Lecture 26:

Addressing the Memory Wall

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Cage the Elephant

Back Against the Wall

(Cage the Elephant)

“This song is for the cores out there that are starving. Cores are hurting, man.“

- Matt Schultz

Saying it once again: moving data is costly!

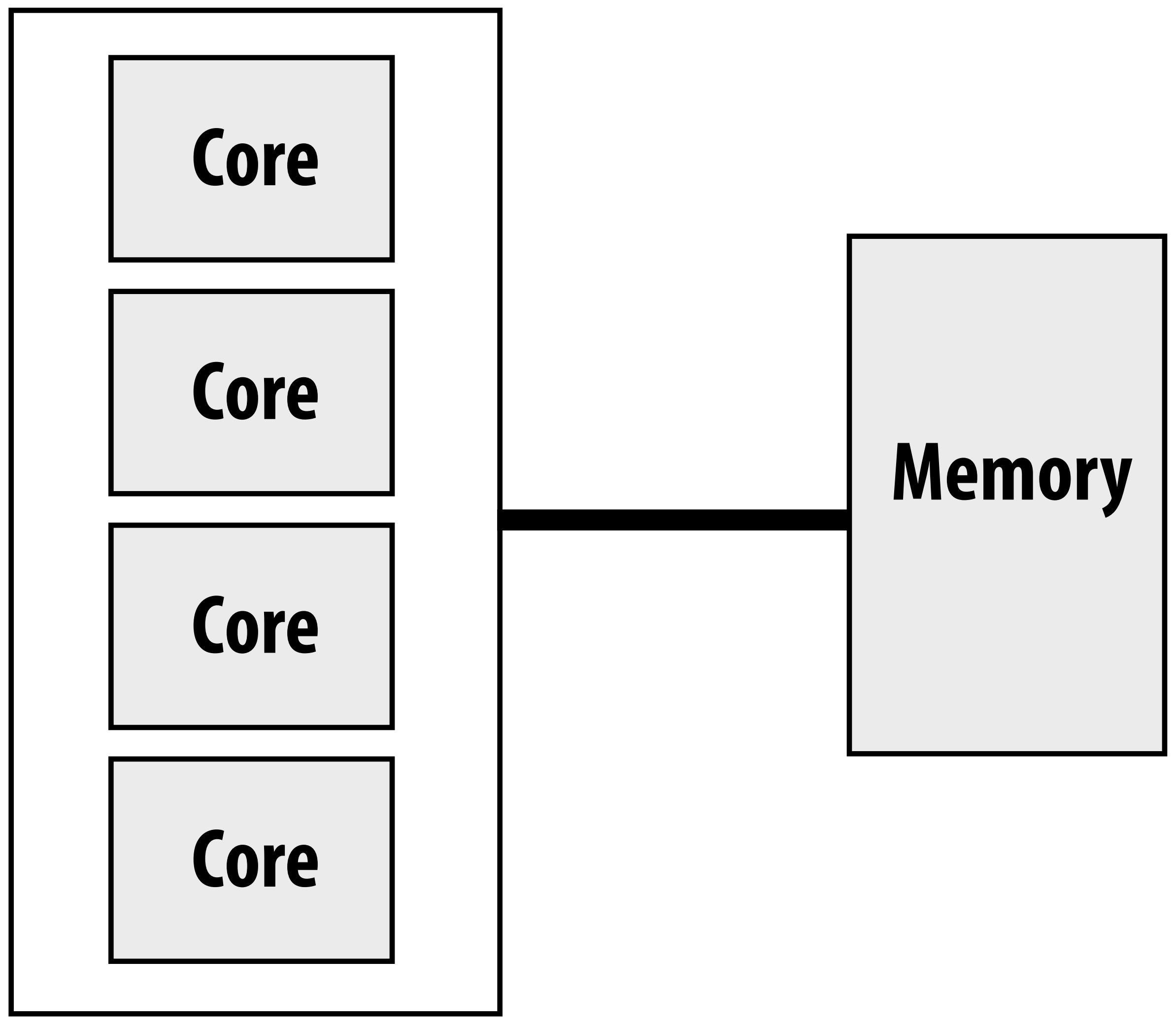
Limits program performance

Multiple processors…

= higher overall rate of memory requests

= need for more bandwidth

(result: bandwidth-limited execution)



High energy cost

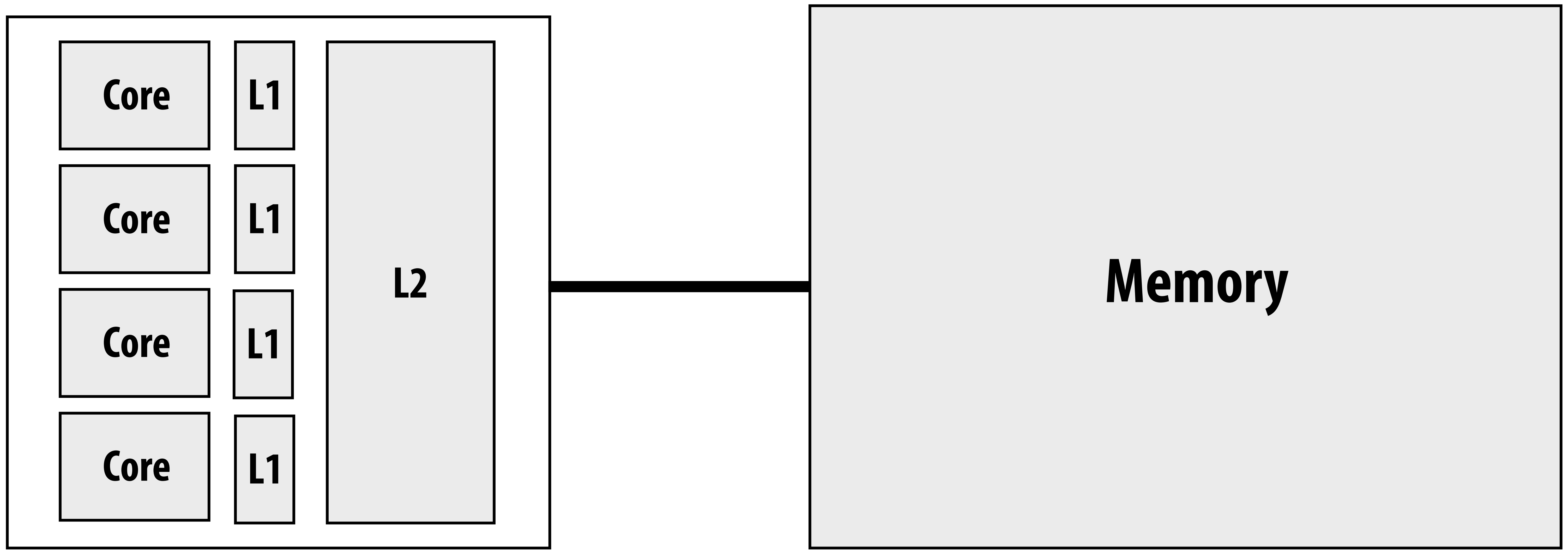
Recall "rough ballpark" numbers from heterogeneity lecture:

~ 20 pJ for an floating-point math op

~1000 pJ to load 64 bits from LPDDR memory

Well written programs exploit locality to avoid redundant transfers

(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 <u>locality</u> can reuse data in local memories
- Software optimization technique:
  - Structure order of computation so that after loading into cache, data is accessed it many times before evicting it
- Performance-aware programmers go to great effort to improve the cache locality of programs
  - What are good examples from this class?

Example 1: improving temporal locality by fusing loops

(recall this slide from the performance optimization lecture)

```
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, *tmp;

// assume arrays are allocated here

// compute D = (A + B) * C
add(n, A, B, tmp);
mult(n, tmp, C, D);
```

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

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

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

Three loads, one store per 2 math ops (arithmetic intensity = 1/2)

Another example of fusing loops

(recall this slide from the Halide lecture)

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

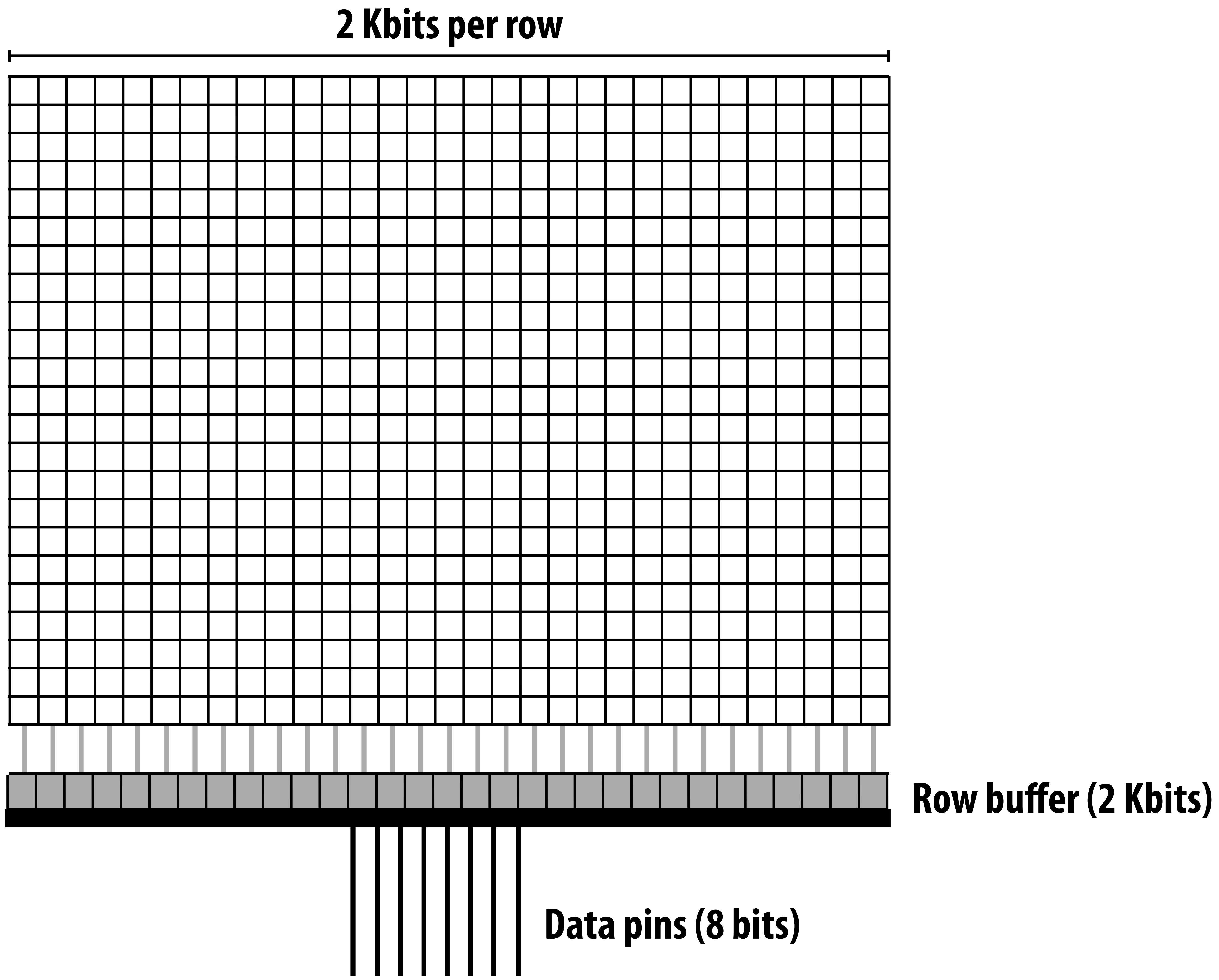
Modified iteration order:
256x32 block-major iteration
(to maximize cache hit rate)

two passes fused into one:
`tmp` data read from cache

Accessing DRAM

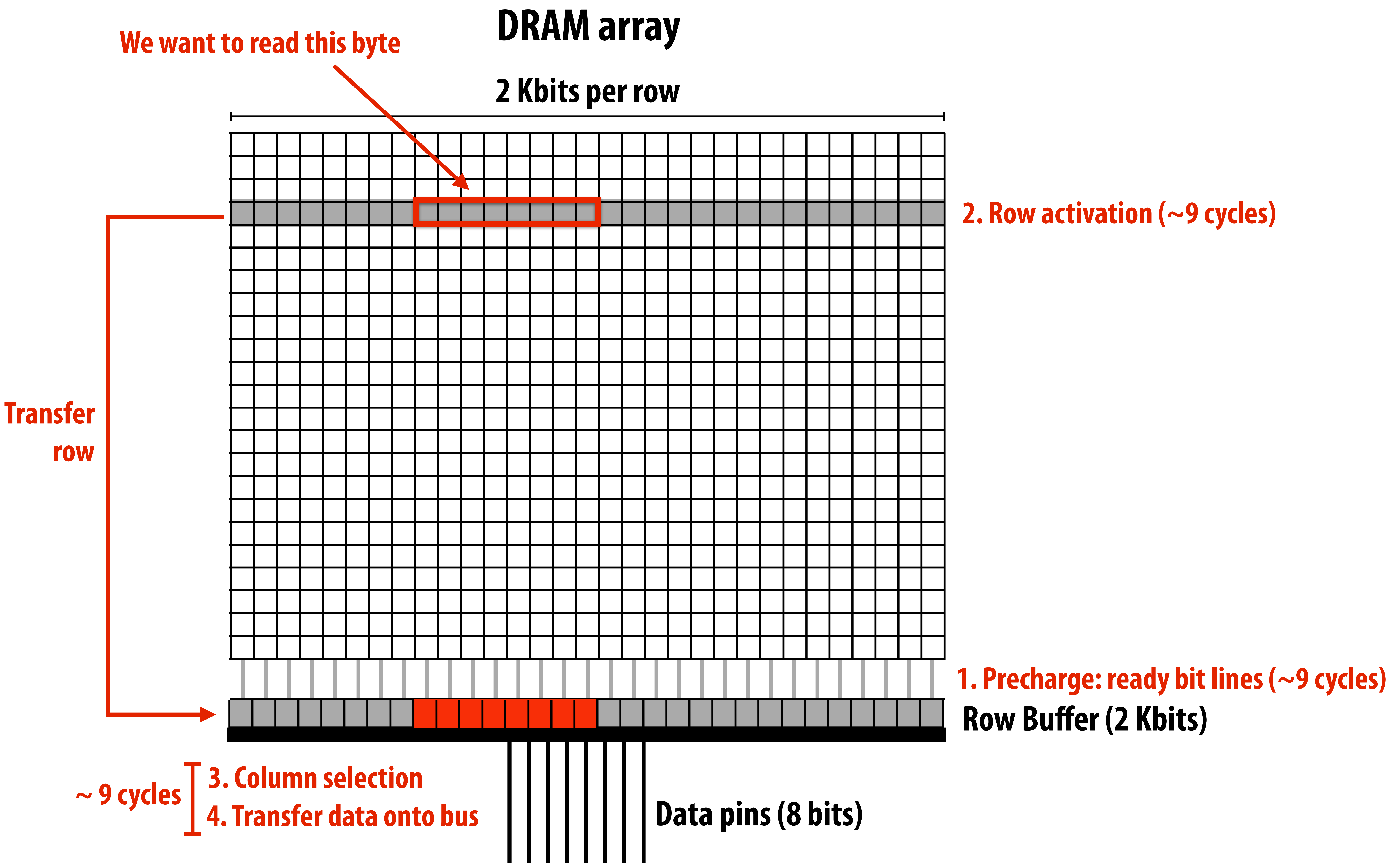
DRAM array

1 transistor + capacitor per bit



DRAM operation (load one byte)

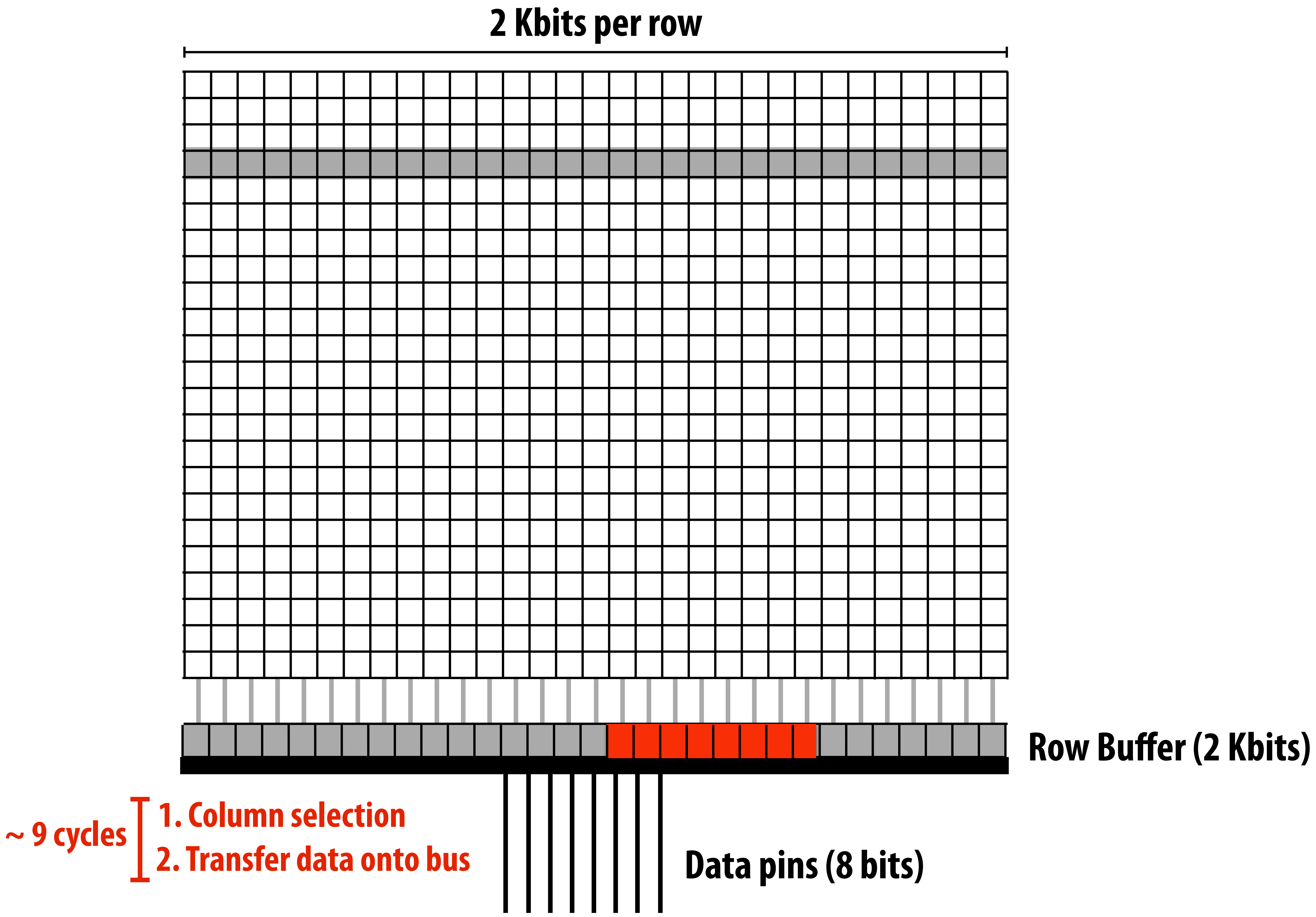
Estimated latencies are in units of memory clocks: DDR3-1600 (Kayvon's laptop)



DRAM array
We want to read this byte
2 Kbits per row
2. Row activation (~9 cycles)
Transfer row
1. Precharge: ready bit lines (~9 cycles)
Row Buffer (2 Kbits)
~ 9 cycles
3. Column selection
4. Transfer data onto bus
Data pins (8 bits)

Load next byte from active row

Lower latency: can skip precharge and row activation steps



DRAM access latency is not fixed

- **Best case latency: read from active row**
  - Column access time (CAS)

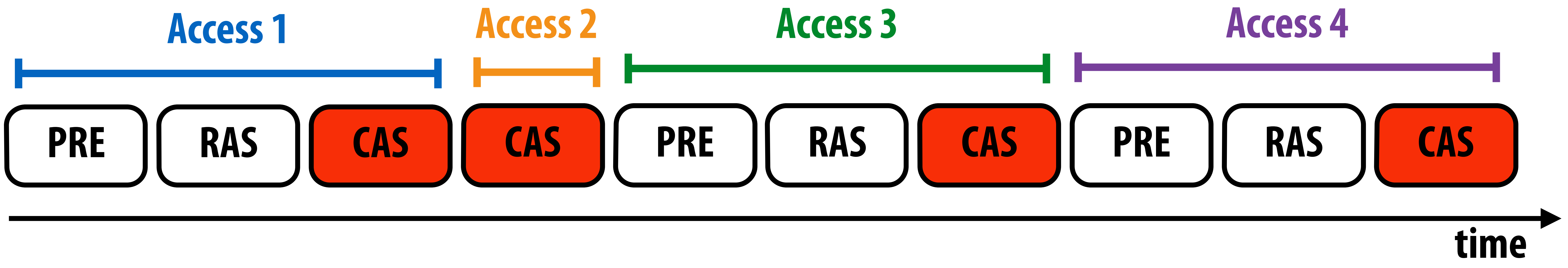
- **Worst case latency: bit lines not ready, read from new row**
  - Precharge (PRE) + row activate (RAS) + column access (CAS)

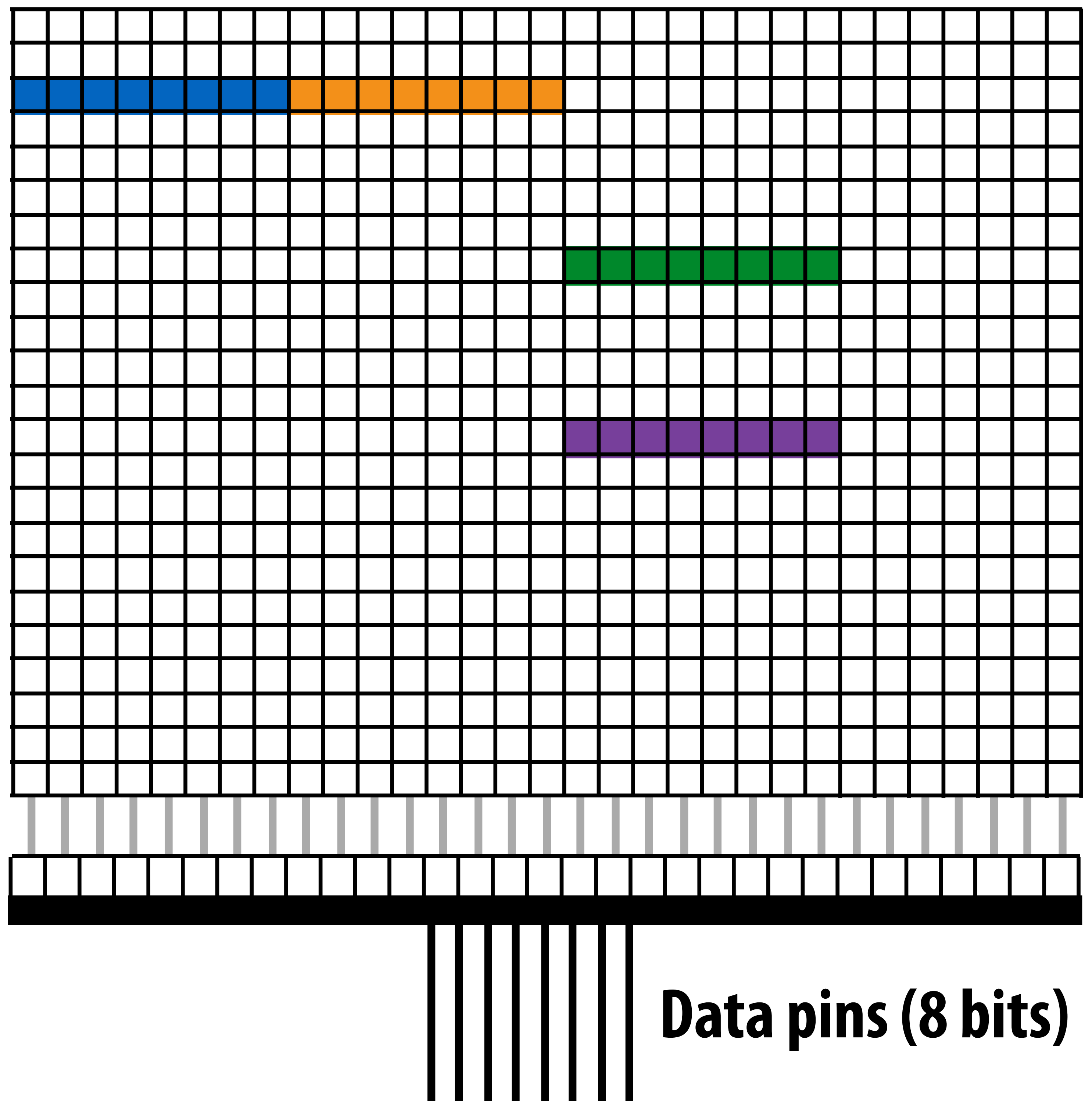
Precharge readies bit lines and writes row-buffer contents back into DRAM array (read was destructive)

- **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

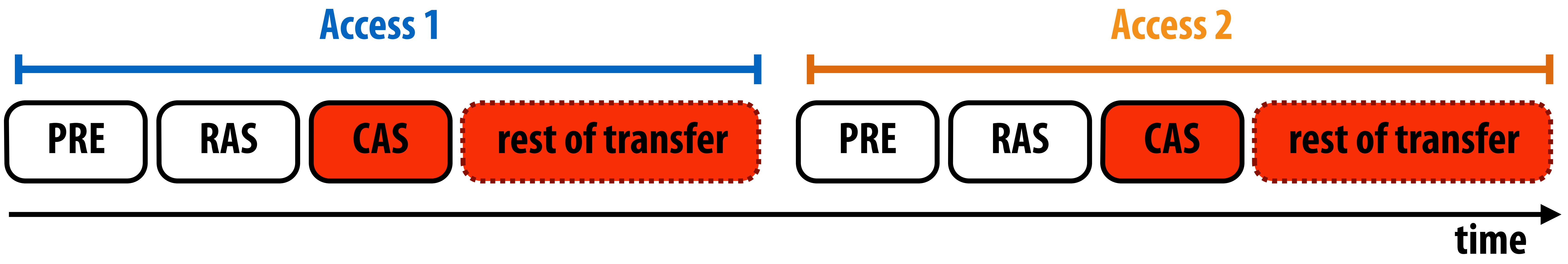


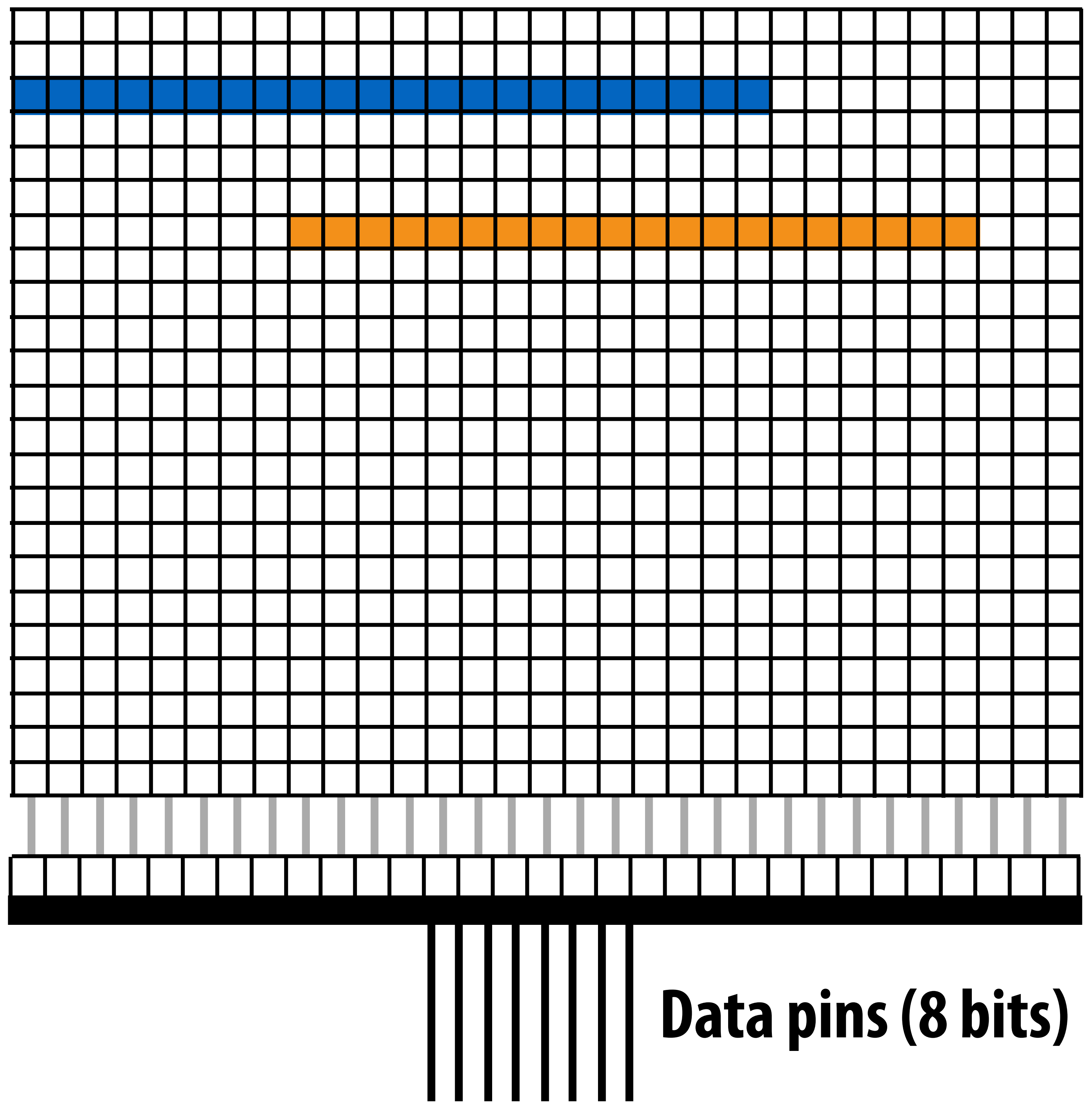


Data pins in use only a small fraction of time (red = data pins busy)

Very bad since they are the scarcest resource!

DRAM burst mode



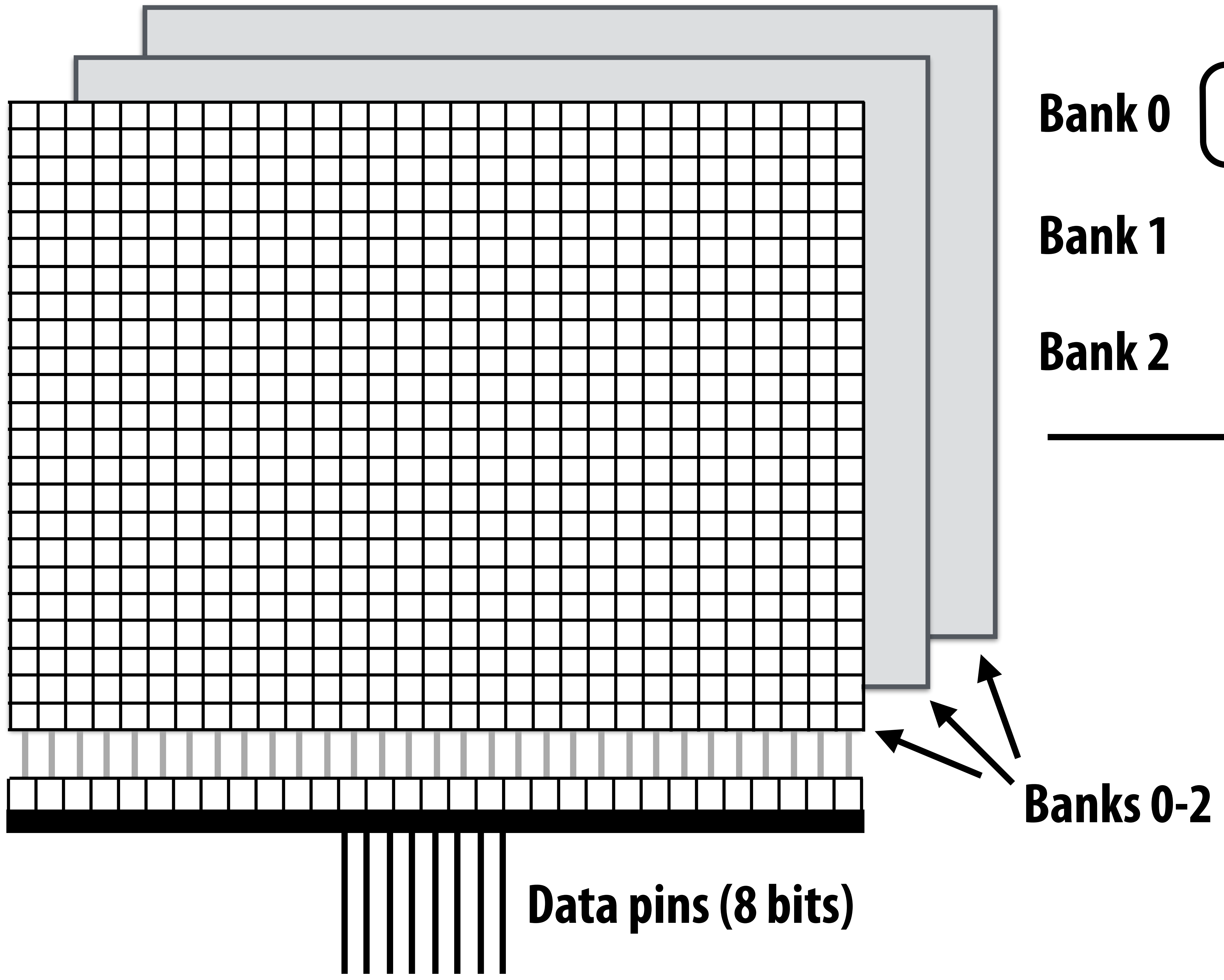


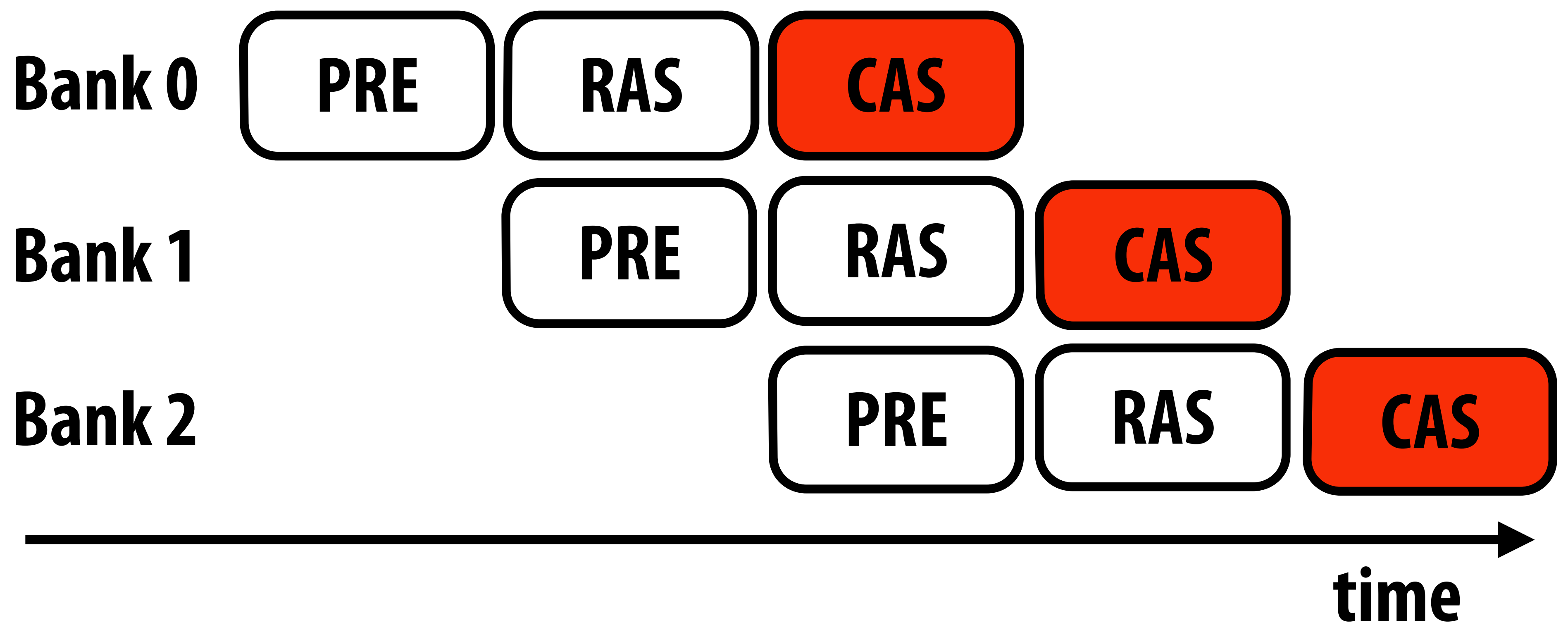
Idea: amortize latency over larger transfers

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

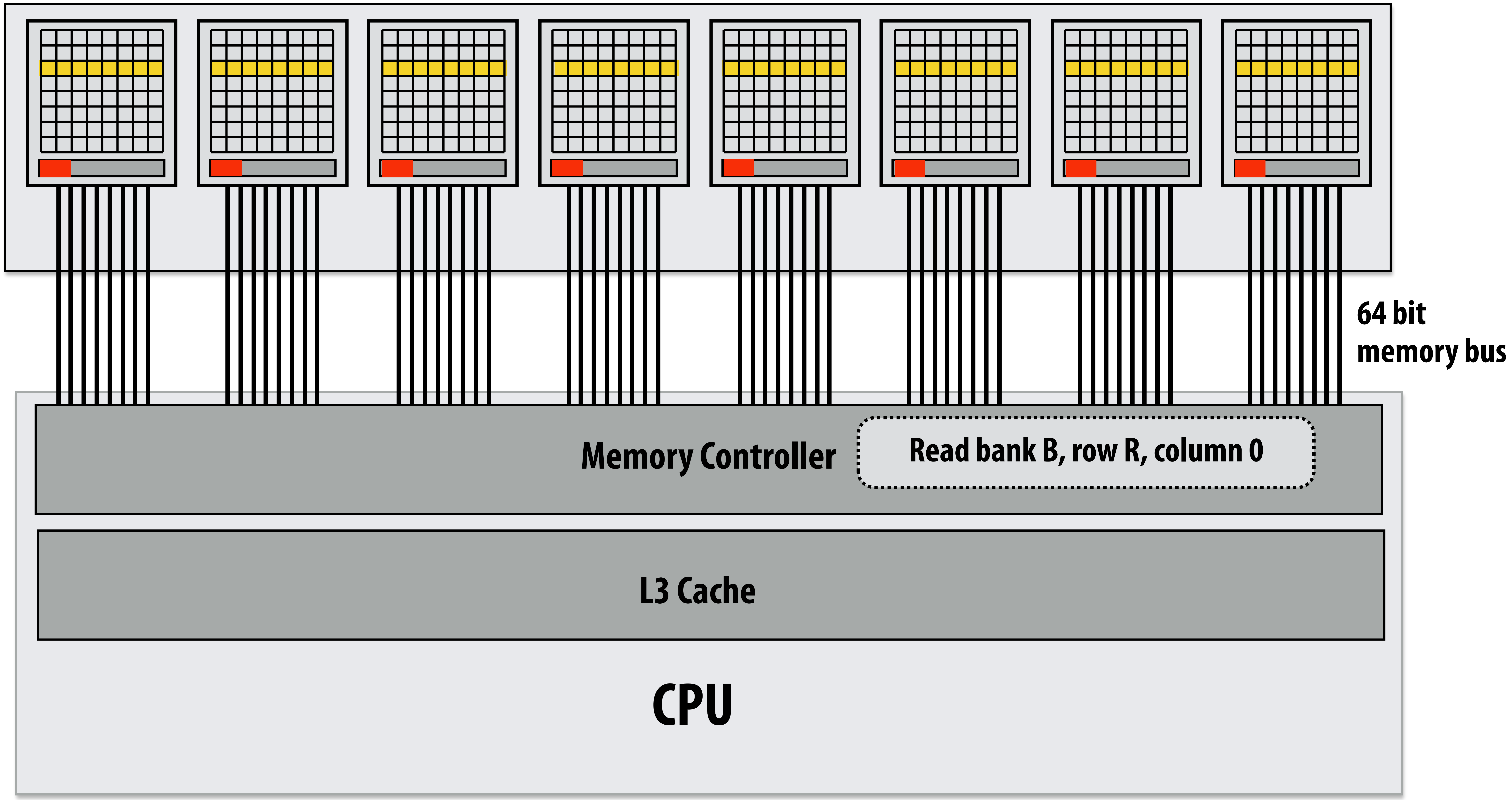




Organize multiple chips into a DIMM

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

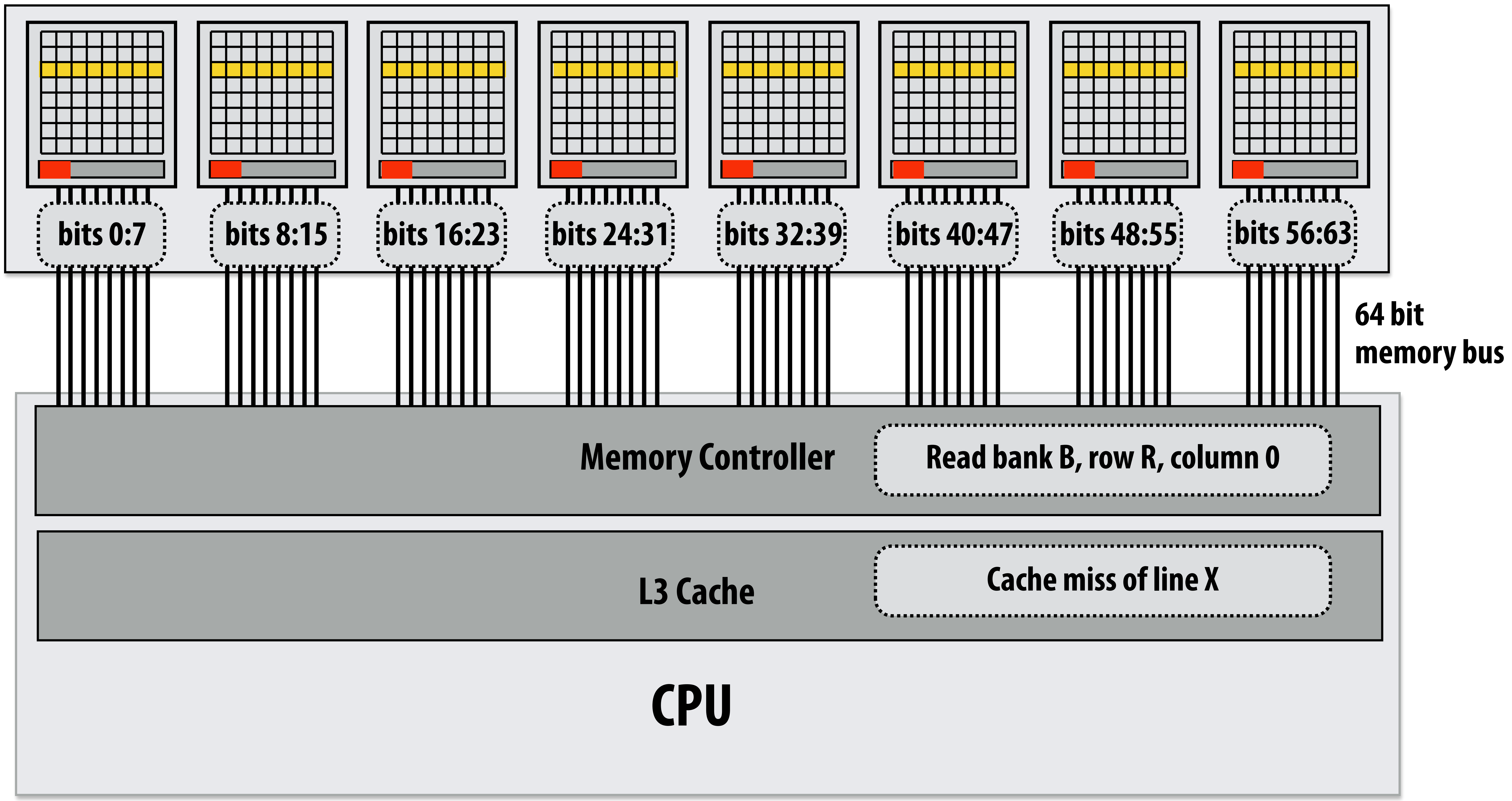
Note: appears as a single higher capacity, wider interface DRAM module to the memory controller. Higher aggregate bandwidth, but minimum transfer granularity is now 64 bits.



Reading one 64-byte (512 bit) cache line

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

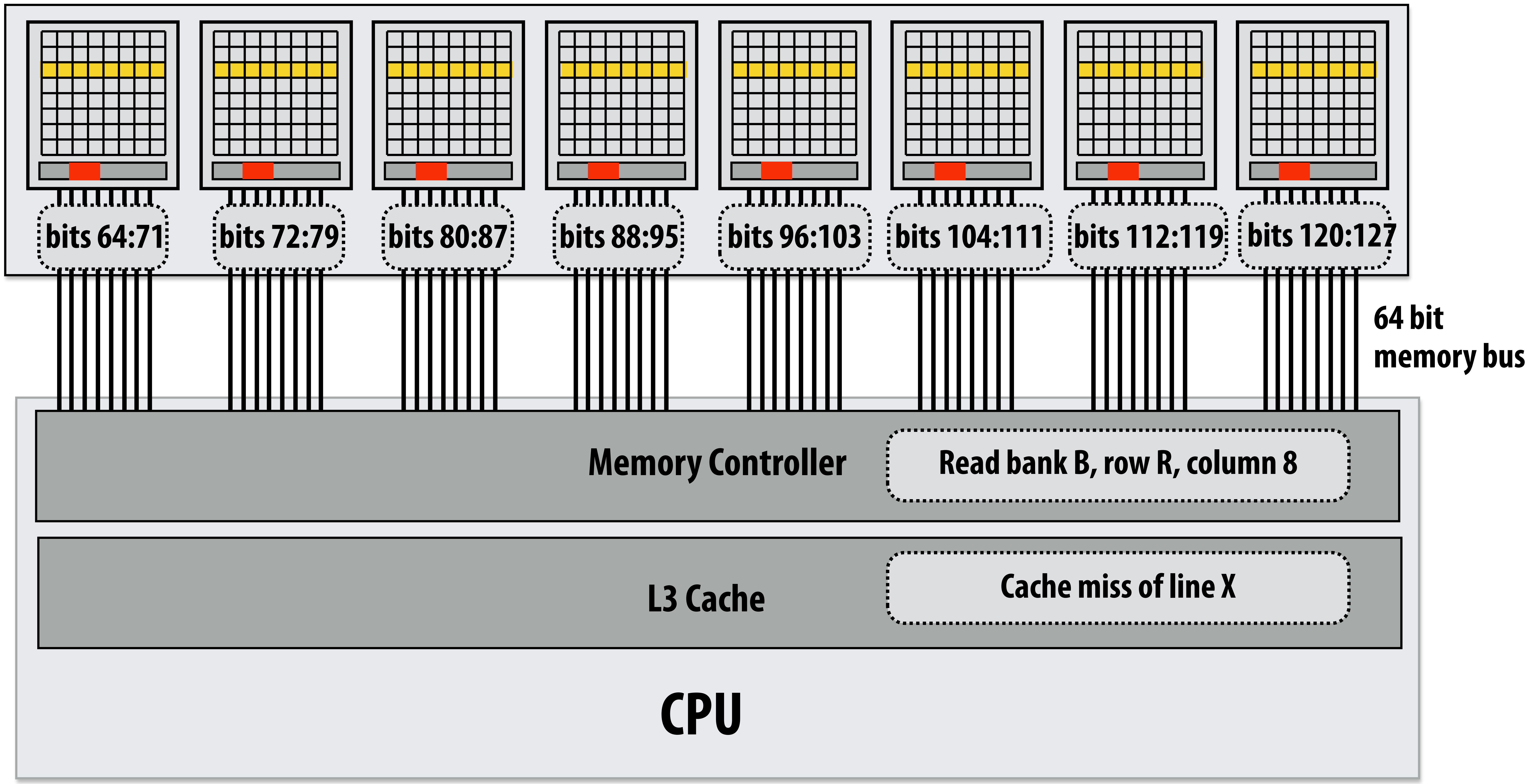
DRAM chips transmit first 64 bits in parallel (must activate row)



Reading one 64-byte (512 bit) cache line

DRAM controller requests data from new column *

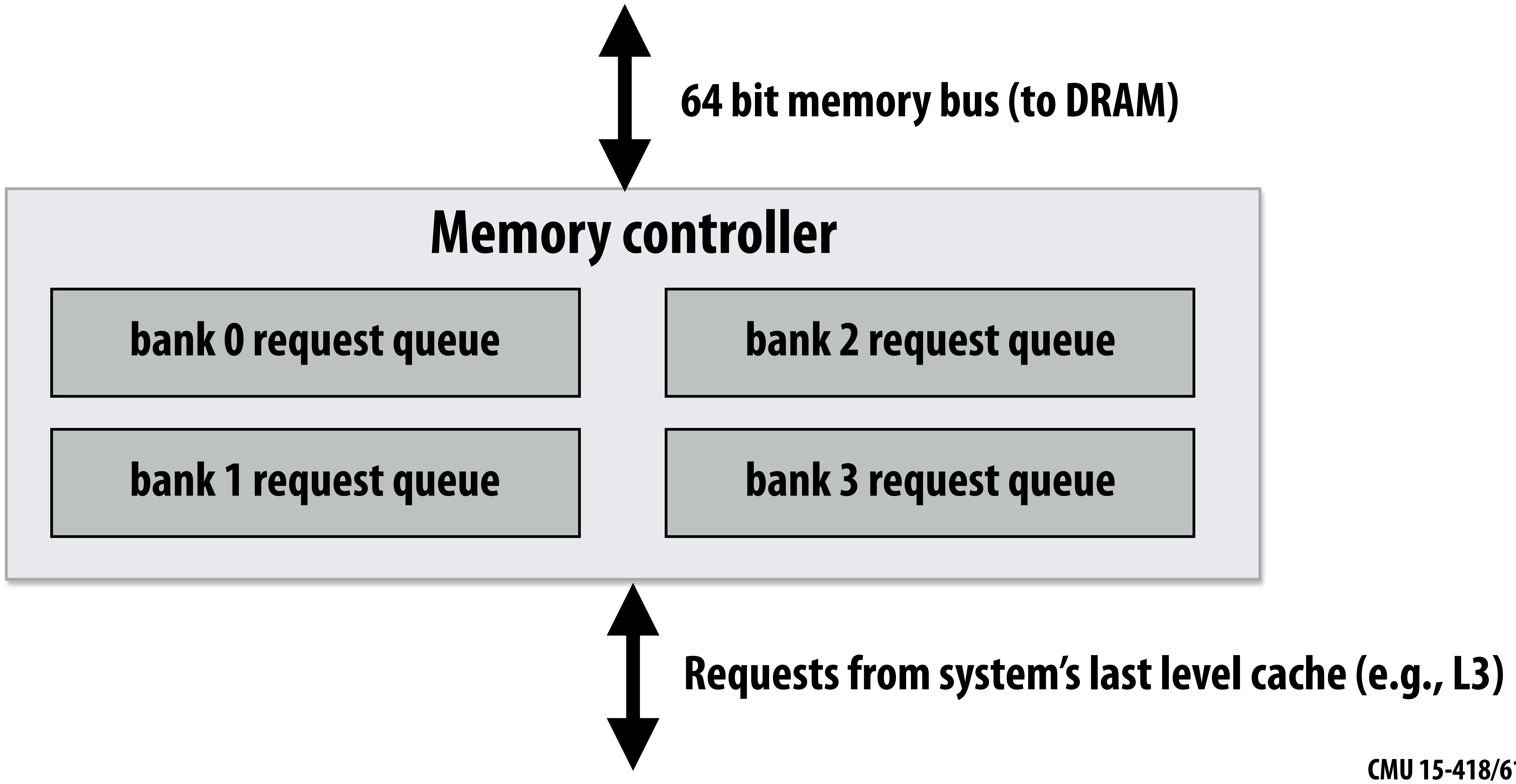
DRAM chips transmit next 64 bits in parallel



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

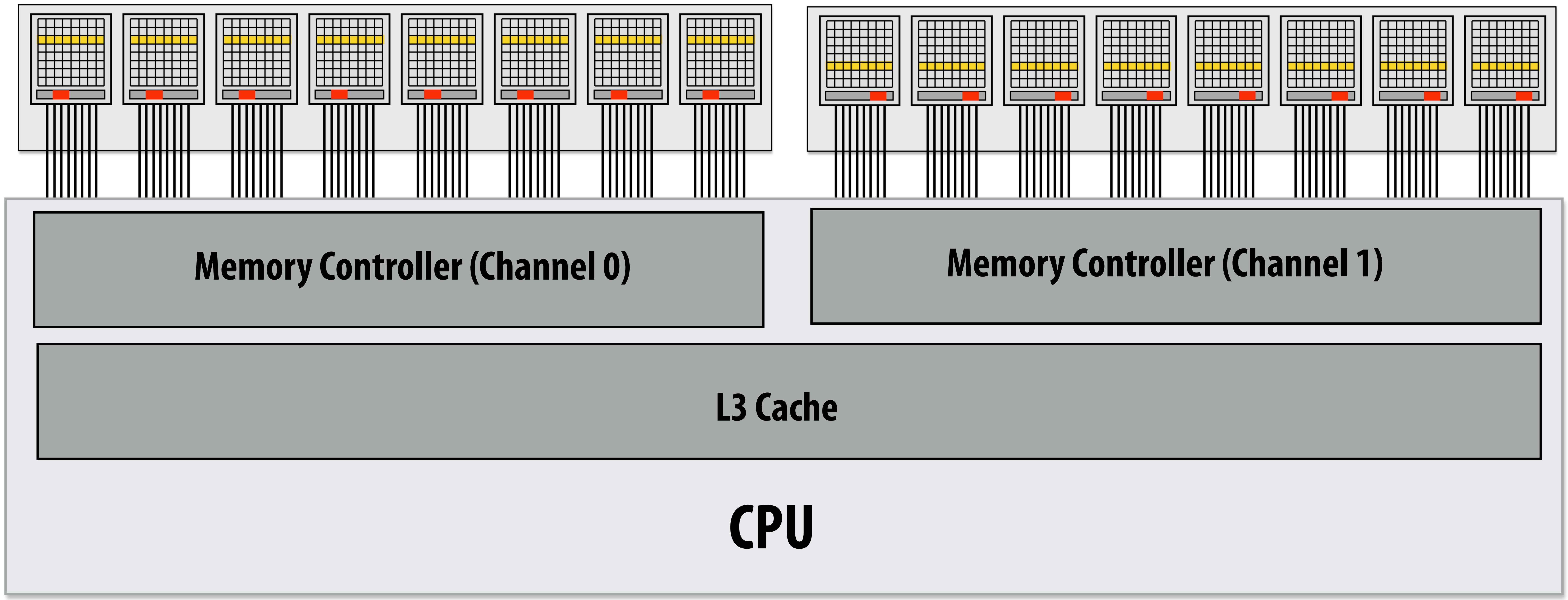
Memory controller is a memory request scheduler

- **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

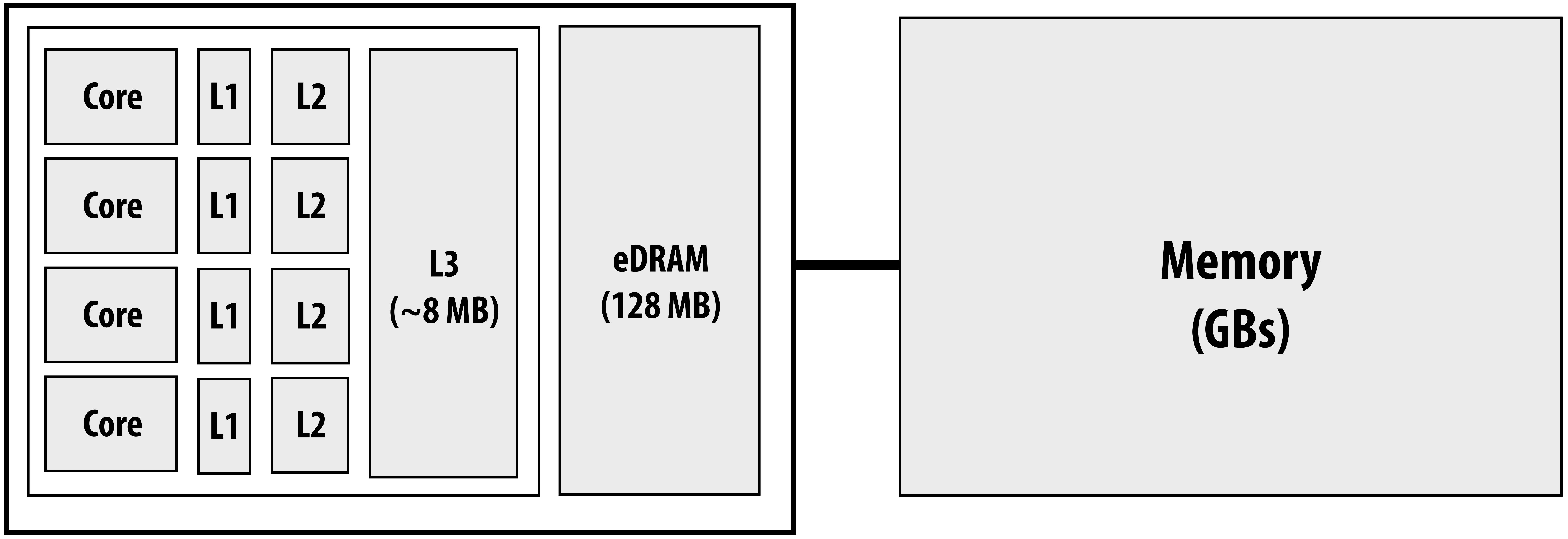
Decrease distance data must move: locate memory near processing

(or processing near memory)

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

eDRAM: another level of the memory hierarchy

- **High-end offerings of the Intel Haswell processors feature 128 MB of embedded DRAM (eDRAM) in the CPU package**
  - **50 GB/sec read + 50 GB/sec write**



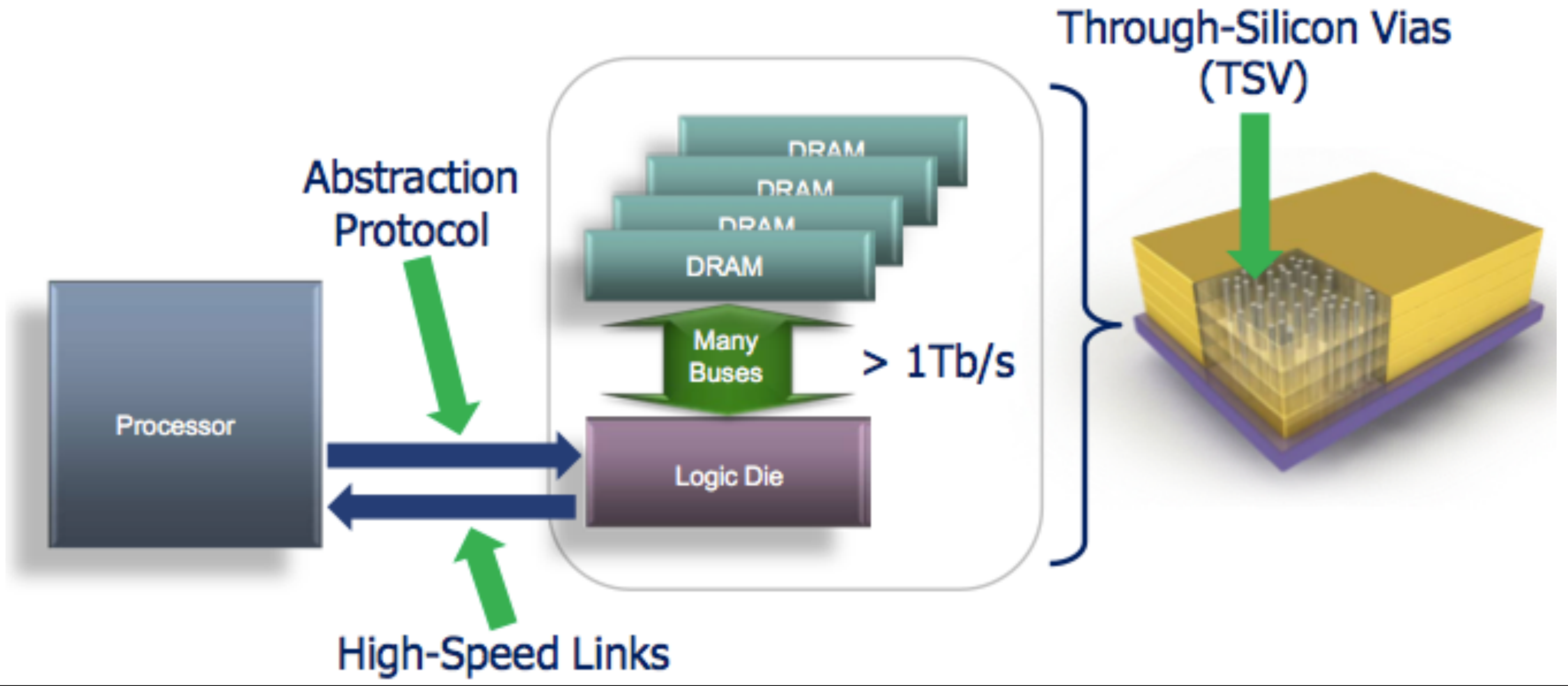
IBM Power 7 server CPUs feature eDRAM

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

Increasingly common in mobile SoC setting

Increase bandwidth by chip stacking

- **Enabling technology: 3D stacking of DRAM chips**
  - DRAMs connected via through-silicon-vias (TSVs) that run through the chips
  - Base layer of stack "logic layer" is memory controller, manages requests from processor
  - TSVs provide highly parallel connection between logic layer and DRAMs
  - 8-link configuration: 320 GB/sec between CPU and memory cube



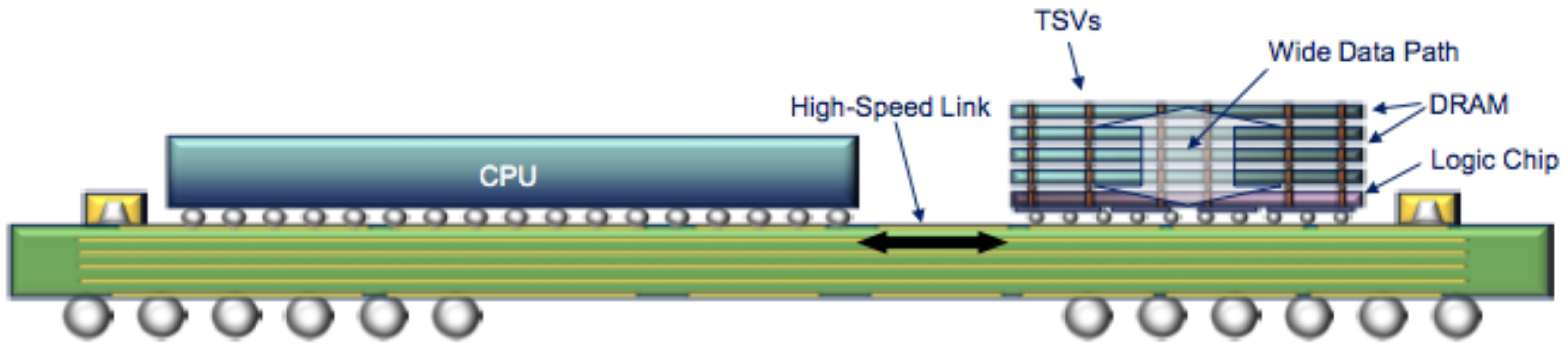
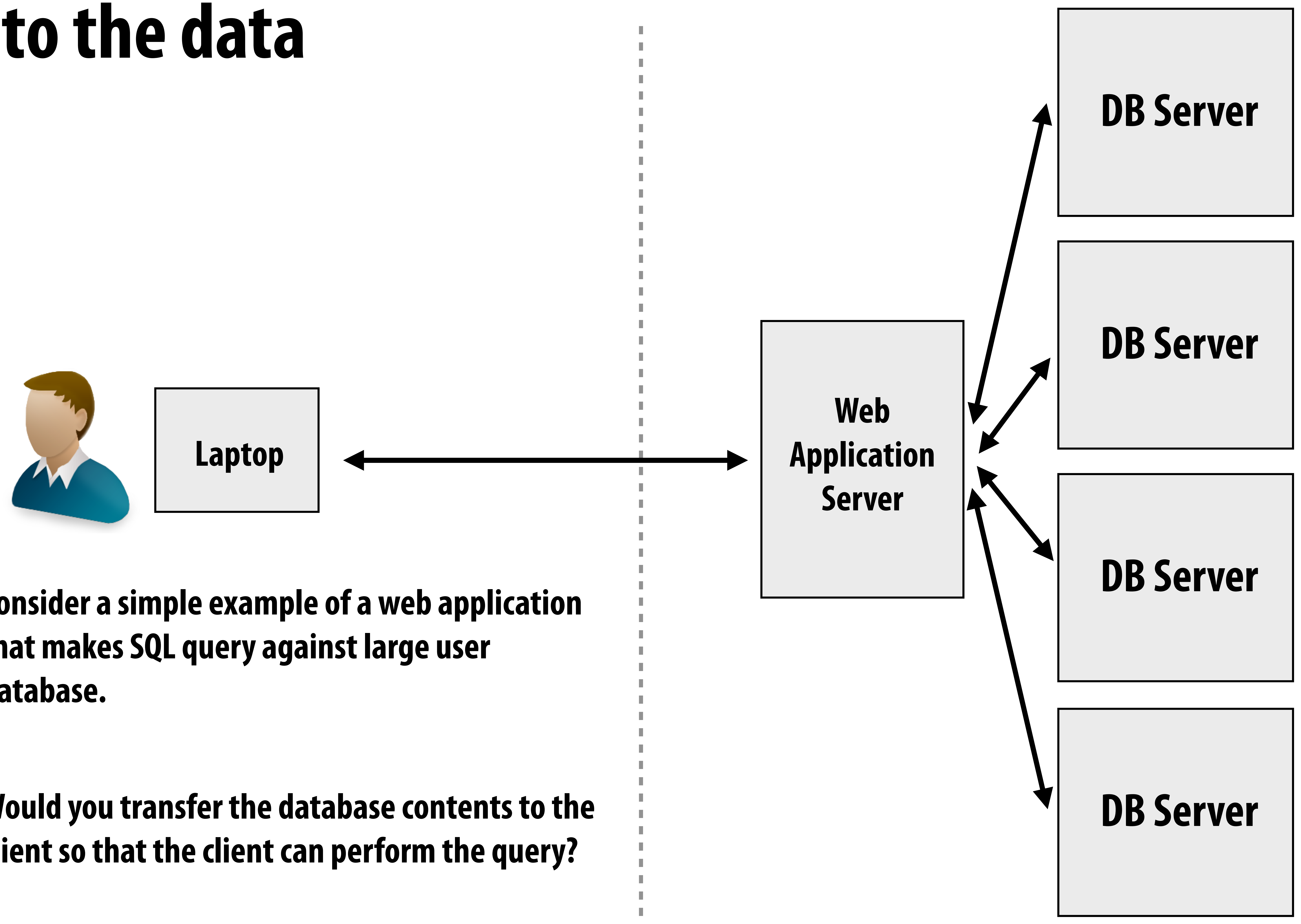


Image credits: Micron, Inc.

Note: height not to scale (actual package not much thicker than a traditional chip)

Reduce data movement by moving computation to the data



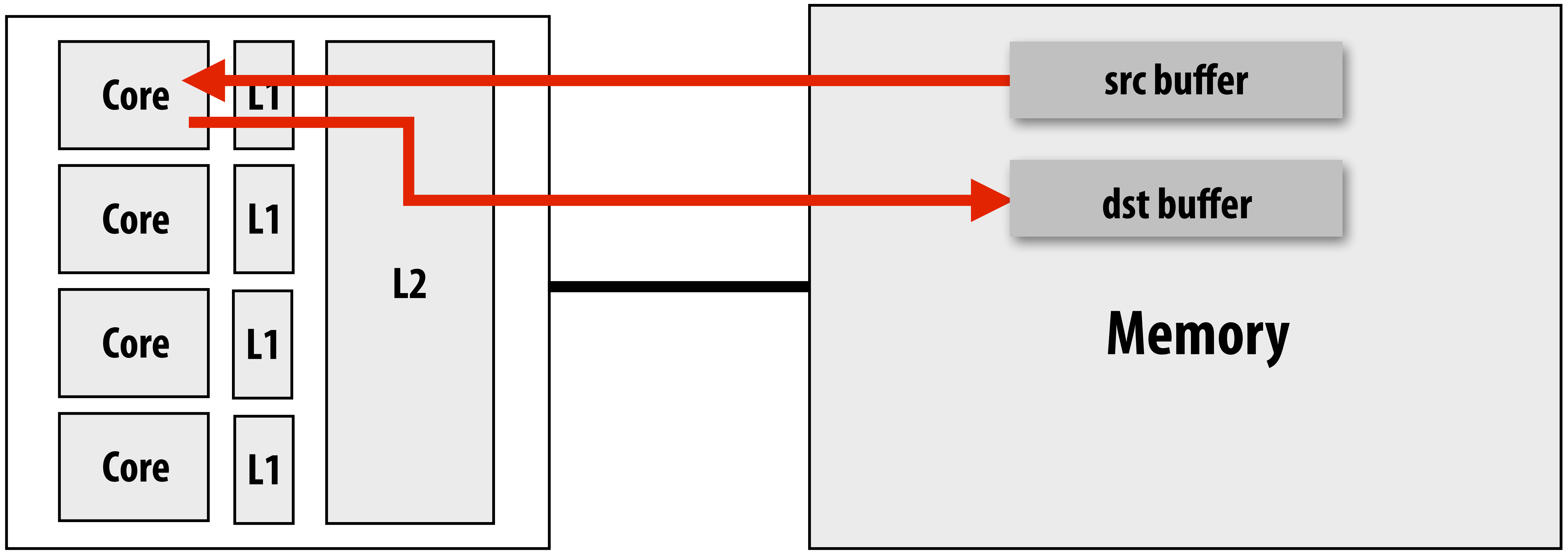
Consider a simple example of a web application that makes SQL query against large user database.

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

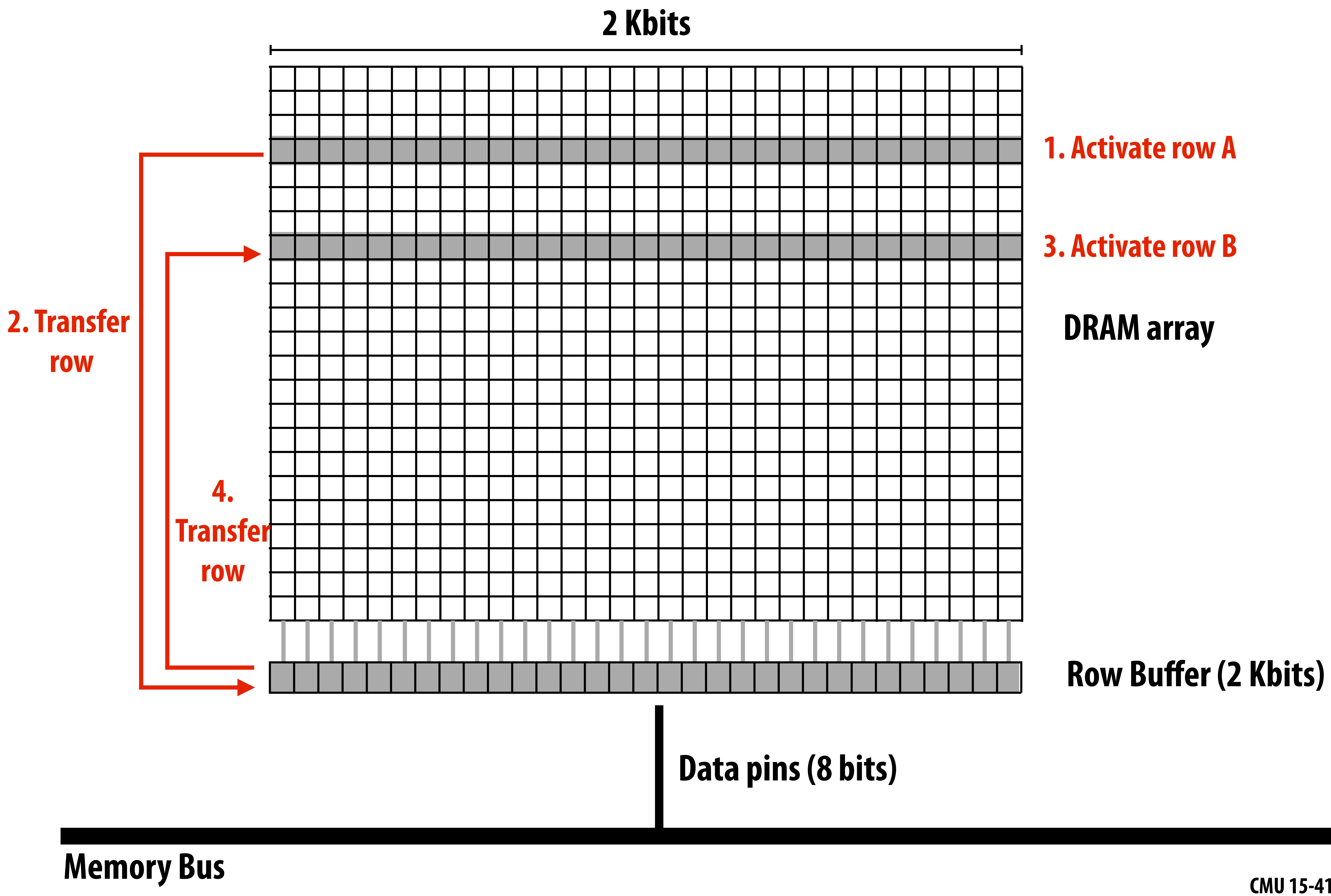
Bits move from DRAM, over bus, through cache hierarchy, into register file, and then retraces steps back out to DRAM

(and no computation is ever performed!)



Idea: perform copy without processor [Seshadri 13]

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



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**

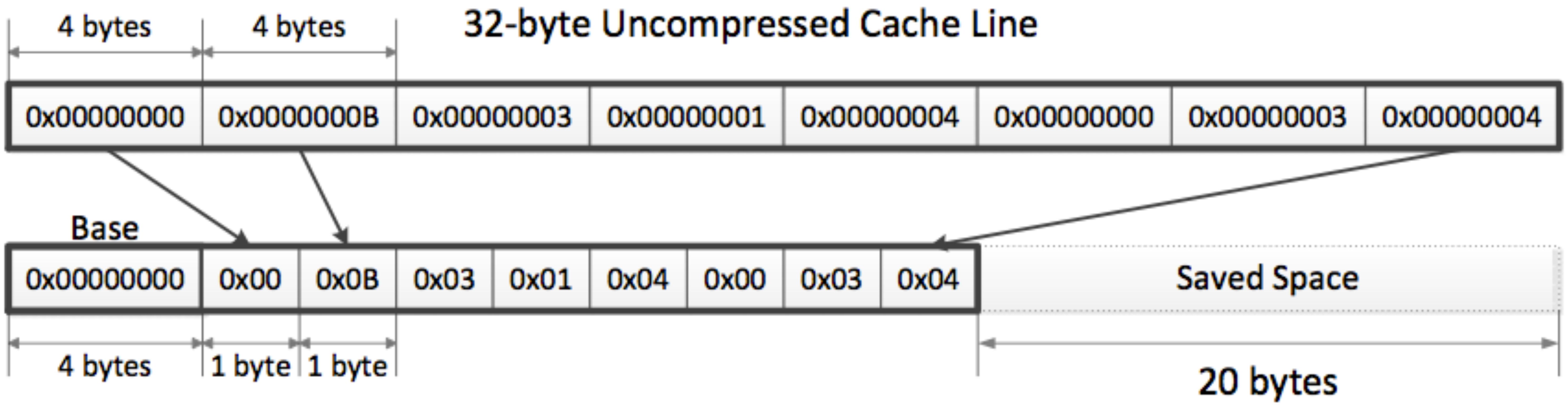
Compress data

- 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

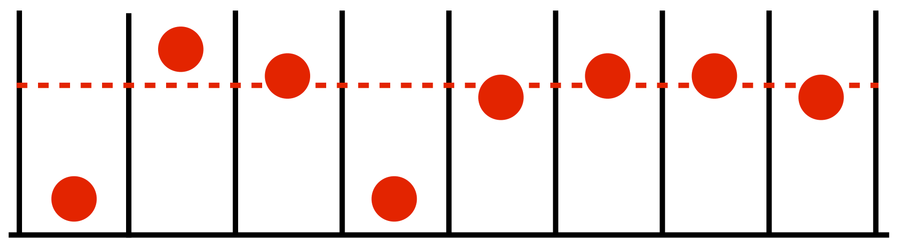
- Compress/decompression scheme must
  - Be simple enough to implement in HW
  - Be fast: decompression is on critical path of loads

One proposed example: BΔI compression [Pekhimenko 12]

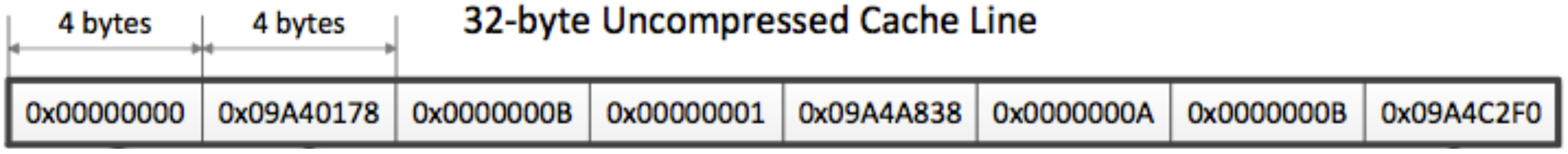
- Observation: data that falls within cache line often has low dynamic range (use base + offset to encode chunks of bits in a line)



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

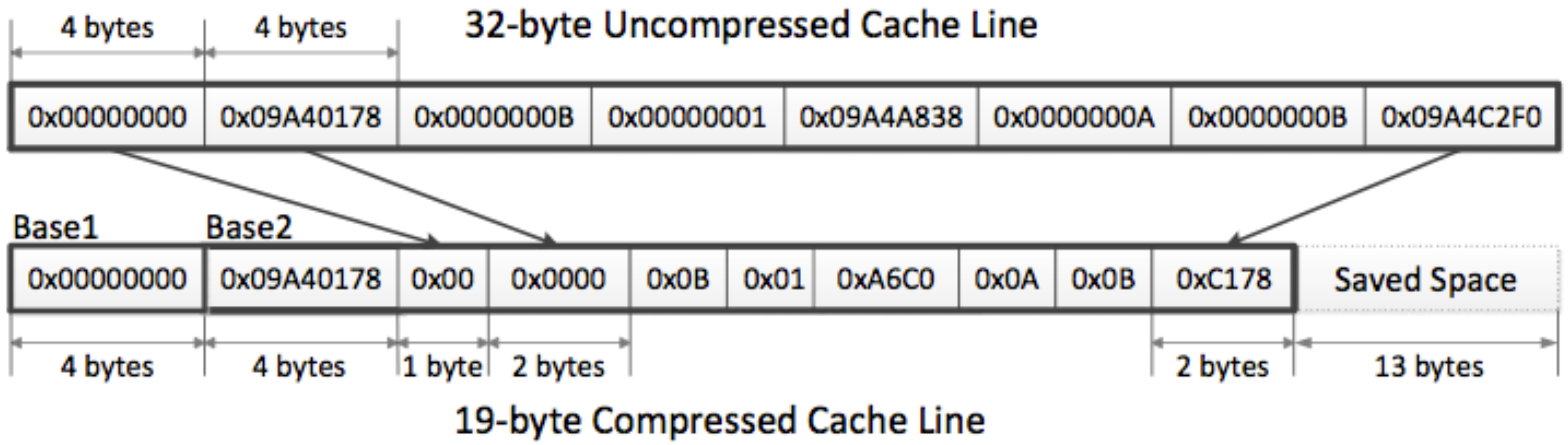


Does this pattern compress well?



0x00000000	0x09A40178	0x0000000B	0x00000001	0x09A4A838	0x0000000A	0x0000000B	0x09A4C2F0

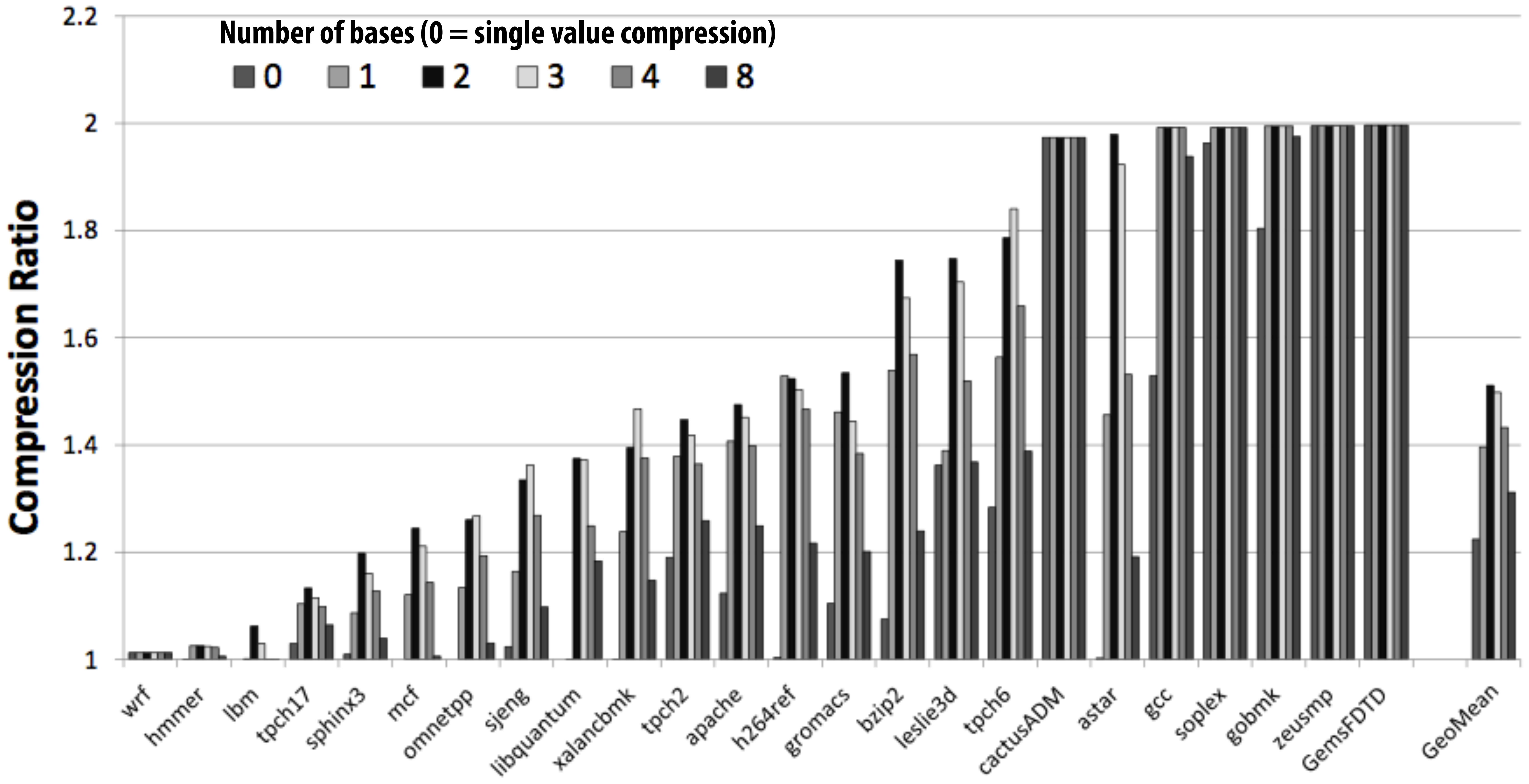
Does this pattern compress well?



- **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

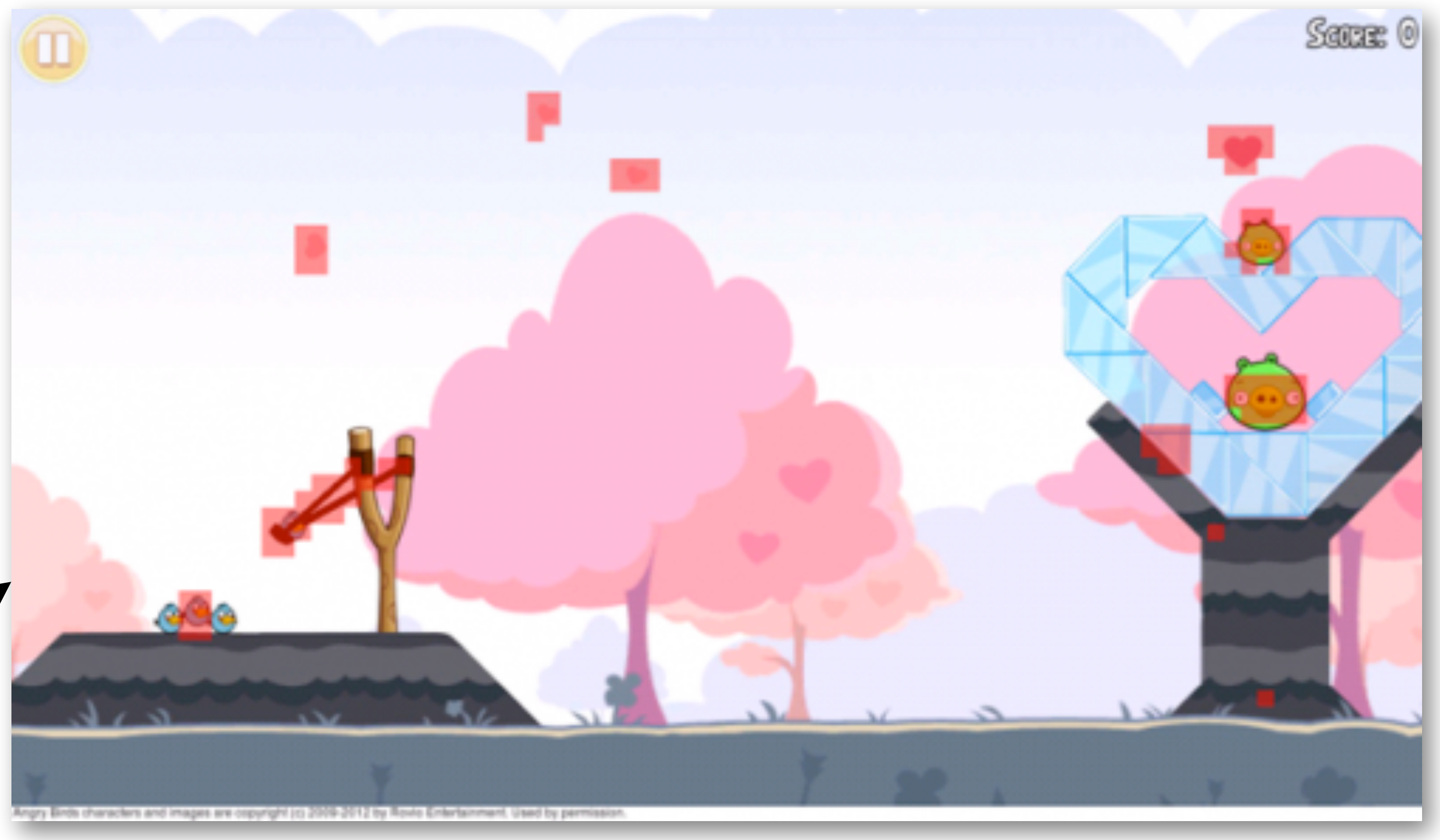
[Pekhimenko 12]



- On average: ~ 1.5x compression ratio
- Translates into ~ 10% performance gain, up to 18% on cache sensitive workloads

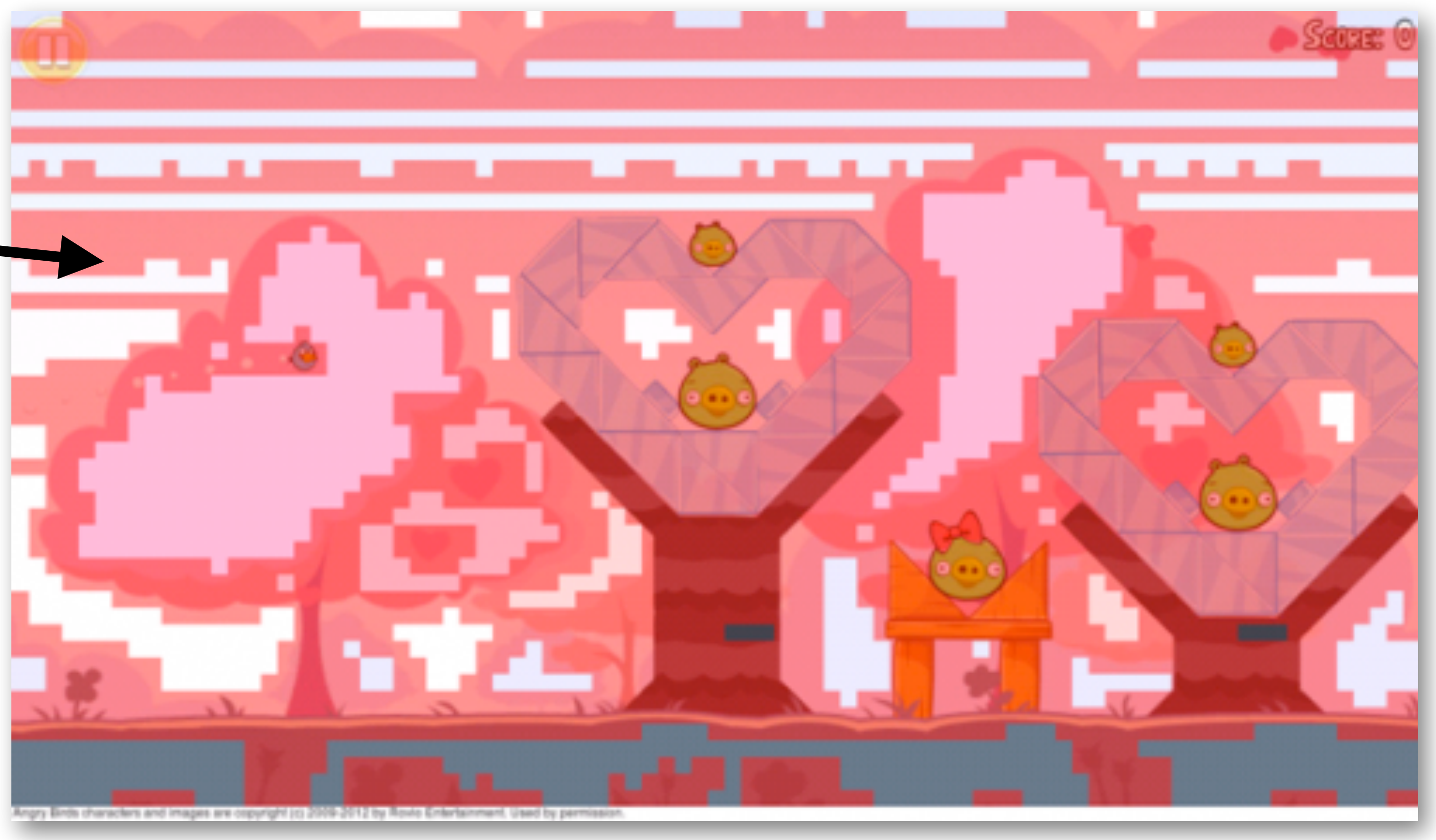
Bandwidth reduction trick in ARM GPUs

- Frame-buffer write during rendering is a bandwidth-heavy operation
- Idea: skip frame-buffer 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



- All GPUs losslessly compress frame-buffer contents prior to writing pixels to memory in order to save bandwidth (data compressed in memory, unlike previous example where data was compressed when in cache)

[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)
- **In hardware implementation by architects**
  - Intelligent DRAM request scheduling
  - Bringing data closer to processor (deep cache hierarchies, eDRAM)
  - Increase bandwidth (wider memory systems, near future: 3D stacking)
  - Ongoing research in locating limited 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)