Lecture 23:

Addressing the Memory Wall

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2016

Tunes

XX

Swept Away

(Coexist)

"Have you seen how much bandwidth we're about to get from 3D memory stacking?"

- Oliver Sim

Announcements

- Deadline adjustments due to Carnival
- Students at not allowed to work on 418/618 on Thursday or Friday
- Exercise 6 (the final one!) will be released on Wed as normal
  - But it will not be due until 10am on Monday
- Project checkpoint is coming up
  - Next Tuesday, 11:56pm

Today's topic: moving data is costly!

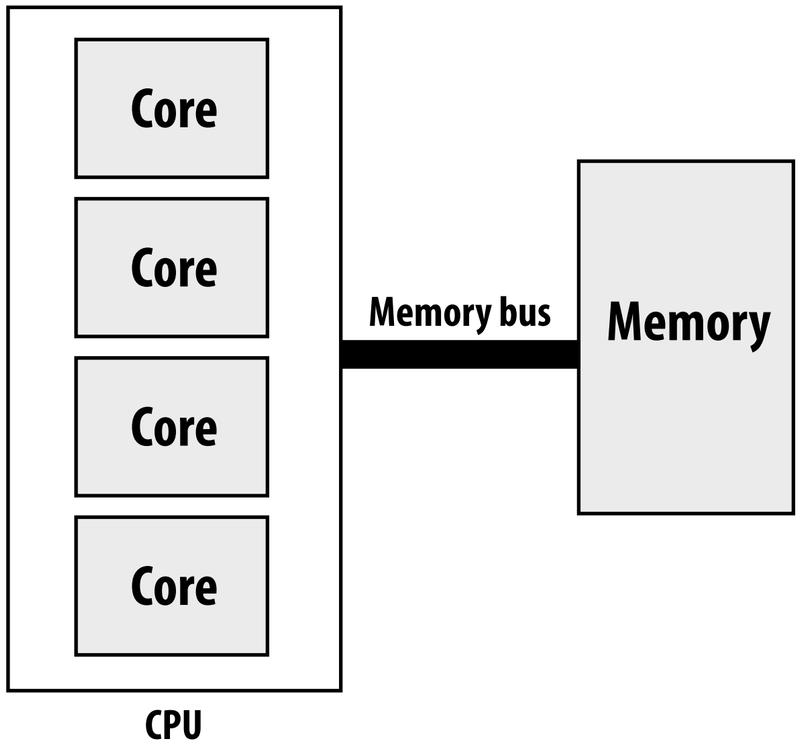
Data movement limits performance

Many processing elements…

= higher overall rate of memory requests

= need for more memory bandwidth

(result: bandwidth-limited execution)

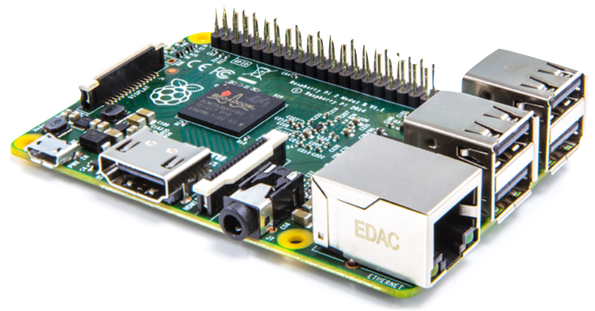


Data movement has high energy cost

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

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

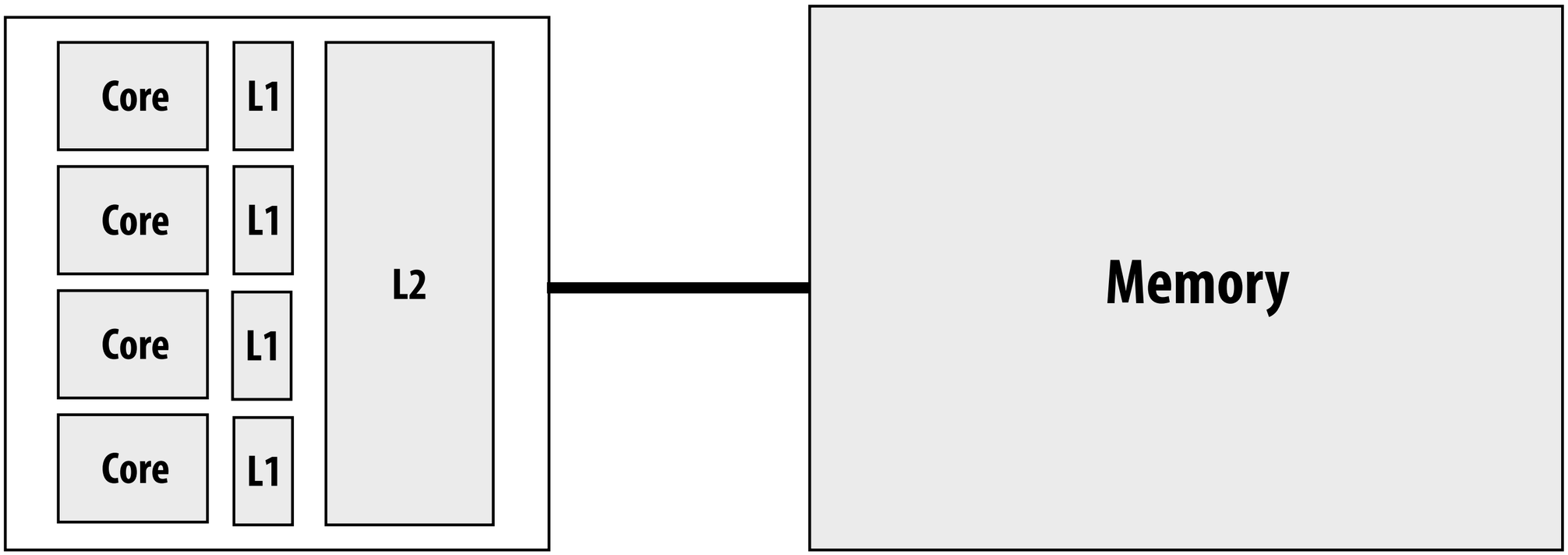
~ 640 pJ to load 32 bits from LPDDR memory



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

Well written programs exploit locality to avoid redundant data transfers between CPU and memory

(Key idea: place frequently accessed data in caches/buffers near processor)



- Modern processors have high-bandwidth (and low latency) access to local memories
  - Computations featuring data access locality can reuse data in local memories
- Software optimization technique: reorder computation so that cached data is accessed it many times before it is evicted
- Performance-aware programmers go to great effort to improve the cache locality of programs
  - What are good examples from this class?

Example 1: restructuring loops for locality

Program 1

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Program 2

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

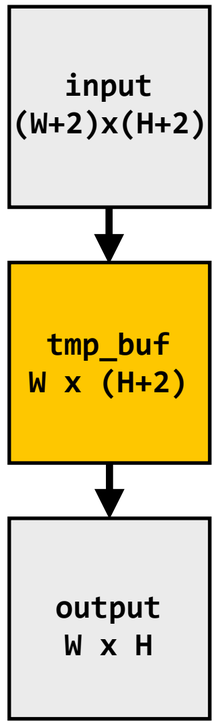
Four loads, one store per 3 math ops (arithmetic intensity = 3/5)

The transformation of the code in program 1 to the code in program 2 is called "loop fusion"

Example 2: restructuring loops for locality

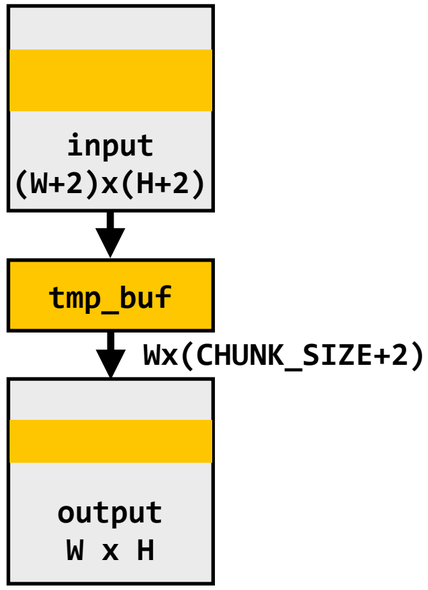
Program 1

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

// blur image horizontally
for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

// blur tmp_buf vertically
for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```



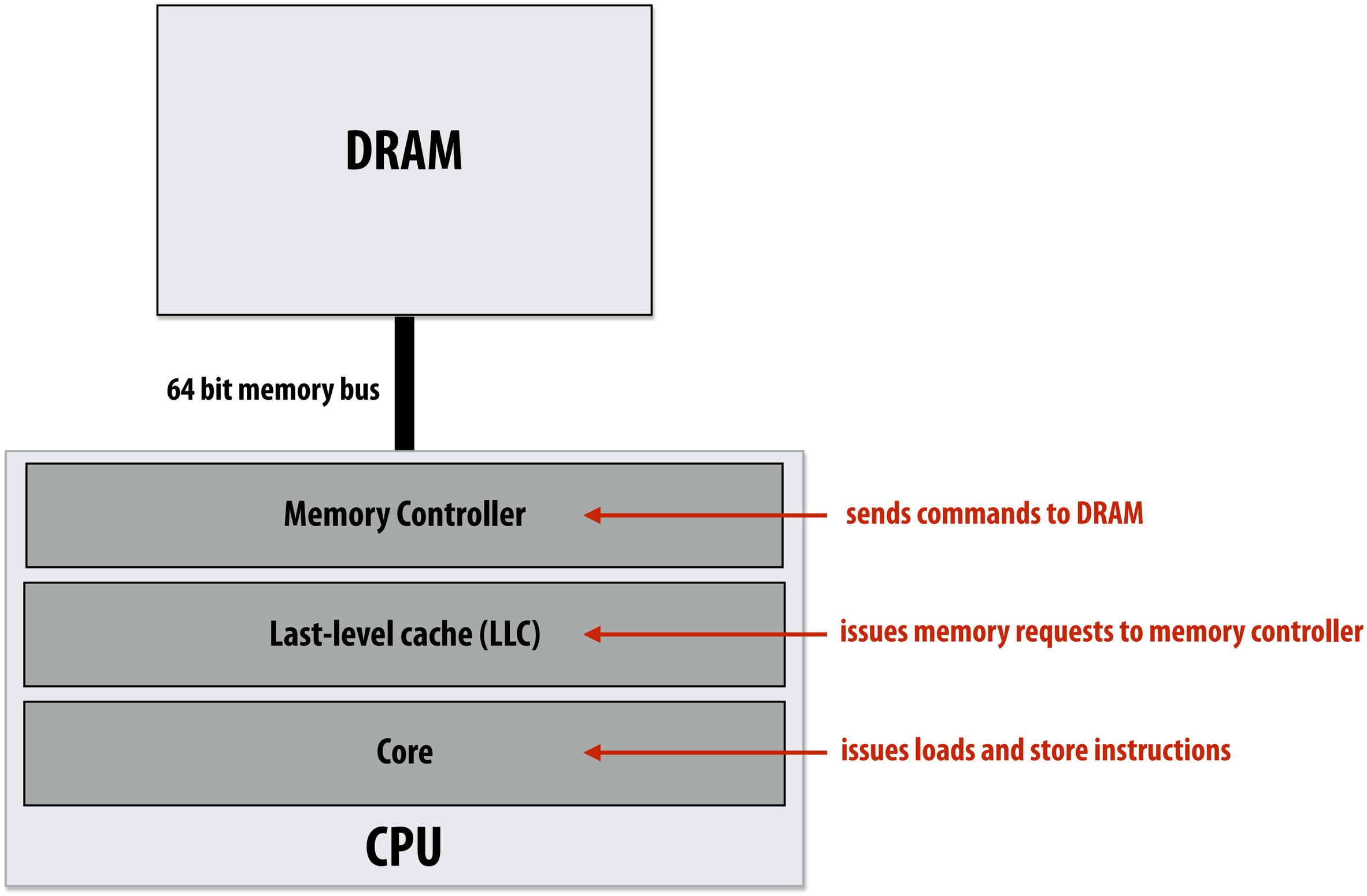
Program 2

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {

  // blur region of image horizontally
  for (int j2=0; j2<CHUNK_SIZE+2; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int ii=0; ii<3; ii++)
        tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
      tmp_buf[j2*WIDTH + i] = tmp;

  // blur tmp_buf vertically
  for (int j2=0; j2<CHUNK_SIZE; j2++)
    for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
        tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
      output[(j+j2)*WIDTH + i] = tmp;
    }
}
```



Accessing DRAM
(a basic tutorial on how DRAM works)

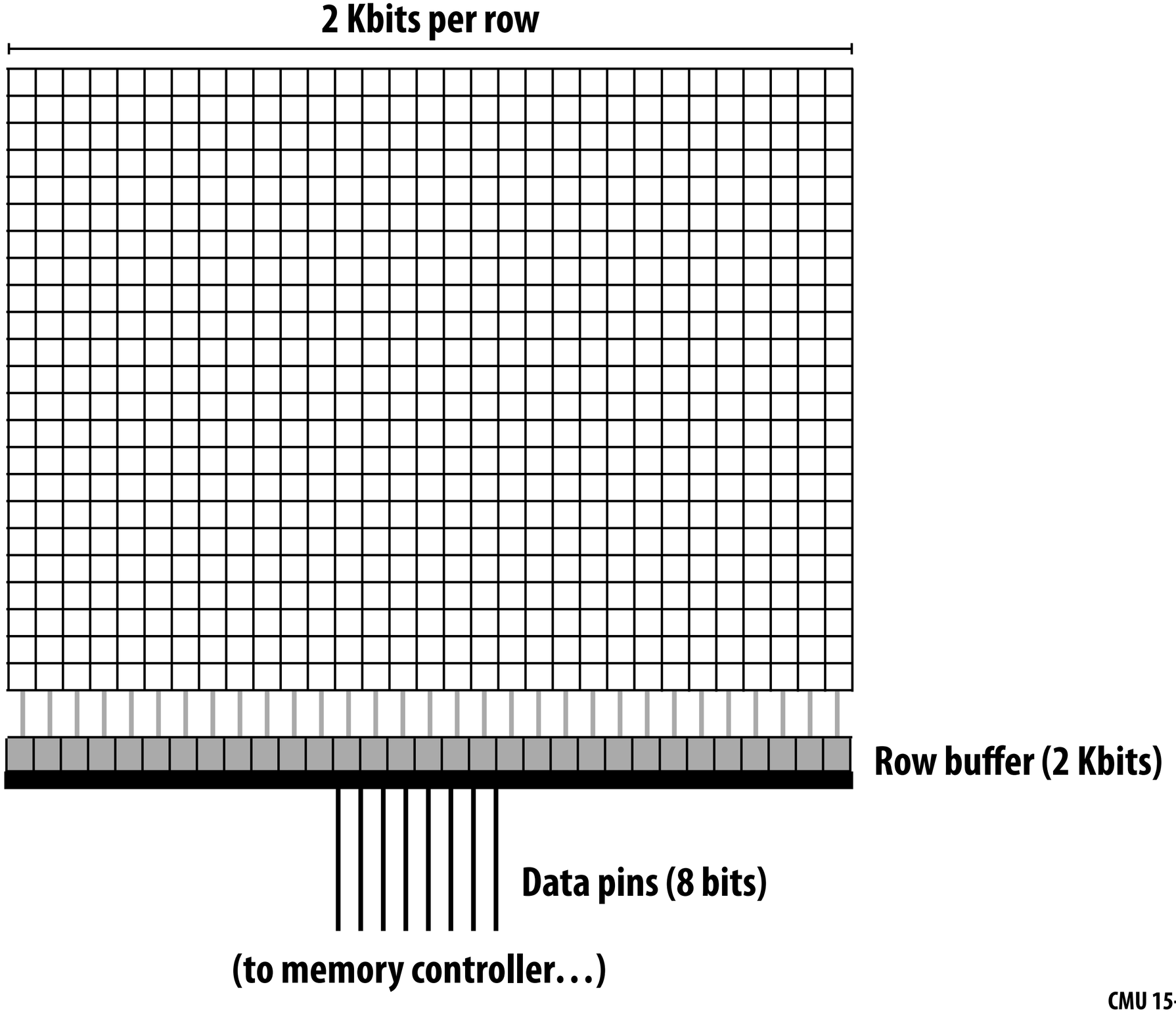
The memory system



DRAM array

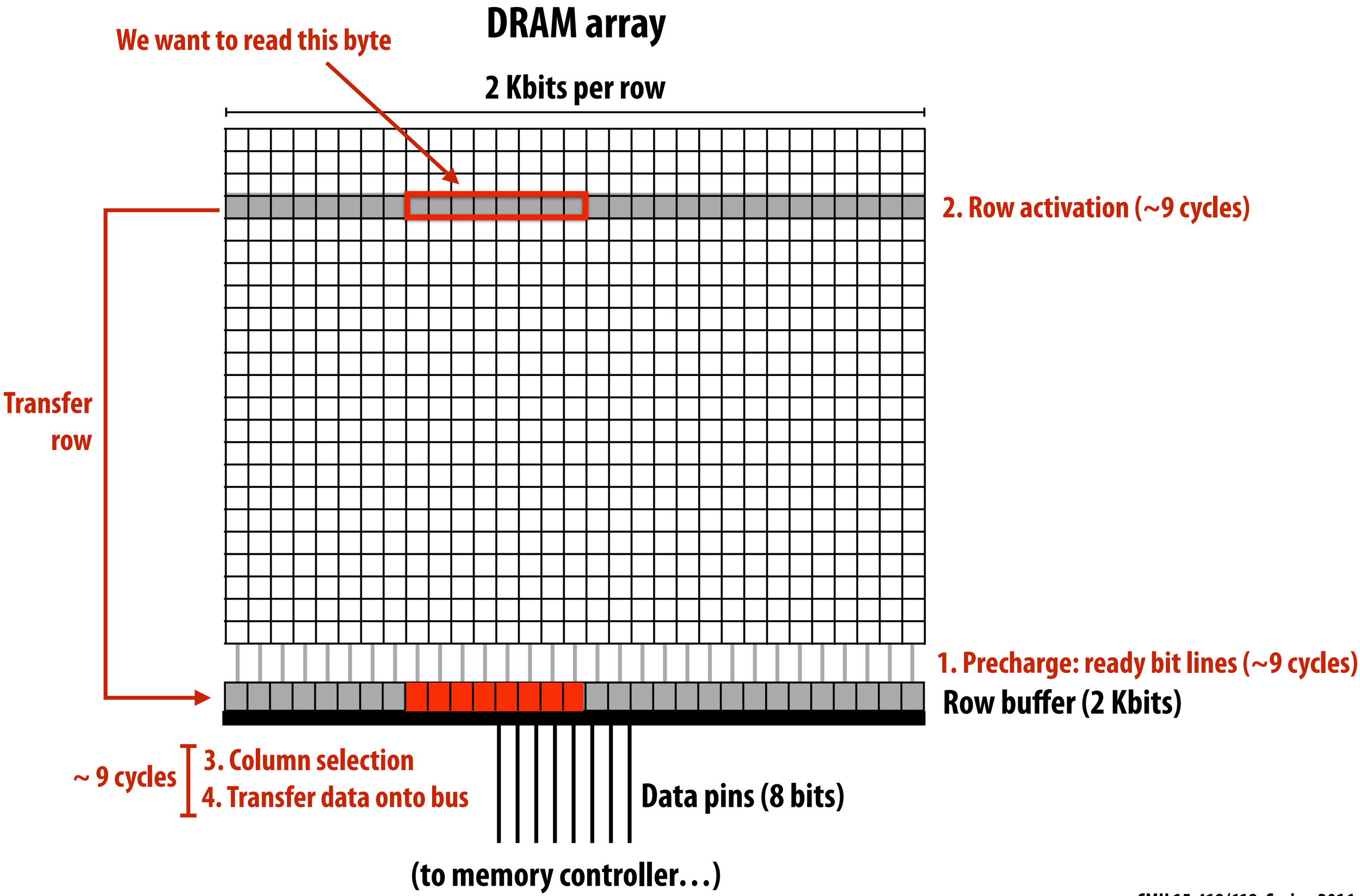
1 transistor + capacitor per "bit"

(Recall: a capacitor stores charge)



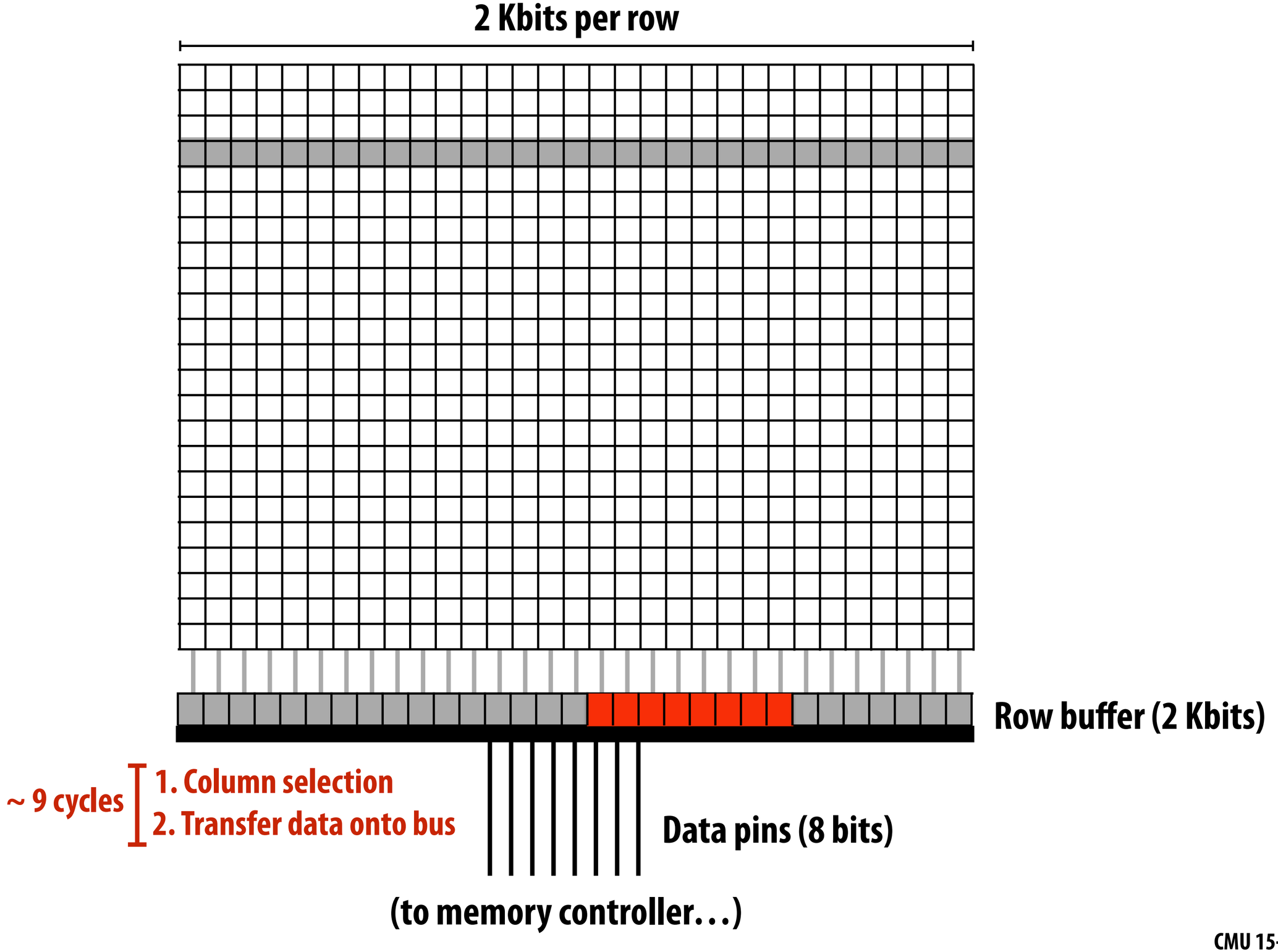
DRAM operation (load one byte)

Estimated latencies are in units of memory clocks: DDR3-1600 (Kayvon's laptop)



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)

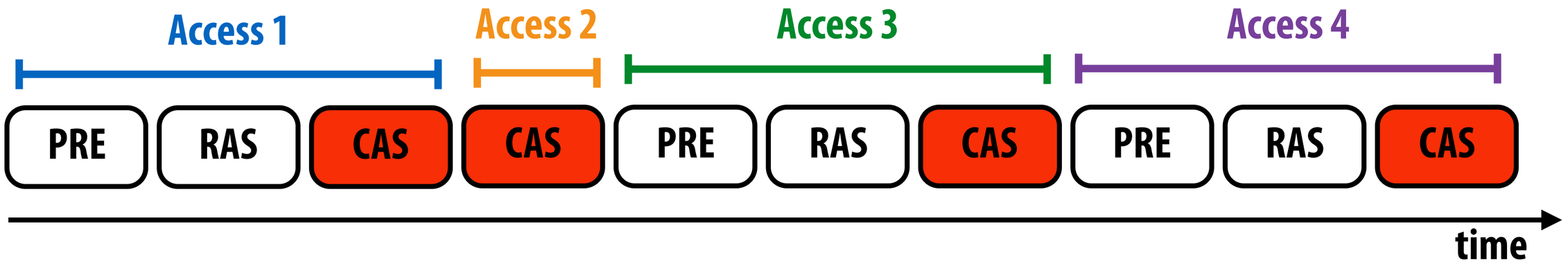
- **Worst case latency: bit lines not ready, read from new row**
  - Precharge (PRE) + row activate (RAS) + column access (CAS)

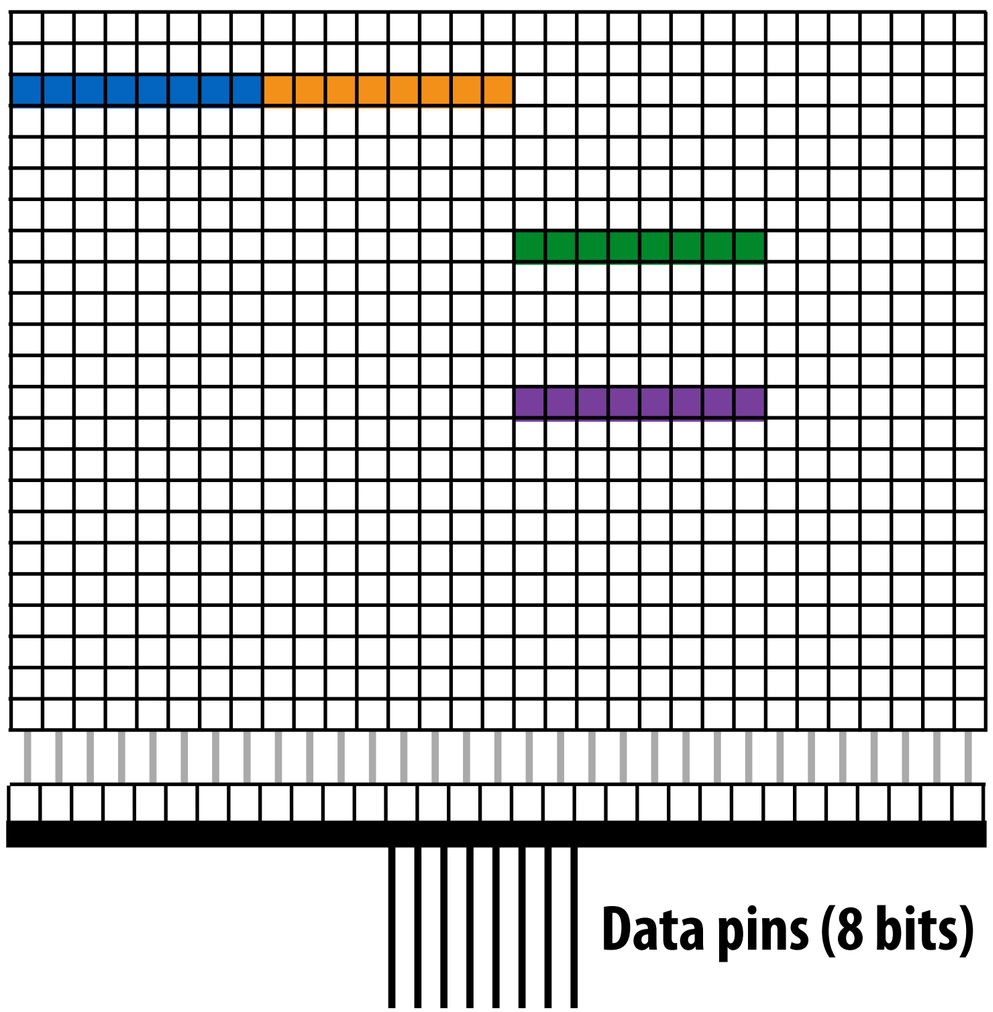
Precharge readies bit lines and writes row buffer contents back into DRAM array (read was destructive)

- **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

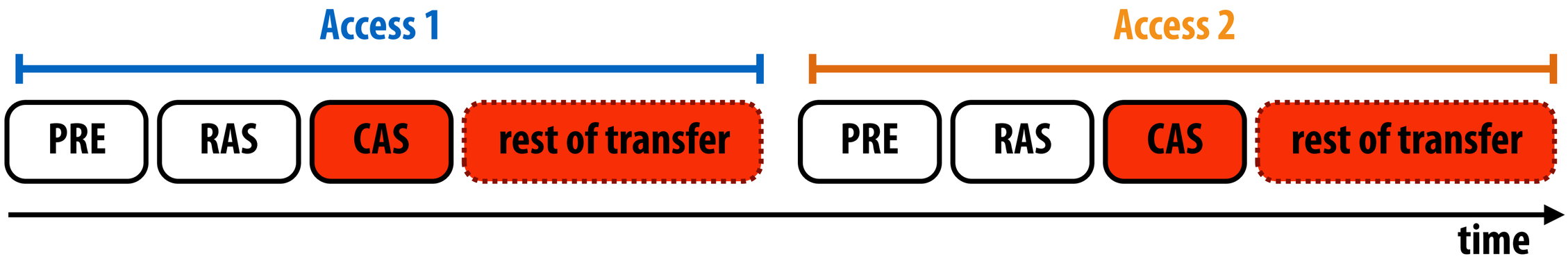


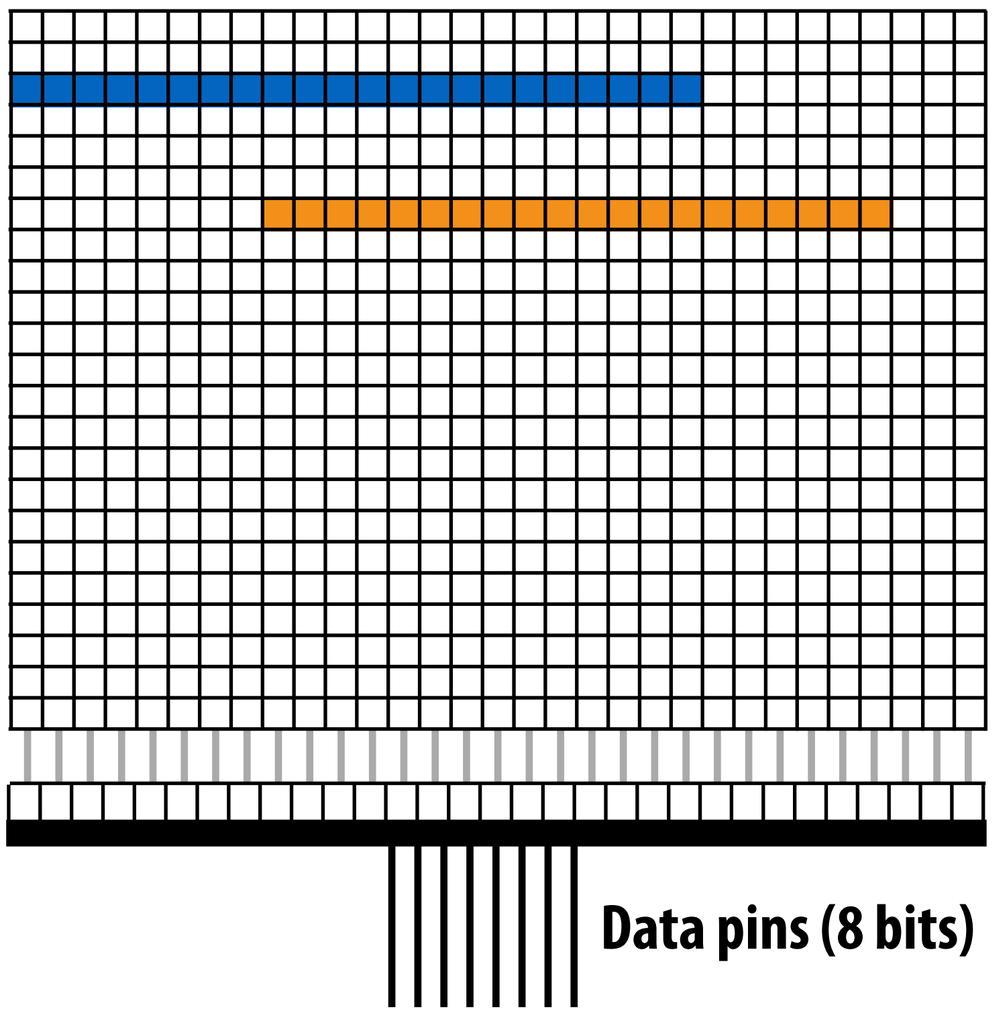


Data pins in use only a small fraction of time (red = data pins busy)

Very bad since they are the scarcest resource!

DRAM burst mode



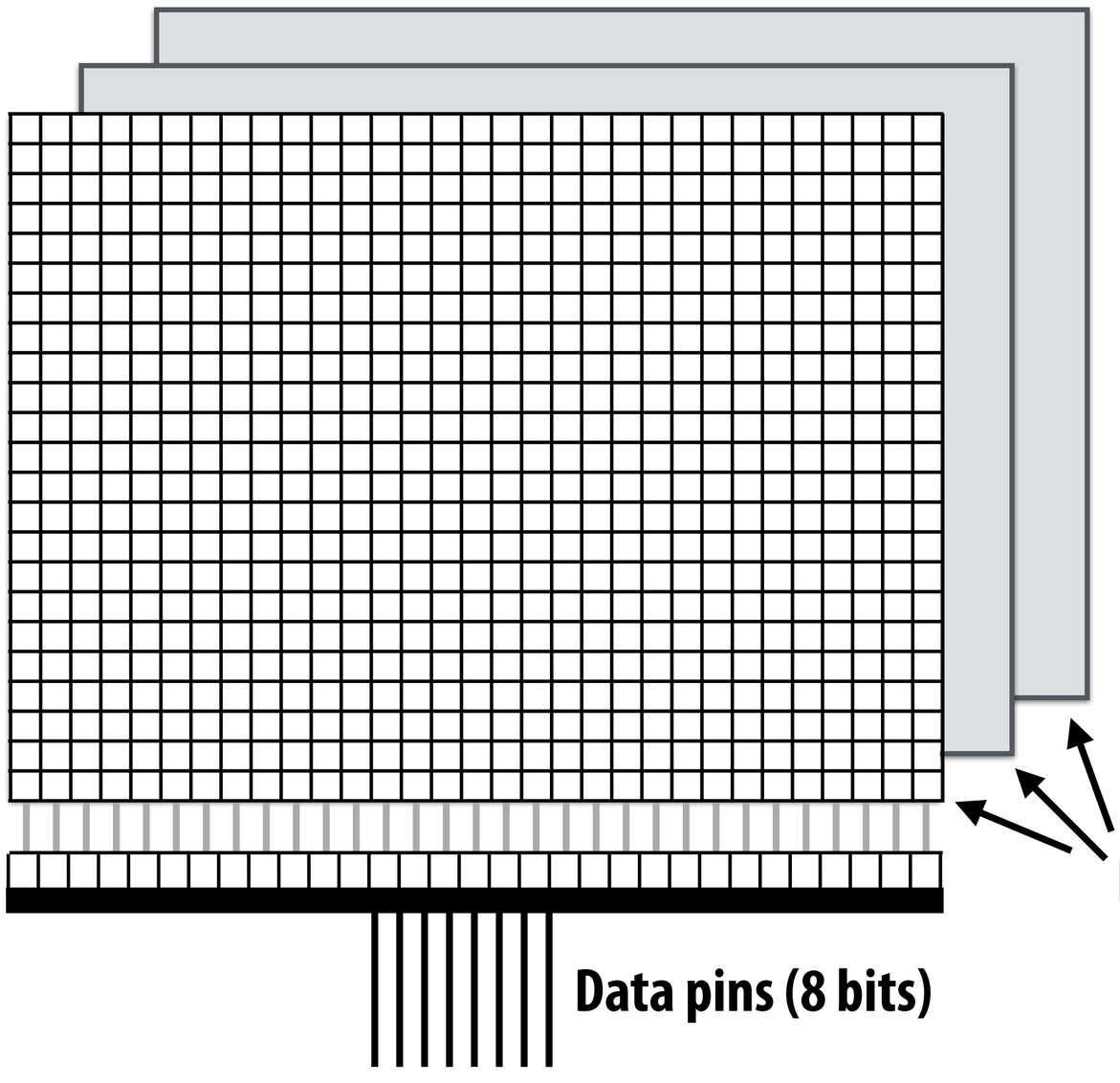


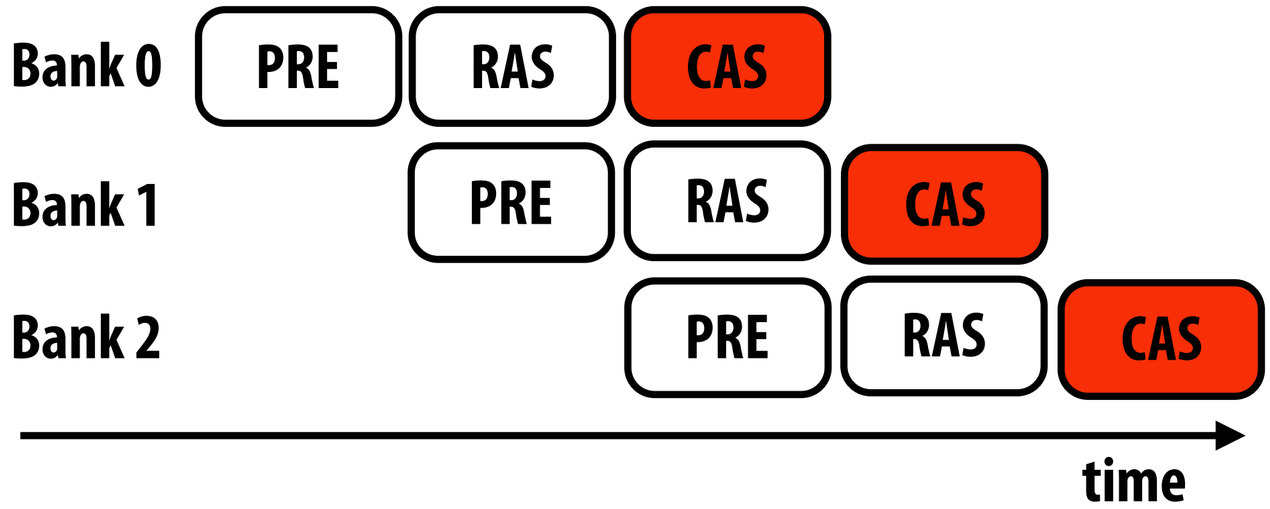
Idea: amortize latency over larger transfers

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

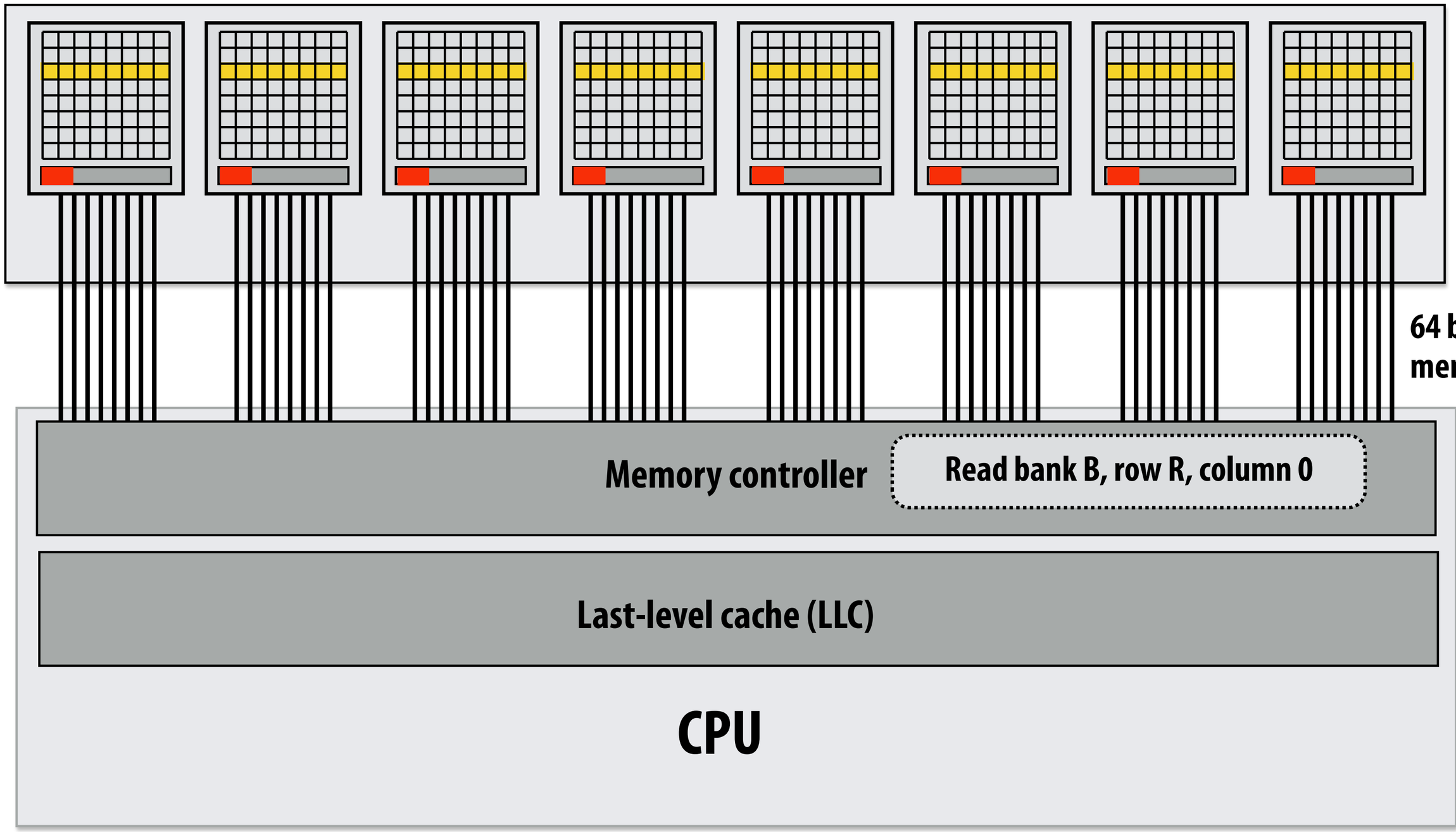




Organize multiple chips into a DIMM

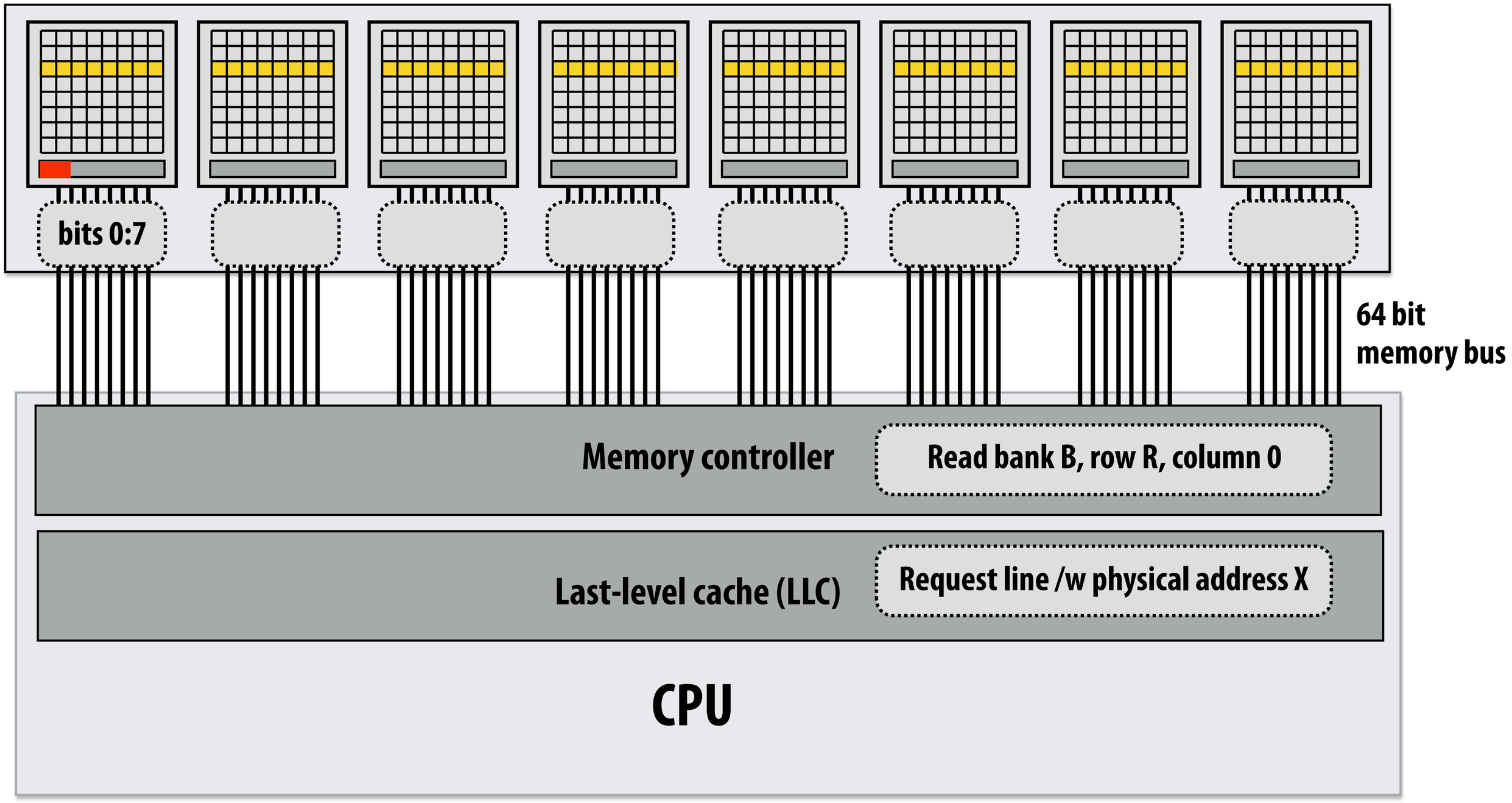
Example: Eight DRAM chips (64-bit memory bus)

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.



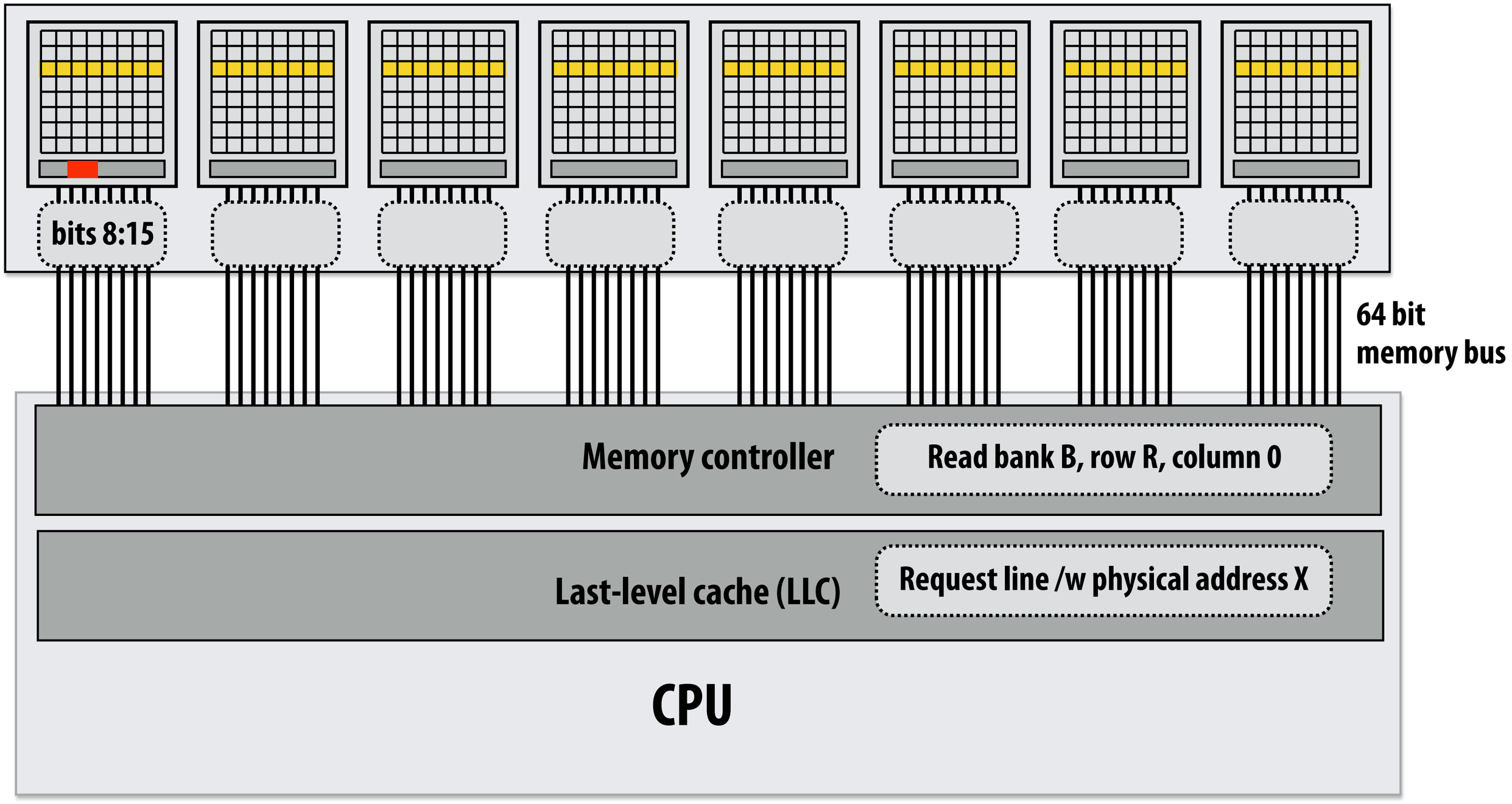
Reading one 64-byte (512 bit) cache line (the wrong way)

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



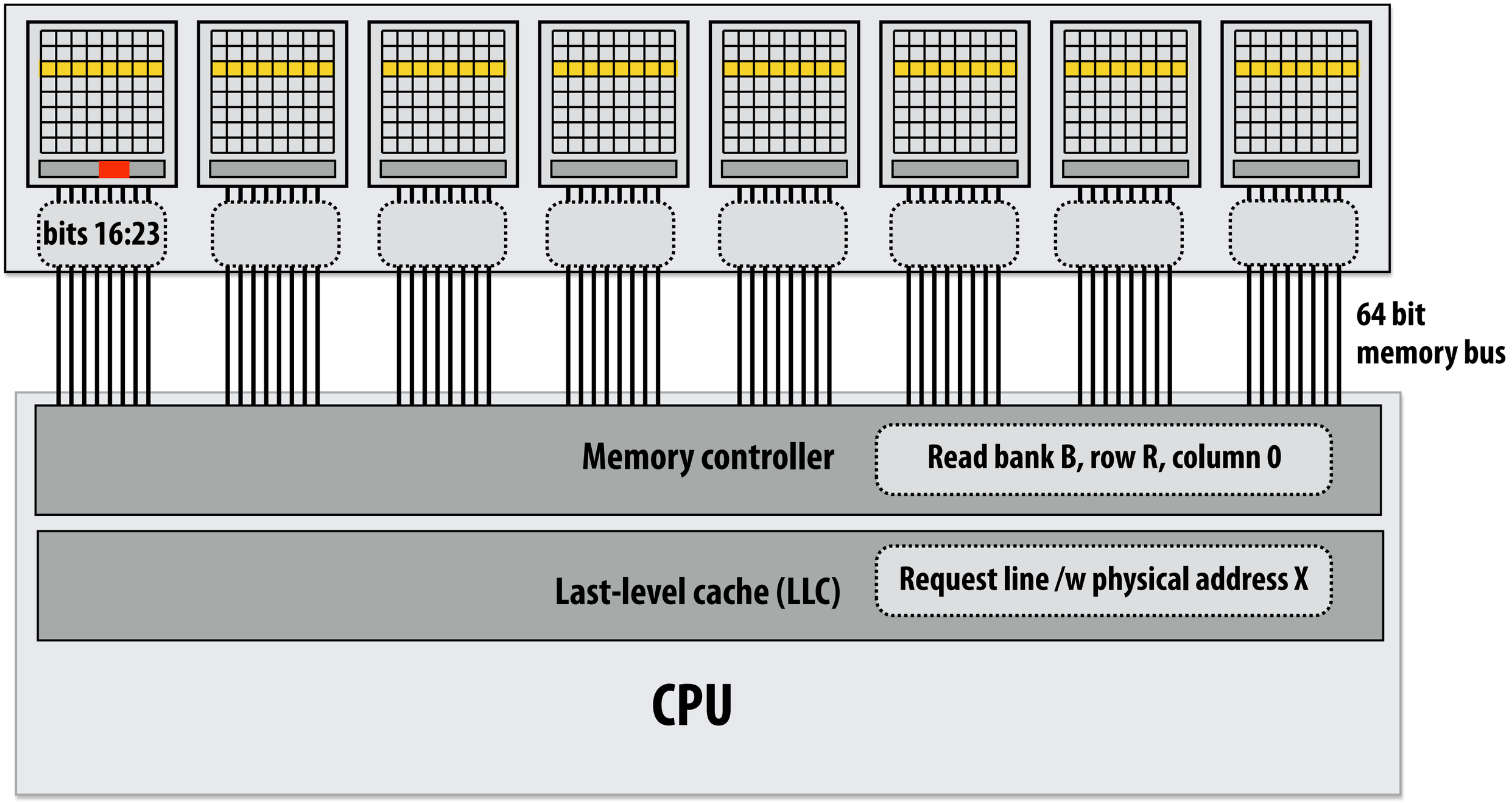
Reading one 64-byte (512 bit) cache line (the wrong way)

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

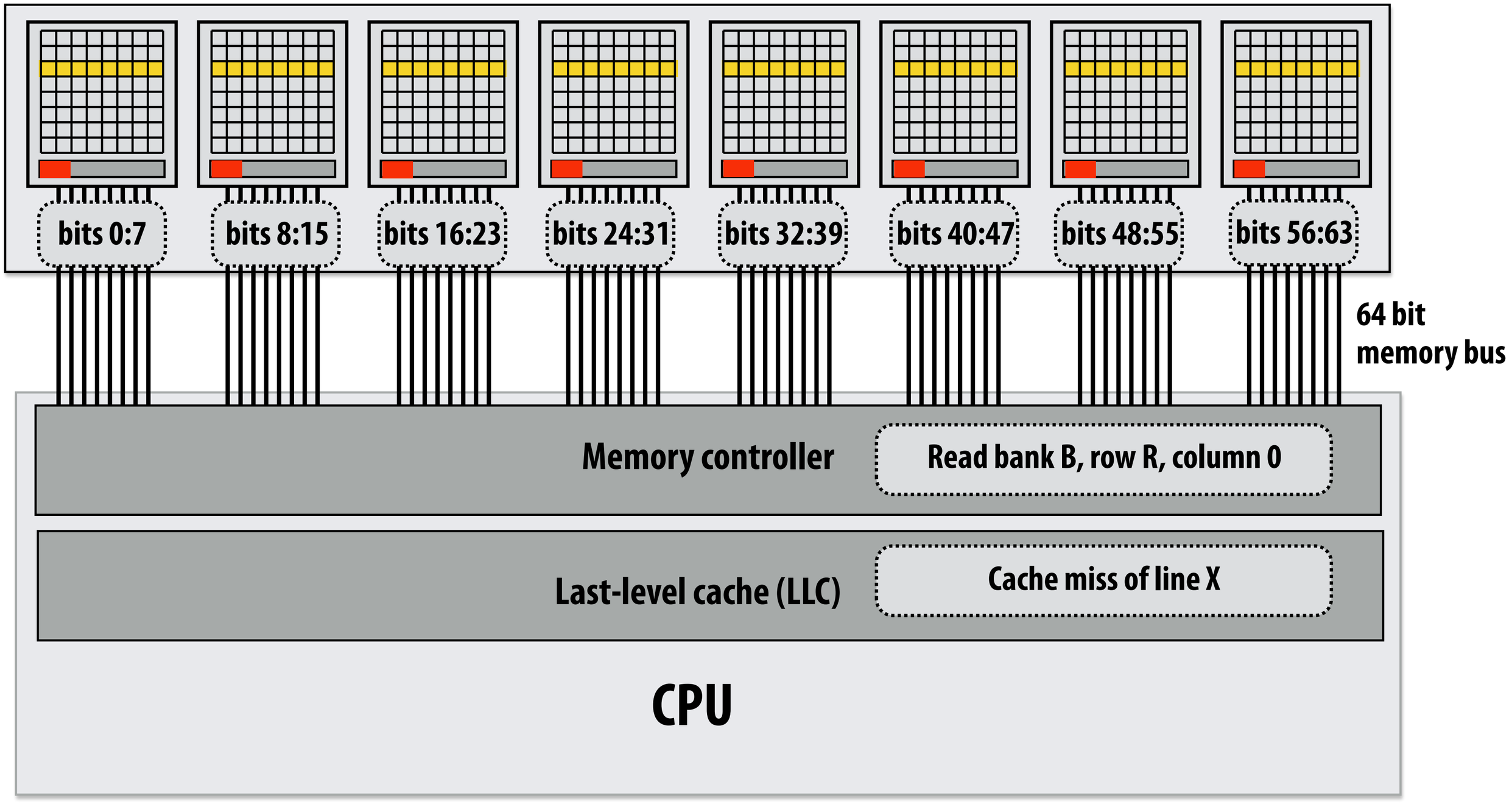


Reading one 64-byte (512 bit) cache line

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

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

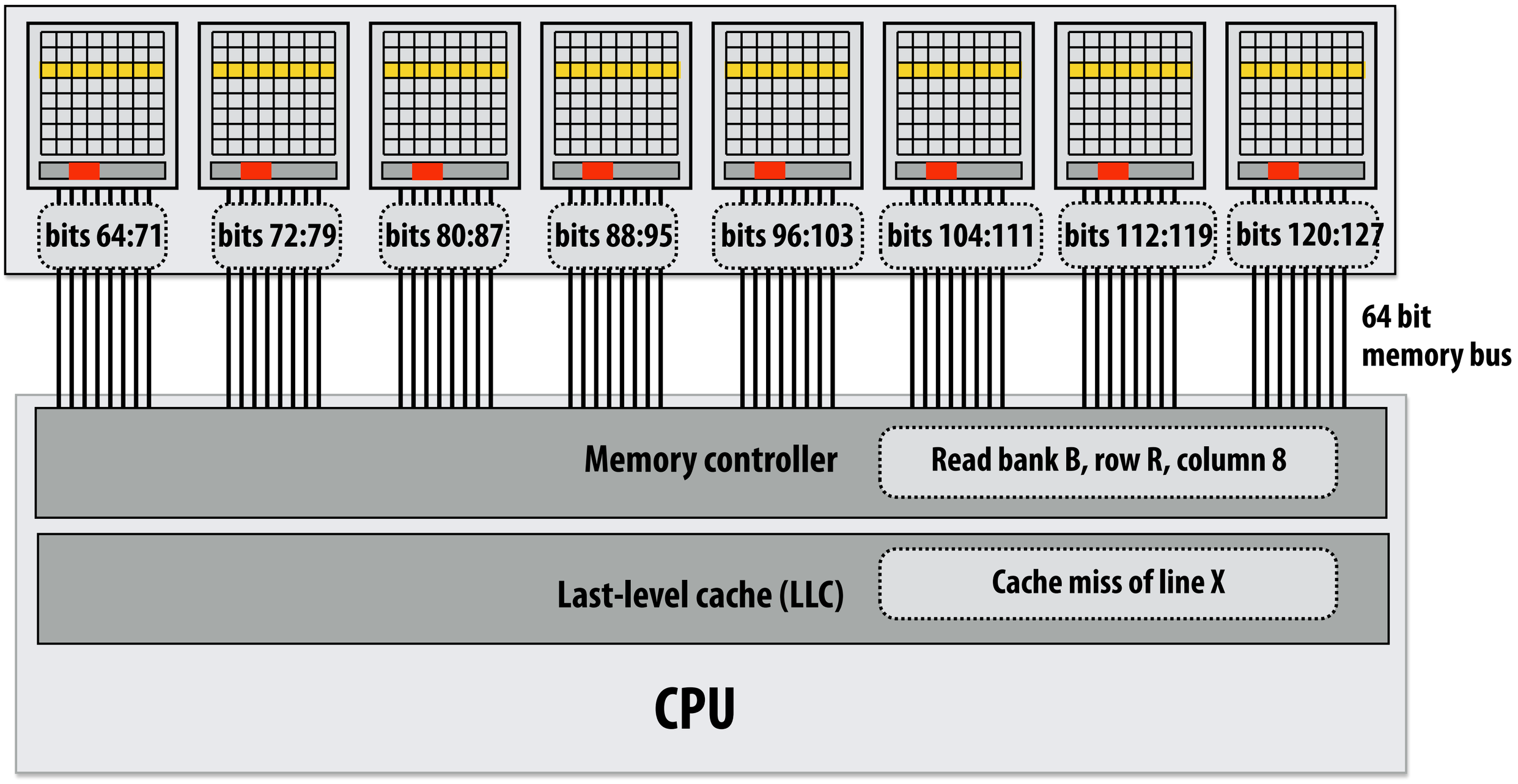
DRAM chips transmit first 64 bits in parallel



Reading one 64-byte (512 bit) cache line

DRAM controller requests data from new column *

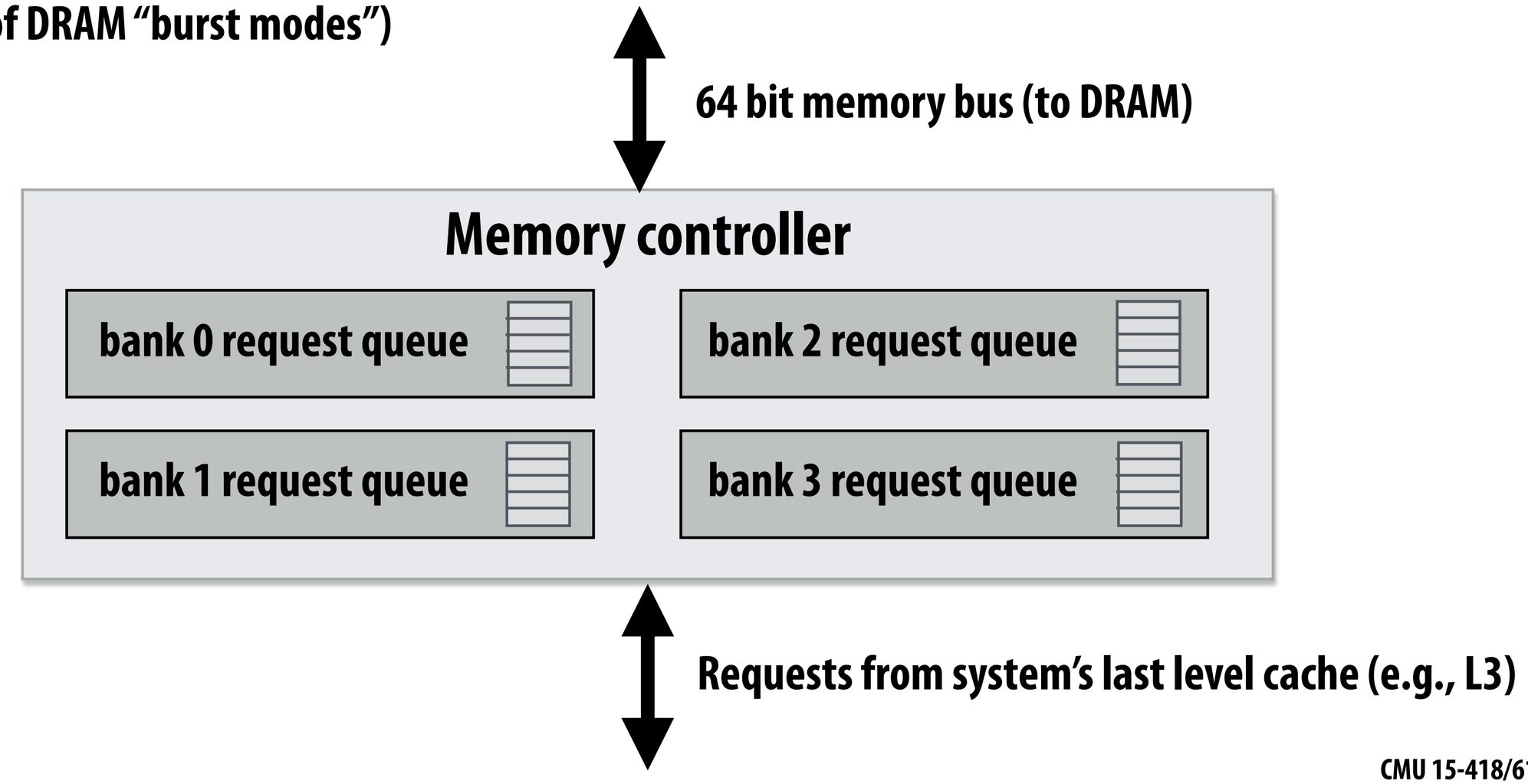
DRAM chips transmit next 64 bits in parallel



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

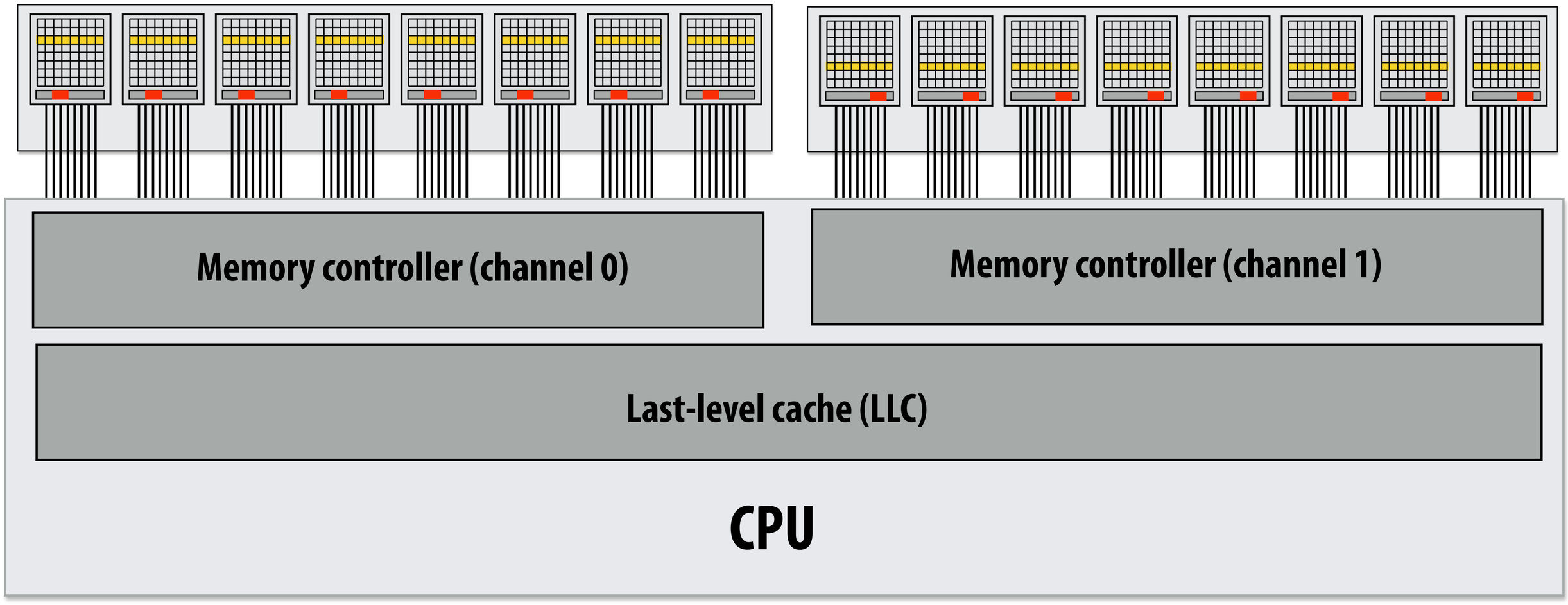
Memory controller is a memory request scheduler

- 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 (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



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 complexity in modern processor has moved into 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

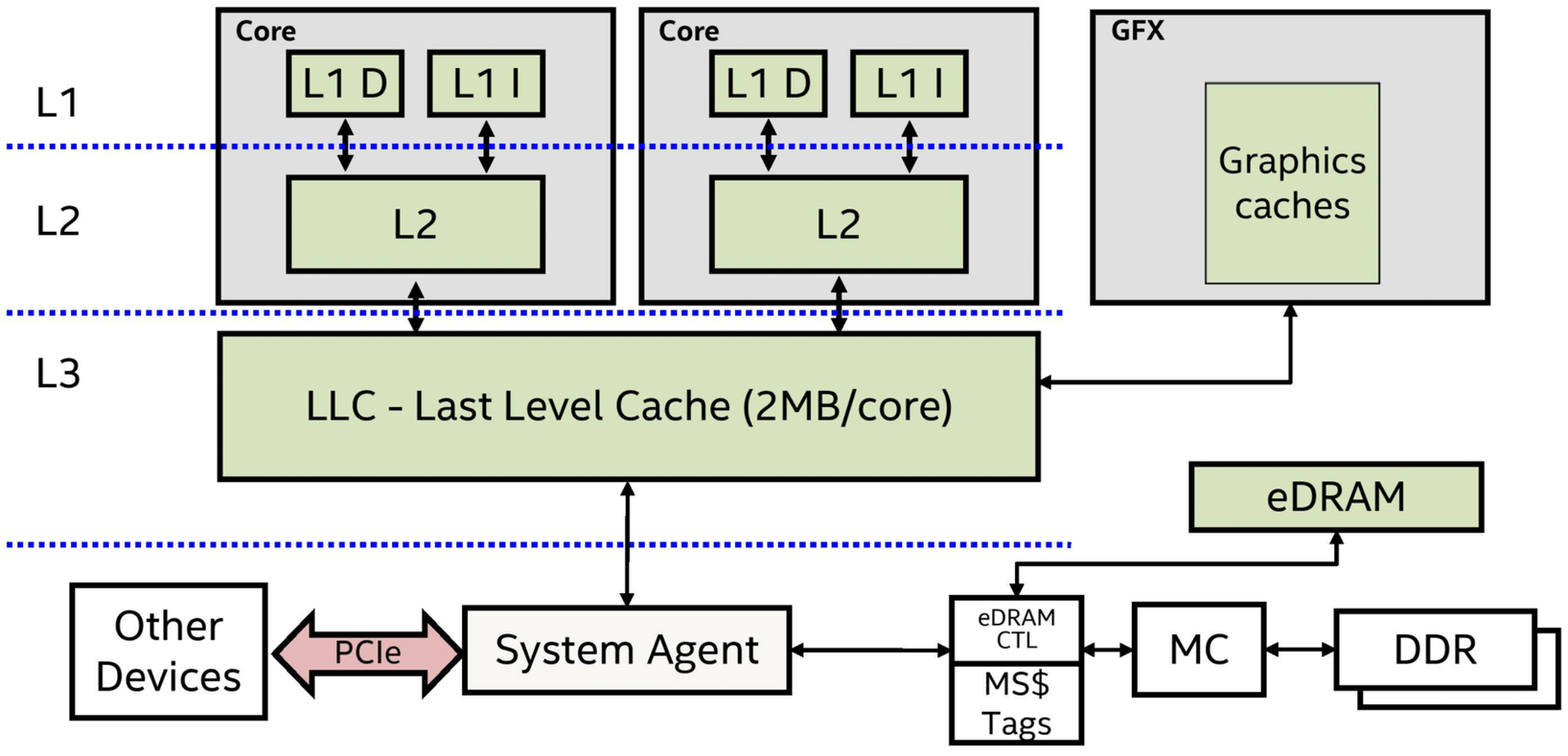
Decrease distance data must move: locate memory near processing (or processing near memory)

Think of logic near memory as yet another processing element in a modern parallel system.

Embedded DRAM (eDRAM): another level of the memory hierarchy

Offerings of Intel Broadwell/Skylake processors feature 128 MB of embedded DRAM (eDRAM) in the CPU package

- **50 GB/sec read + 50 GB/sec write**



IBM Power 7 server CPUs feature eDRAM

GPU in XBox 360 had 10 MB of embedded DRAM to store the frame buffer

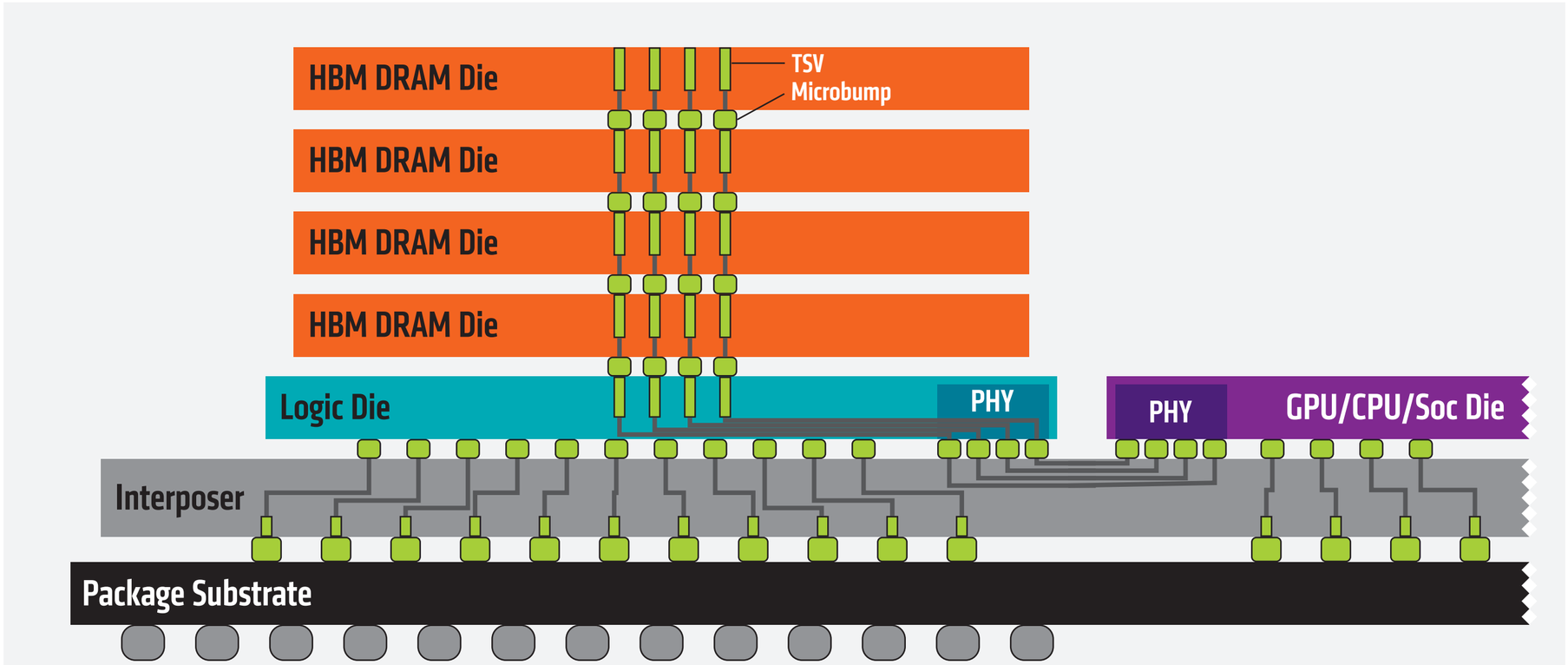
Attractive in mobile SoC setting

Image credit: Intel

Increase bandwidth, reduce power by chip stacking

Enabling technology: 3D stacking of DRAM chips

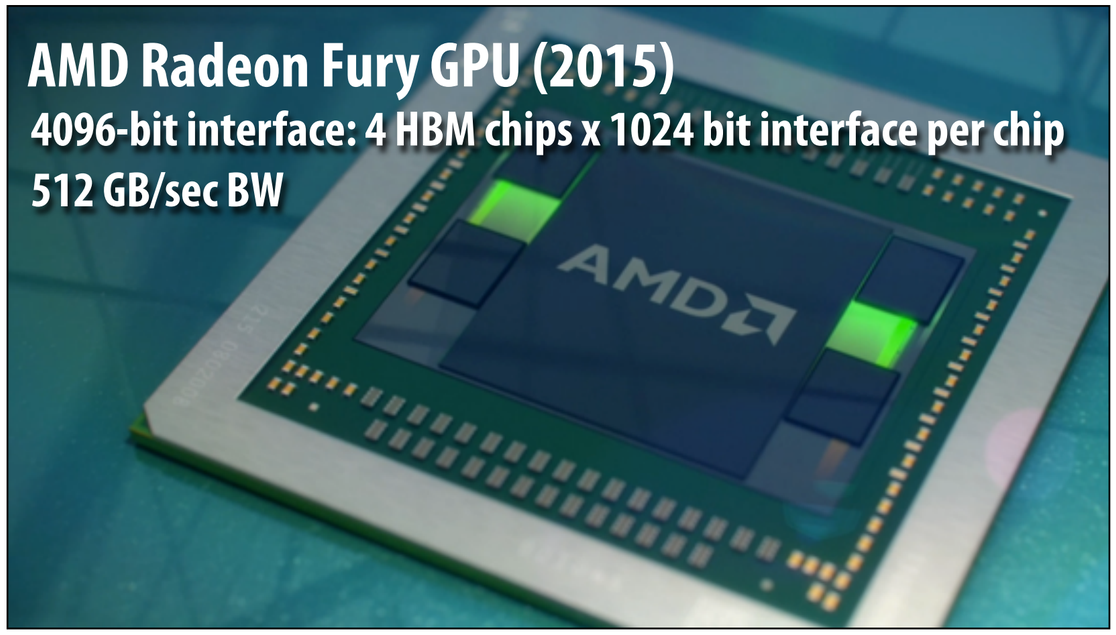
- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
- TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack "logic layer" is memory controller, manages requests from processor
- Silicon "interposer" serves as high-bandwidth interconnect between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

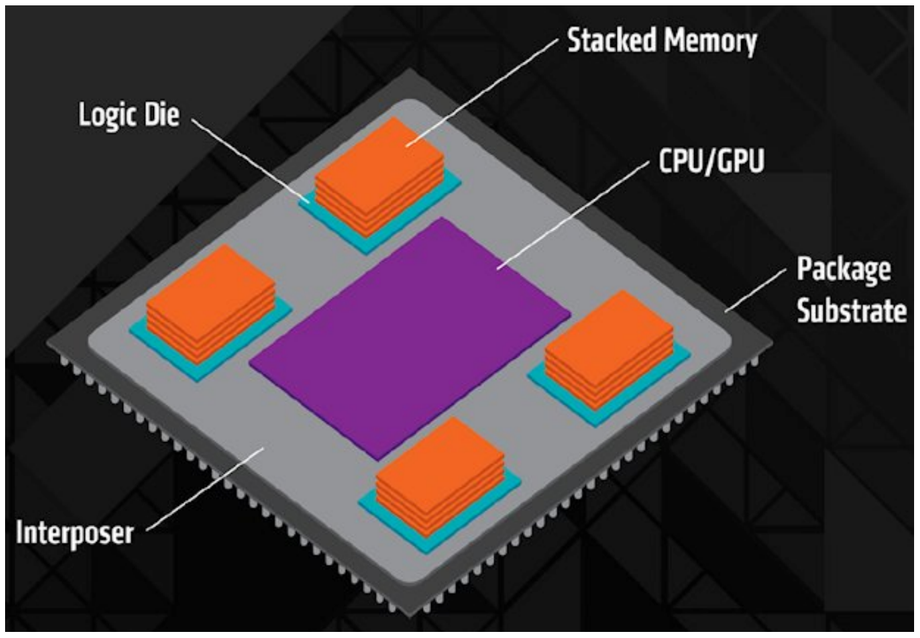


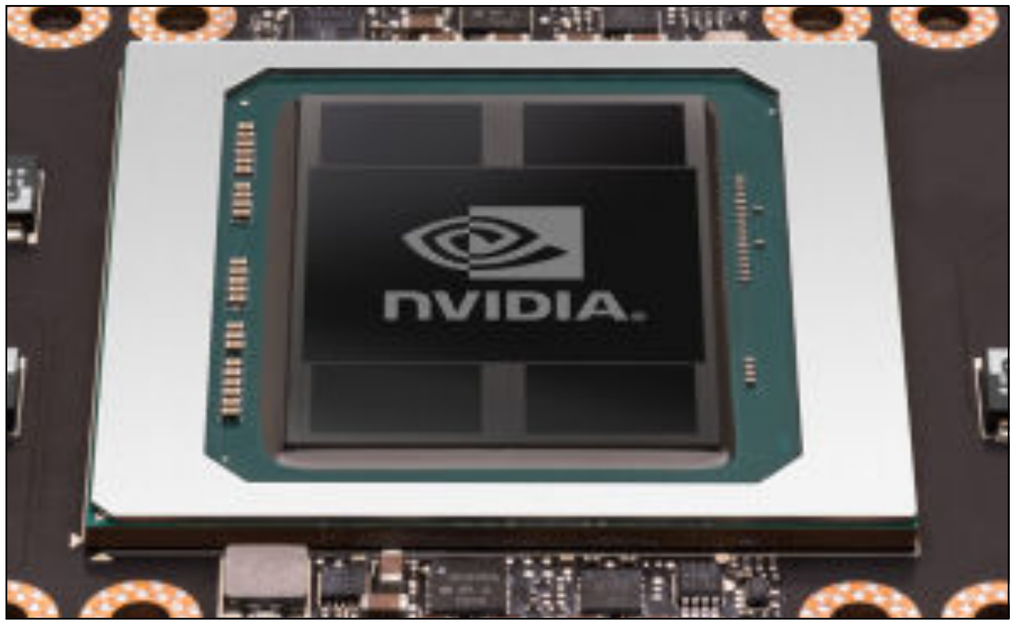
Technologies:
Micron/Intel Hybrid Memory Cube (HBC)
High-bandwidth memory (HBM) - 1024 bit interface to stack

Image credit: AMD

GPUs adopting HBM technologies







NVIDIA P100 GPU (2016)

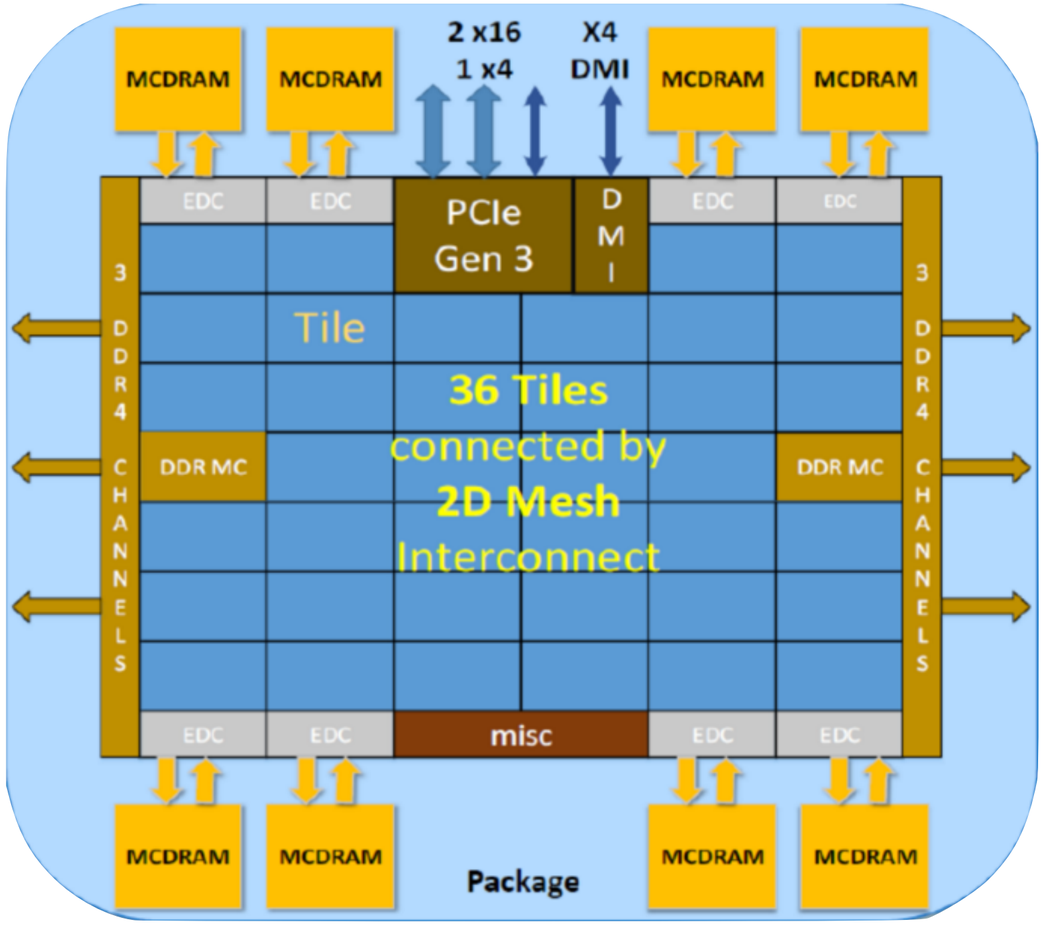
4096-bit interface: 4 HBM2 chips x 1024 bit interface per chip

720 GB/sec peak BW

4 x 4 GB = 16 GB capacity

Xeon Phi (Knights Landing) MCDRAM

- 16 GB in package stacked DRAM
- Can be treated as a 16 GB last level cache
- Or as a 16 GB separate address space ("flat mode")
- Intel's claims:
  - ~ same latency at DDR4
  - ~5x bandwidth of DDR4
  - ~5x less energy cost per bit transferred




```
// allocate buffer in MCDRAM ("high bandwidth" memory malloc)
float* foo = hbw_malloc(sizeof(float) * 1024);
```

Recall: reduce data movement by moving computation to the data

Laptop

Web Application Server

DB Server

DB Server

DB Server

DB Server

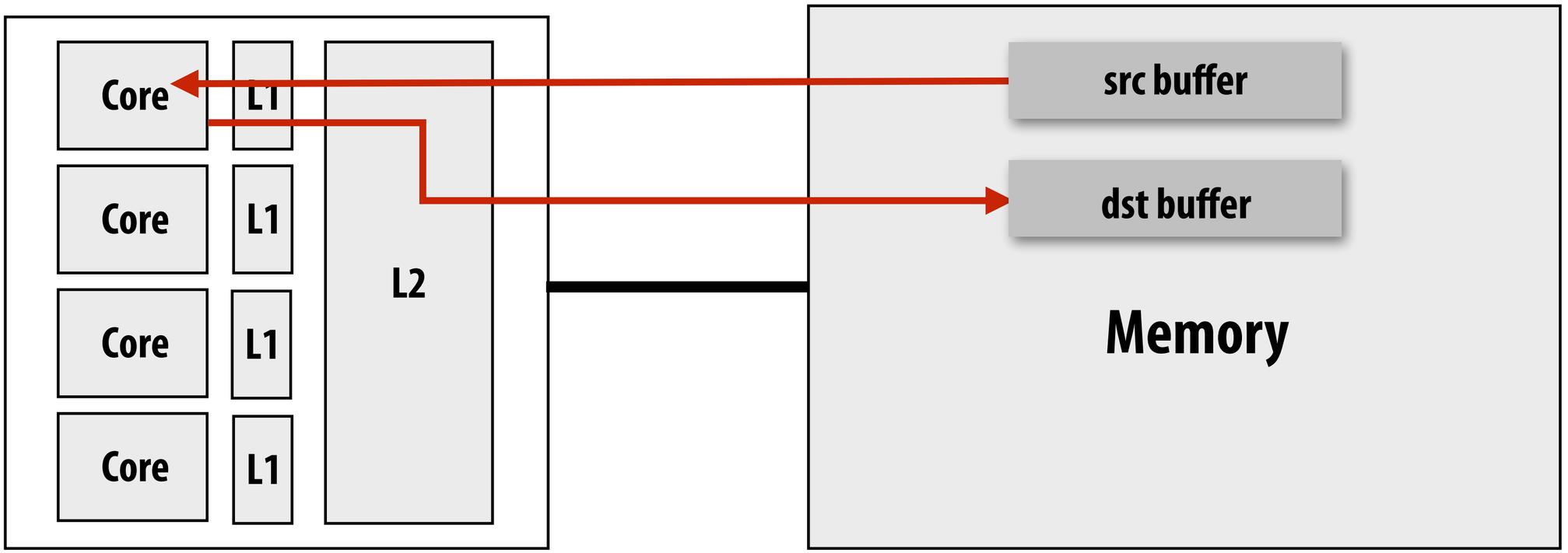
Consider a simple example of a web application that makes SQL query against large user database.

Would you transfer the database contents to the client so that the client can perform the query?

Example: memcpy = data movement through entire processor cache hierarchy

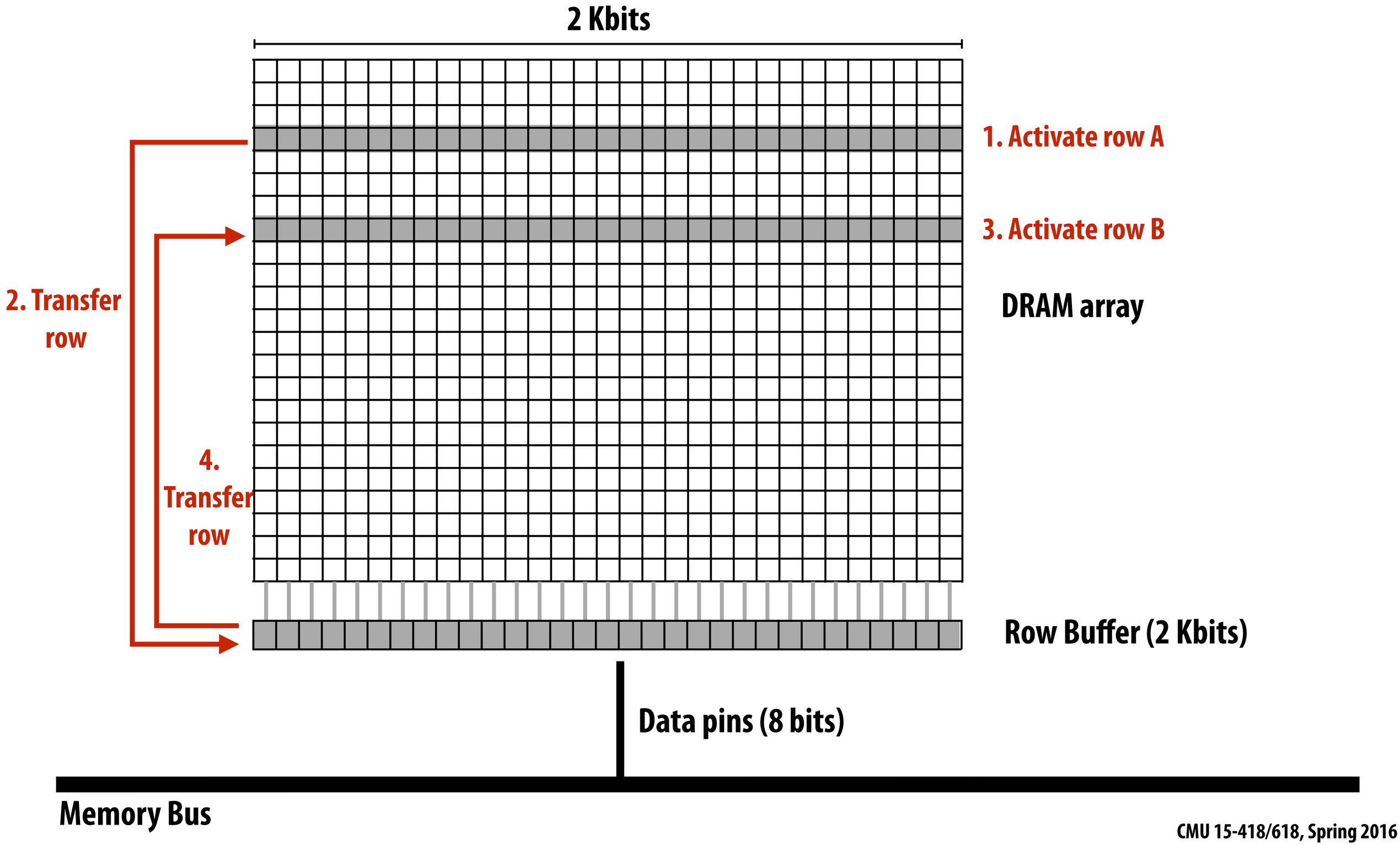
Bits move from DRAM, over bus, through cache hierarchy, into register file, and then retraces steps back out to DRAM

(and no computation is ever performed!)



Idea: perform copy without processor [Seshadri 13]

Modify memory system to support loads, stores, and bulk copy.



Hardware accelerated data compression

Upconvert/downconvert instructions

- **Example: `__mm512_extload_ps`**
  - **Load 8-bit values from memory, convert to 32-bit float representation for storage in register**

- **Very common functionality for graphics/image processing**

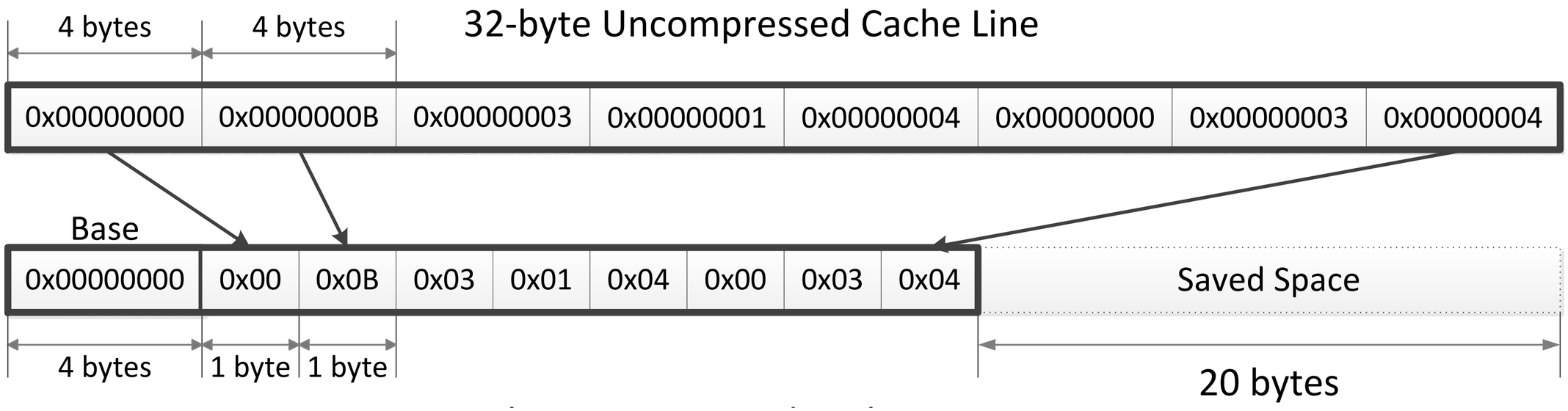
Cache compression

- Idea: increase cache's effective capacity by compressing data resident in cache
  - Idea: expend computation (compression/decompression) to save bandwidth
  - More cache hits = fewer transfers

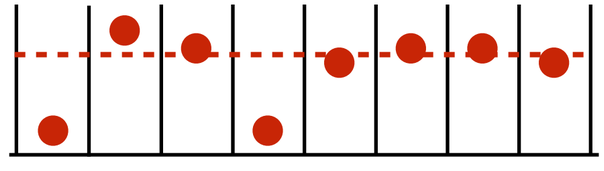
- A hardware compression/decompression scheme must
  - Be simple enough to implement in HW
  - Be fast: decompression is on critical path of loads
    - Cannot notably increase cache hit latency

One proposed example: BΔI compression [Pekhimenko 12]

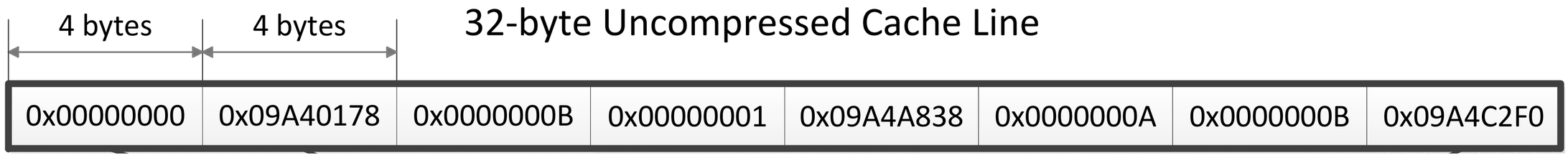
- **Observation: data that falls within cache line often has low dynamic range (use base + offset to encode chunks of bits in a line)**



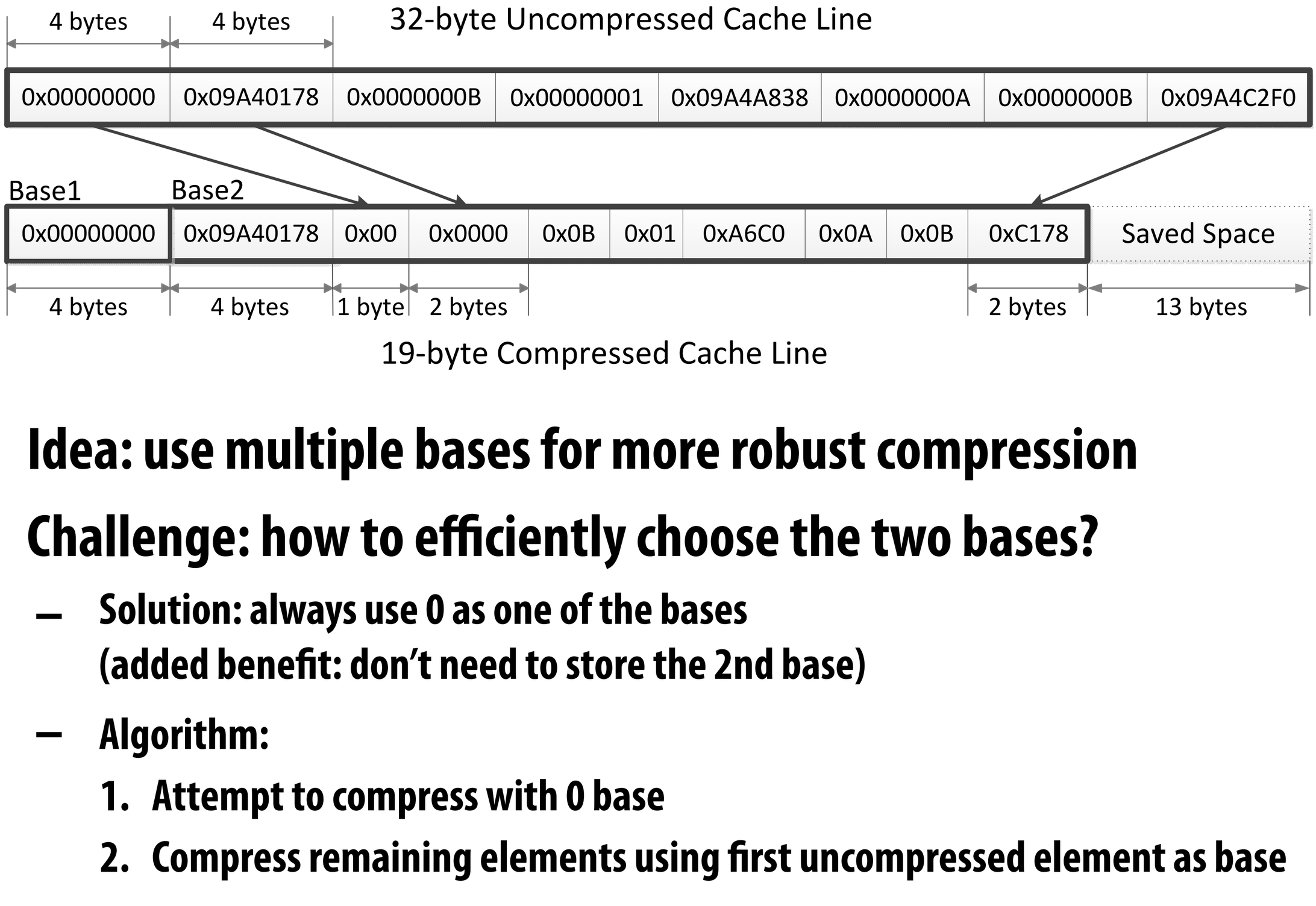
- **How does implementation quickly find a good base?**
  - **Use first word in line**
  - **Compression/decompression of line is data-parallel**



Does this pattern compress well?



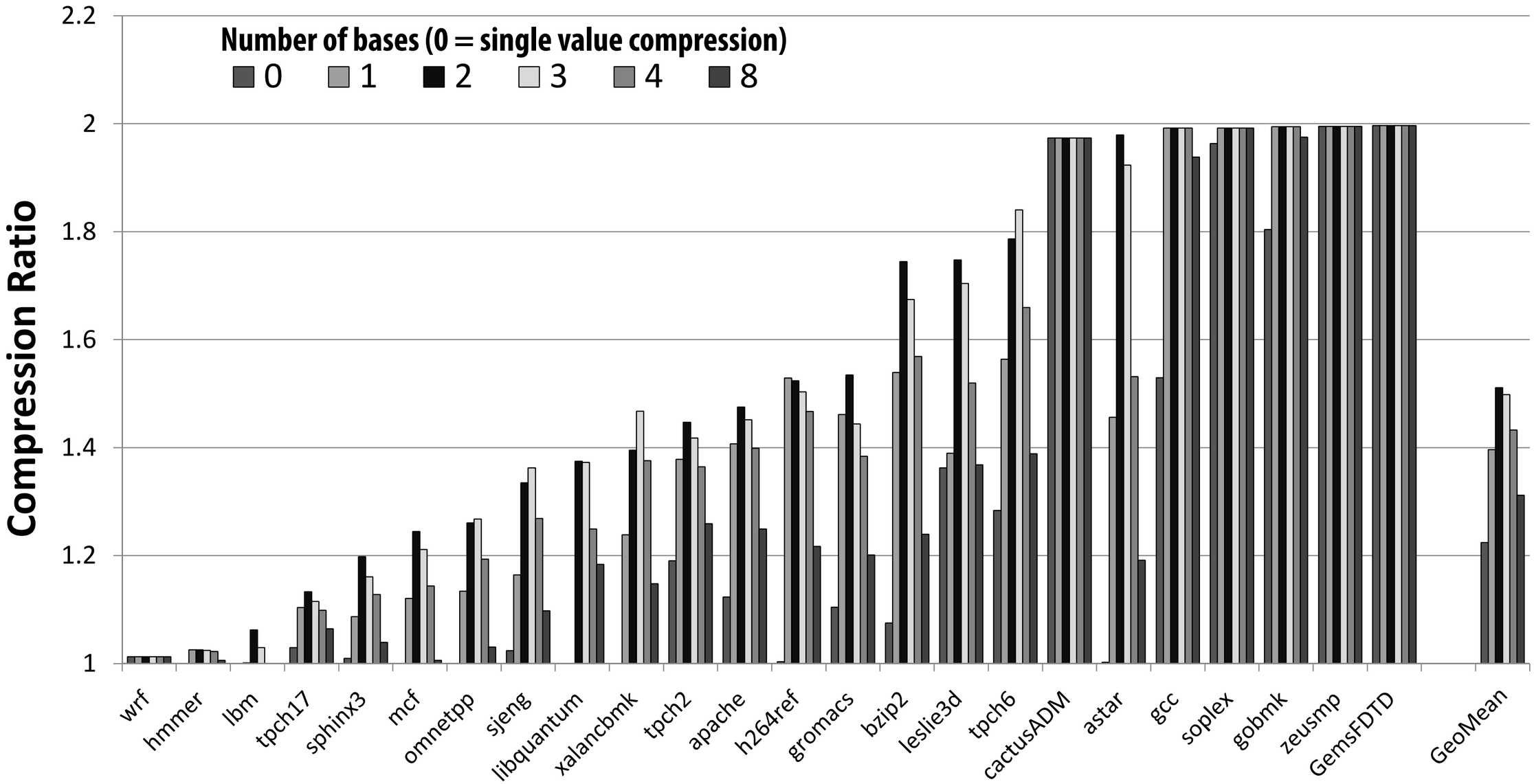
Does this pattern compress well?



- Idea: use multiple bases for more robust compression
- Challenge: how to efficiently choose the two bases?
  - Solution: always use 0 as one of the bases
    (added benefit: don't need to store the 2nd base)
  - Algorithm:
    1. Attempt to compress with 0 base
    2. Compress remaining elements using first uncompressed element as base

Effect of cache compression

[Pekhimenko 12]



On average: ~ 1.5x compression ratio

Translates into ~ 10% performance gain, up to 18% on cache sensitive workloads

Frame buffer compression in GPUs

- **All modern GPUs have hardware support for losslessly compressing frame buffer contents before transfer to/from memory**
  - On cache line load: transfer compressed data from memory and decompress into cache
  - On evict: compress cache line and only transfer compressed bits to memory
- **For example: anchor encoding (domain specific compression scheme)**
  - Compress 2D tiles of screen
  - Store value of "anchor pixel" p and compute Δx and Δy of adjacent pixels (fit a plane to the data)
  - Predict color of other pixels in tile based on offset from anchor
    - value(i,j) = $p + i\Delta x + j\Delta y$
  - Store "correction" c_i on prediction at each pixel
  - Consider encoding single channel image:
    - Store anchor at full resolution (e.g., 8 bits)
    - Store Δx, Δy, and correction at low bit depth

p →	Δx	c_0	c_1
Δy ↓	c_2	c_3	c_4
c_5	c_6	c_7	c_8
c_9	c_{10}	c_{11}	c_{12}

“Memory transaction elimination” in ARM GPUs

- Writing pixels in output image is a bandwidth-heavy operation
- Idea: skip output image write if it is unnecessary
  - Frame 1:
    - Render frame tile at a time
    - Compute hash for each tile on screen
  - Frame 2:
    - Render frame tile at a time
    - Before writing pixel values for tile, compute hash and see if tile is the same as last frame
      - If yes, skip write

Slow camera motion: 96% of writes avoided

Fast camera motion: ~50% of writes avoided

[Source: Tom Olson http://community.arm.com/groups/arm-mali-graphics/blog/2012/08/17/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus]

Summary: the memory wall 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, eDRAM)
  - Increase bandwidth (wider memory systems, 3D memory stacking)
  - Ongoing research in locating limited forms of computation "in" or near memory
  - Ongoing research in hardware accelerated compression
- General principles
  - Locate data storage near processor
  - Move computation to data storage
  - Data compression (trade-off extra computation for less data transfer)