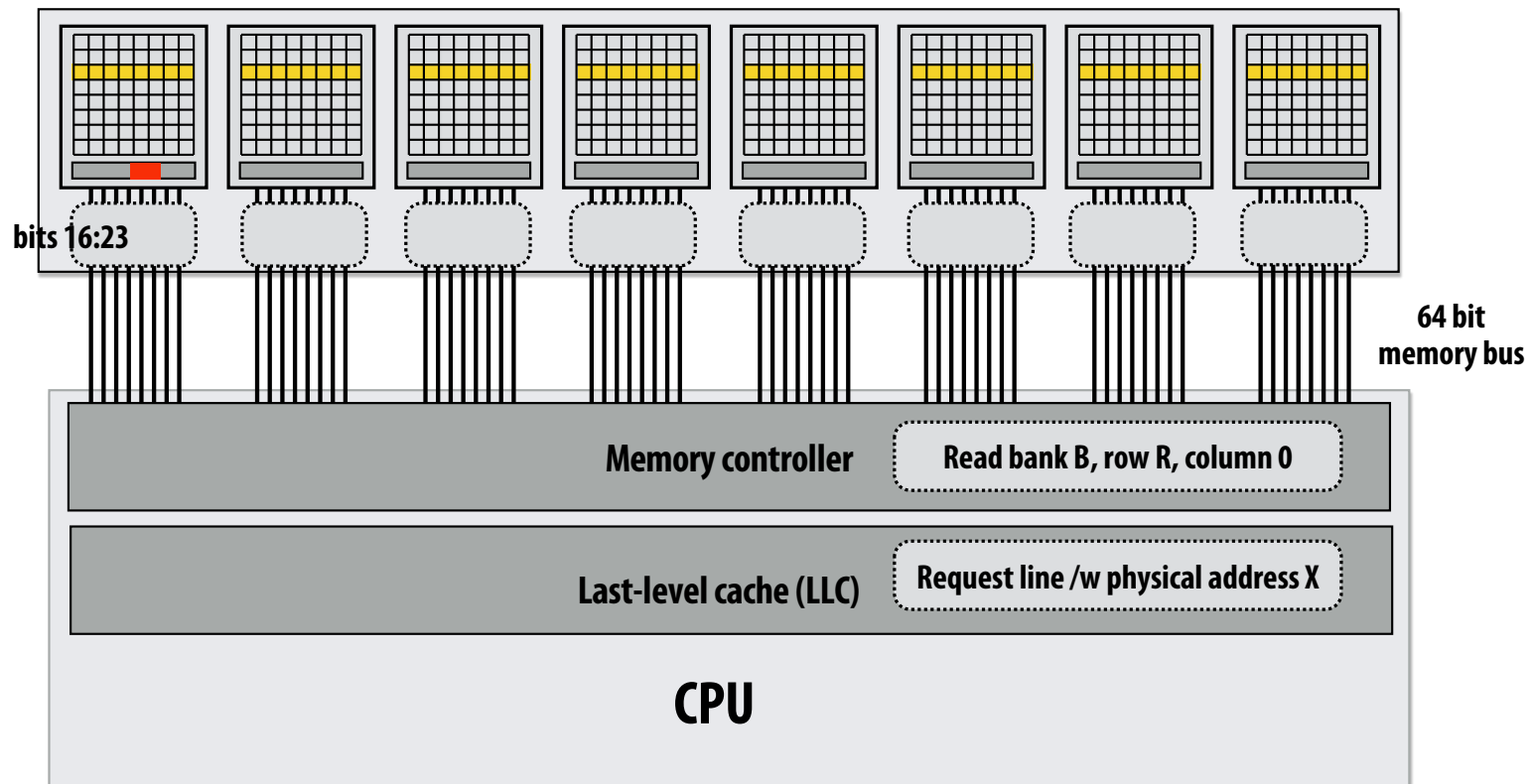
Reading one 64-byte (512 bit) cache line (the wrong way)

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

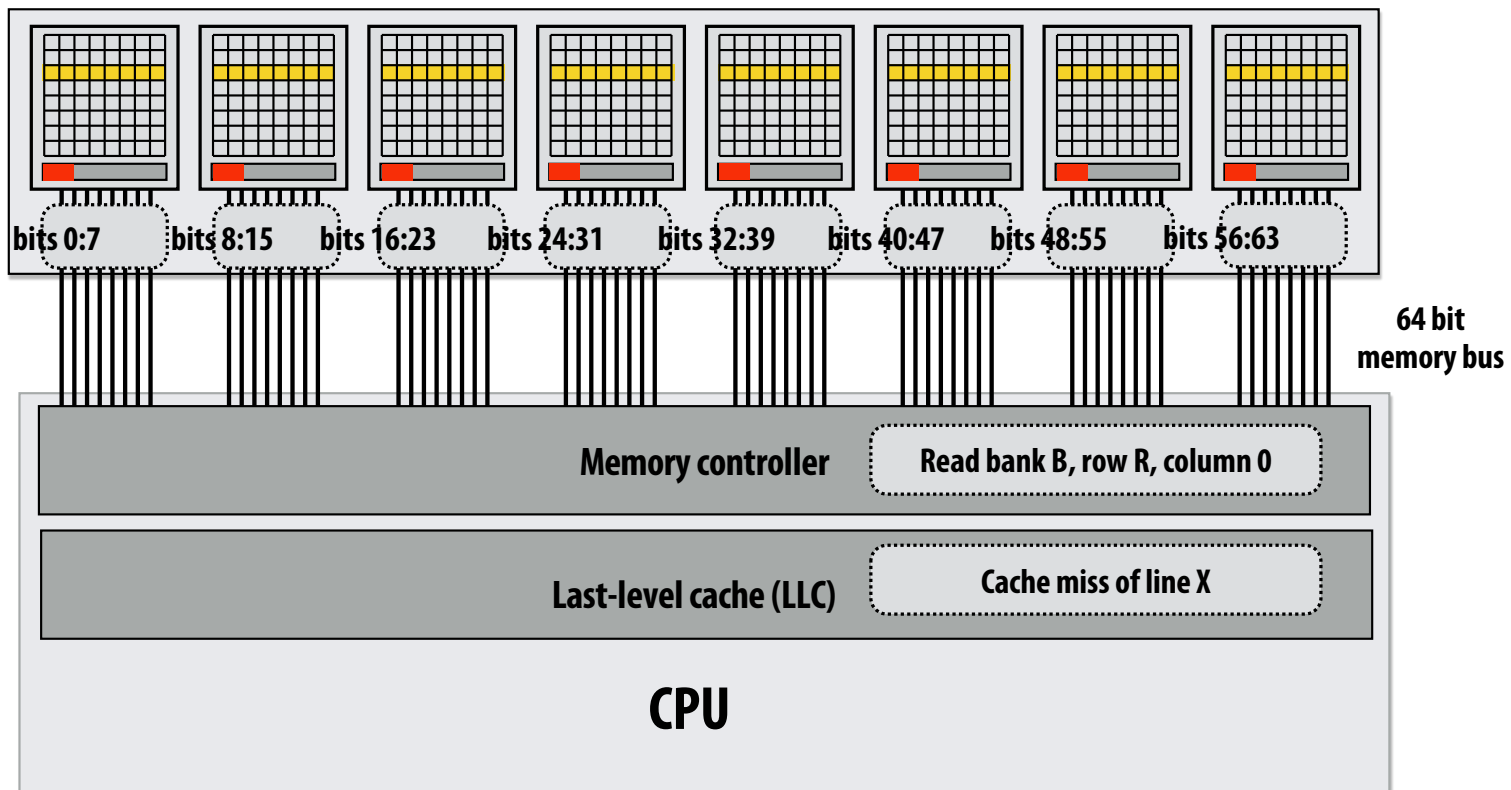


Reading one 64-byte (512 bit) cache line

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

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

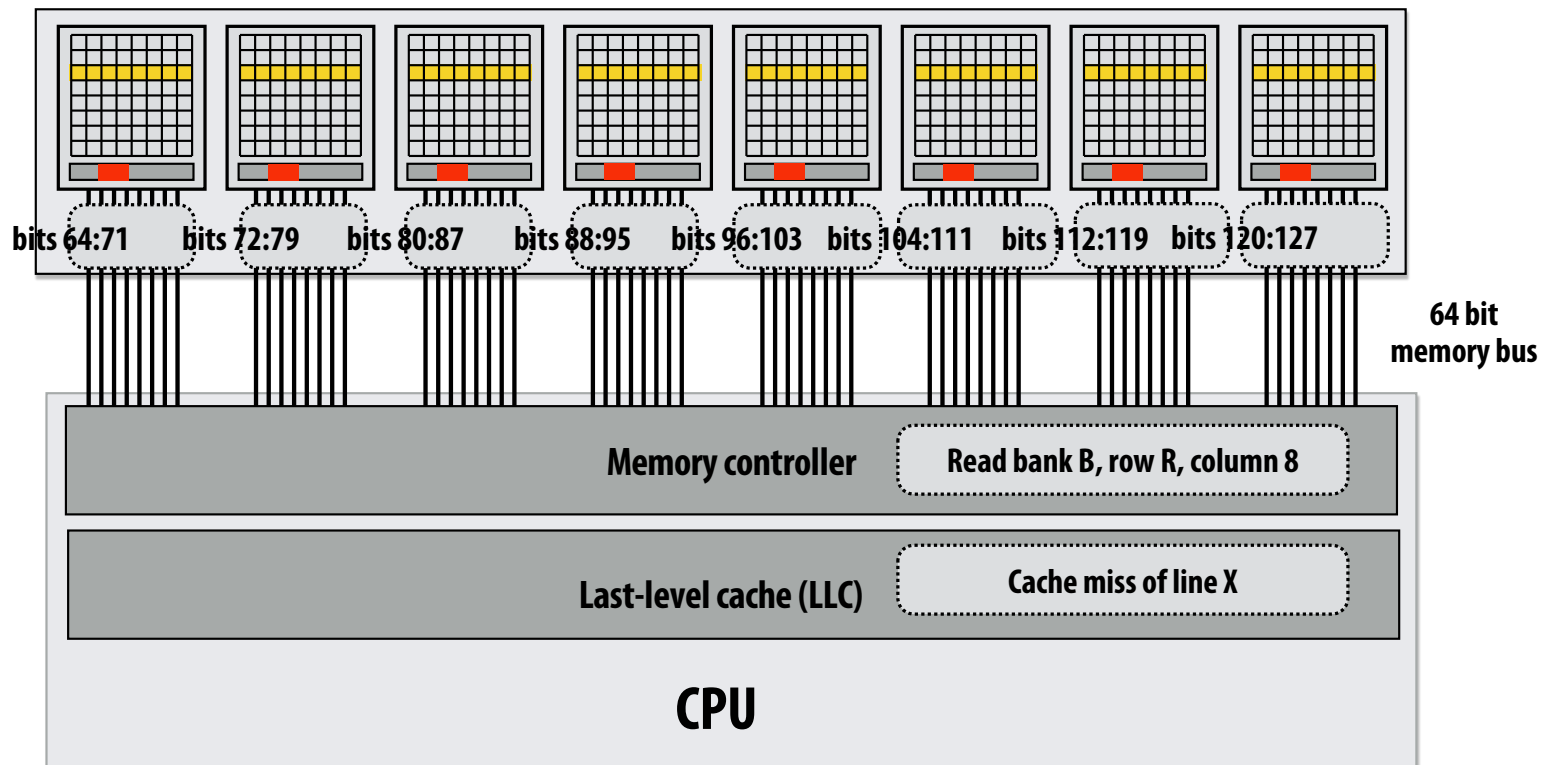
DRAM chips transmit first 64 bits in parallel



Reading one 64-byte (512 bit) cache line

DRAM controller requests data from new column *

DRAM chips transmit next 64 bits in parallel



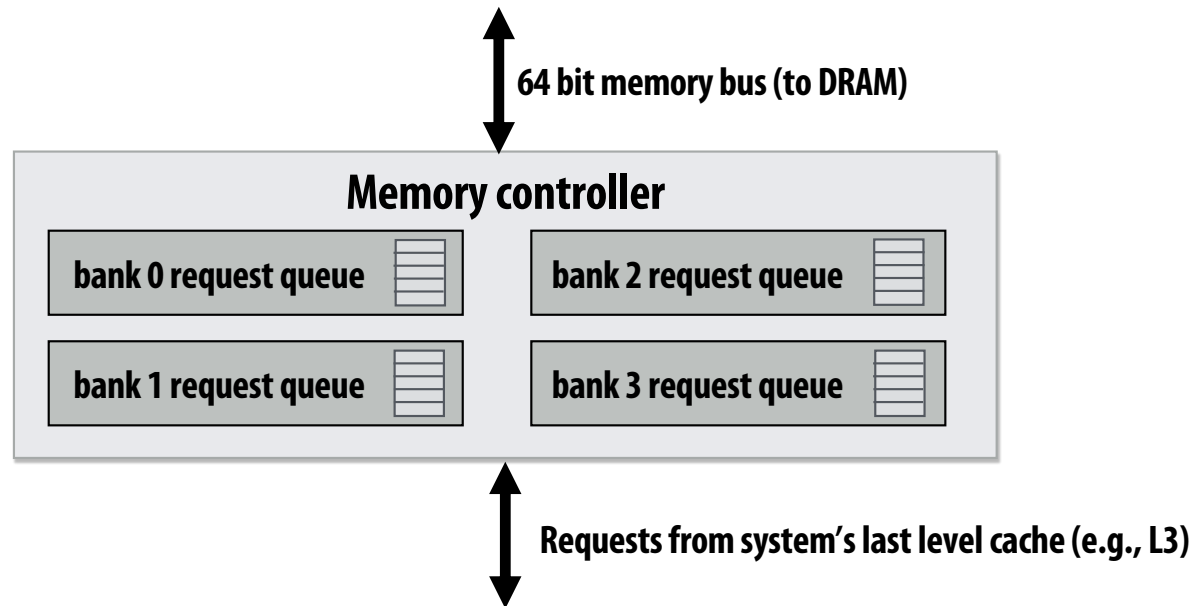
* Recall modern DRAM's support burst mode transfer of multiple consecutive columns, which would be used here

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”)

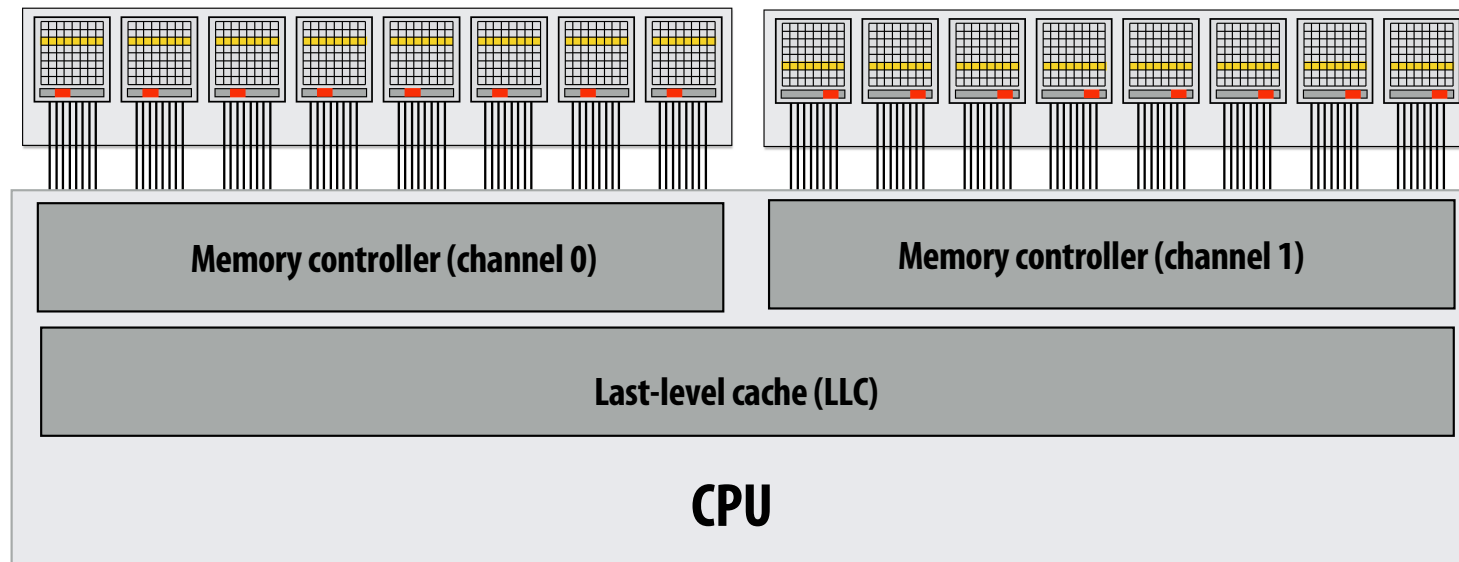


Dual-channel memory system

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:**

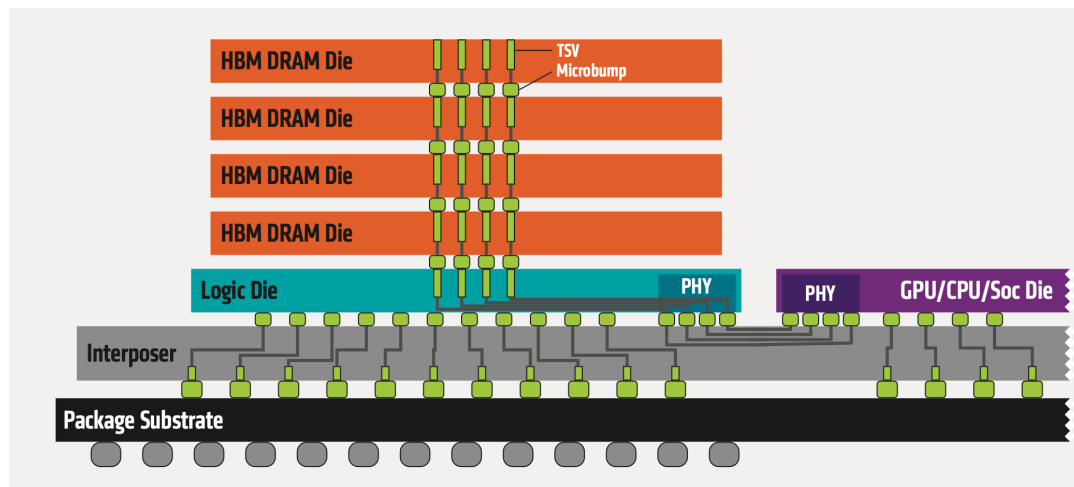
**Decrease distance data must move by
locating memory closer to processors**

(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

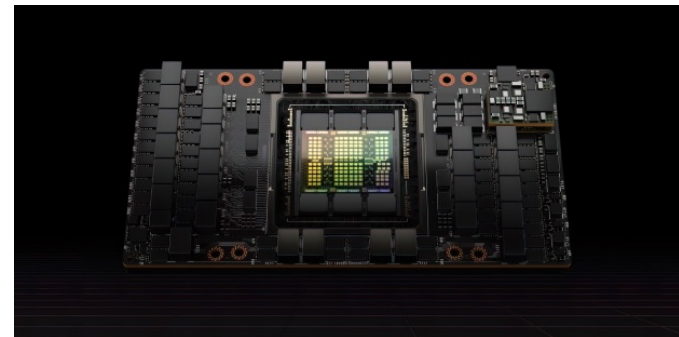
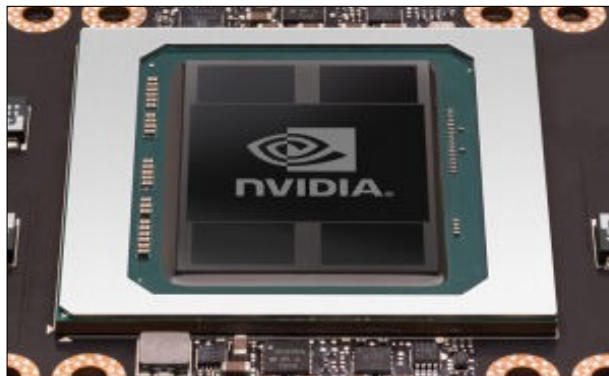
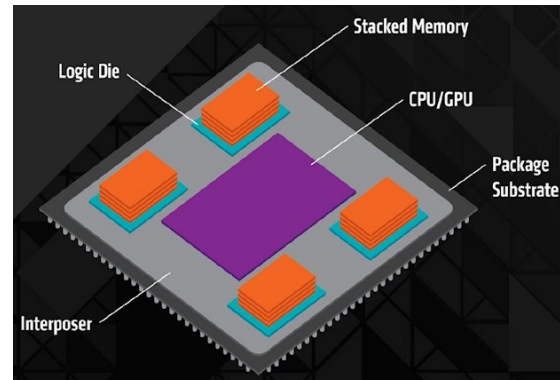
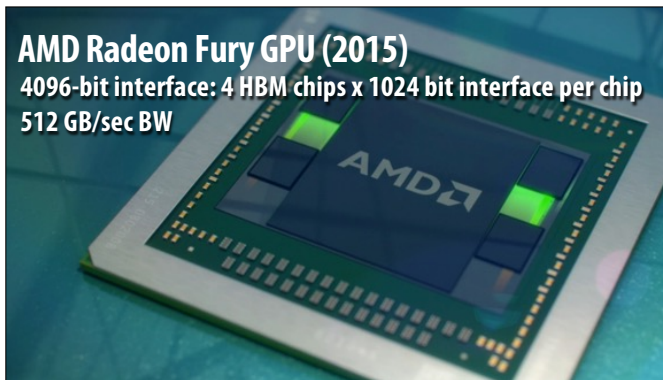
Image credit: AMD

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 16Gbps)	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/24Gb/32Gb	8Gb/16Gb	4GB/8GB	8GB/16GB	8GB/16GB/24GB (TBD)

GPUs are adopting HBM technologies



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

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)