

**Lecture 18:**

# **Parallel Programming on Graphs + How Memory Works**

---

**Parallel Computing  
Stanford CS149, Fall 2019**

# Today's topics

- **Programming abstractions for processing graphs**
  - **More examples of “domain-specific” programming systems**
- **Common optimizations when processing graphs**
- **How memory (DRAM) works**
  - **Today is a good lecture to talk about memory since graph algorithms are often bandwidth bound!**

# Last time: increasing acceptance of domain-specific programming systems

- **Challenge to programmers: modern computers are parallel, heterogeneous machines (HW architects striving for high area and power efficiency)**
- **Trend: domain-specific programming system design: give up generality in space of programs that can be expressed in exchange for achieving high productivity and high performance**
- **“Performance portability” is a key goal: programs should execute efficiently on a variety of parallel platforms**
  - **Good implementations of same program for different systems require different data structures, algorithms, and approaches to parallelization — not just differences in low-level code generation (optimization is not only a matter of generating SSE vs. AVX vs ARM Neon vs. NVIDIA PTX instructions)**

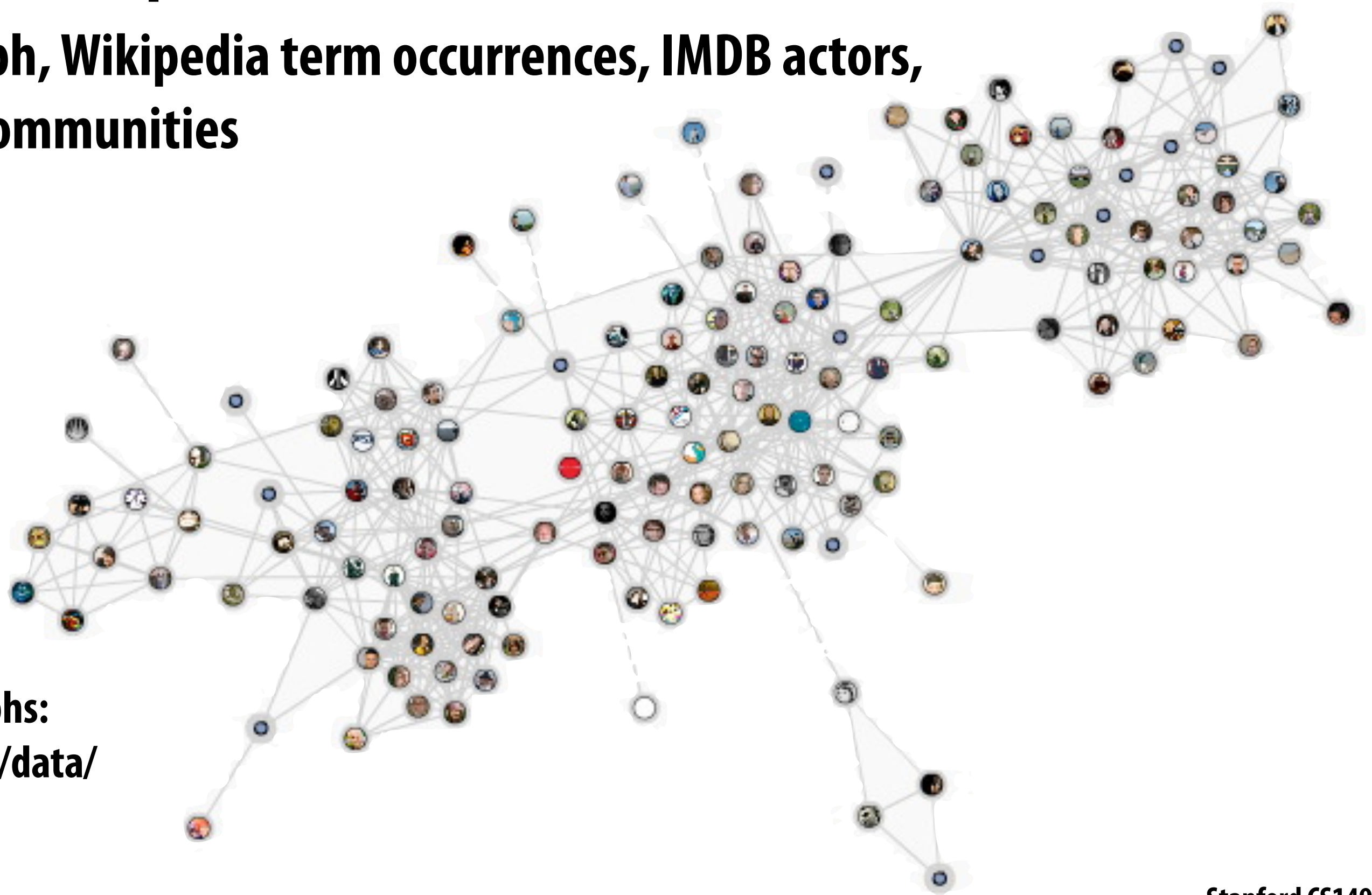
# Today's topic: analyzing big graphs

- **Many modern applications:**

- **Web search results, recommender systems, influence determination, advertising, anomaly detection, etc.**

- **Public dataset examples:**

**Twitter social graph, Wikipedia term occurrences, IMDB actors, Netflix, Amazon communities**



**Good source of public graphs:**  
<https://snap.stanford.edu/data/>

**Thought experiment: if we wanted to design a programming system for computing on graphs, where might we begin?**

**What abstractions do we need?**

# Whenever I'm trying to assess the importance of a new programming system, I ask two questions:

**“What tasks/problems does the system take off the programmer's hands? (are these problems challenging or tedious enough that I feel the system is adding sufficient value for me to want to use it?)”**

**“What problems does the system leave as the responsibility for the programmer?” (usually because the programmer is better at these tasks)**

**Halide (recall previous class):**

**Programmer's responsibility:**

- Describing image processing algorithm as pipeline of operations on images
- Describing the schedule for executing the pipeline (e.g., “block this loop, “parallelize this loop”, “fuse these stages”)

**Halide system's responsibility:**

- Implementing the schedule using mechanisms available on the target machine (spawning pthreads, allocating temp buffers, emitting vector instructions, loop indexing code)

# Programming system design questions:

- **What are the fundamental operations we want to be easy to express AND efficient to execute?**
- **What are the key optimizations used when authoring the best implementations of these operations?  
(high-level abstractions provided by a programming system should not stand in the way of these optimizations, and maybe even allow the system to perform them for the application)**



# Example graph computation: Page Rank

Page Rank: iterative graph algorithm

Graph nodes = web pages

Graph edges = links between pages

$$R[i] = \frac{1 - \alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

Rank of page  $i$

discount

Weighted combination of rank of pages that link to it



# GraphLab

- A system for describing iterative computations on graphs
- Implemented as a C++ library
- Runs on shared memory machines or distributed across clusters
  - GraphLab runtime takes responsibility for scheduling work in parallel, partitioning graphs across clusters of machines, communication between master, etc.

# GraphLab programs: state

## ■ The graph: $G = (V, E)$

- Application defines data blocks on each vertex and directed edge
- $D_v =$  data associated with vertex  $v$
- $D_{u \rightarrow v} =$  data associated with directed edge  $u \rightarrow v$

## ■ Read-only global data

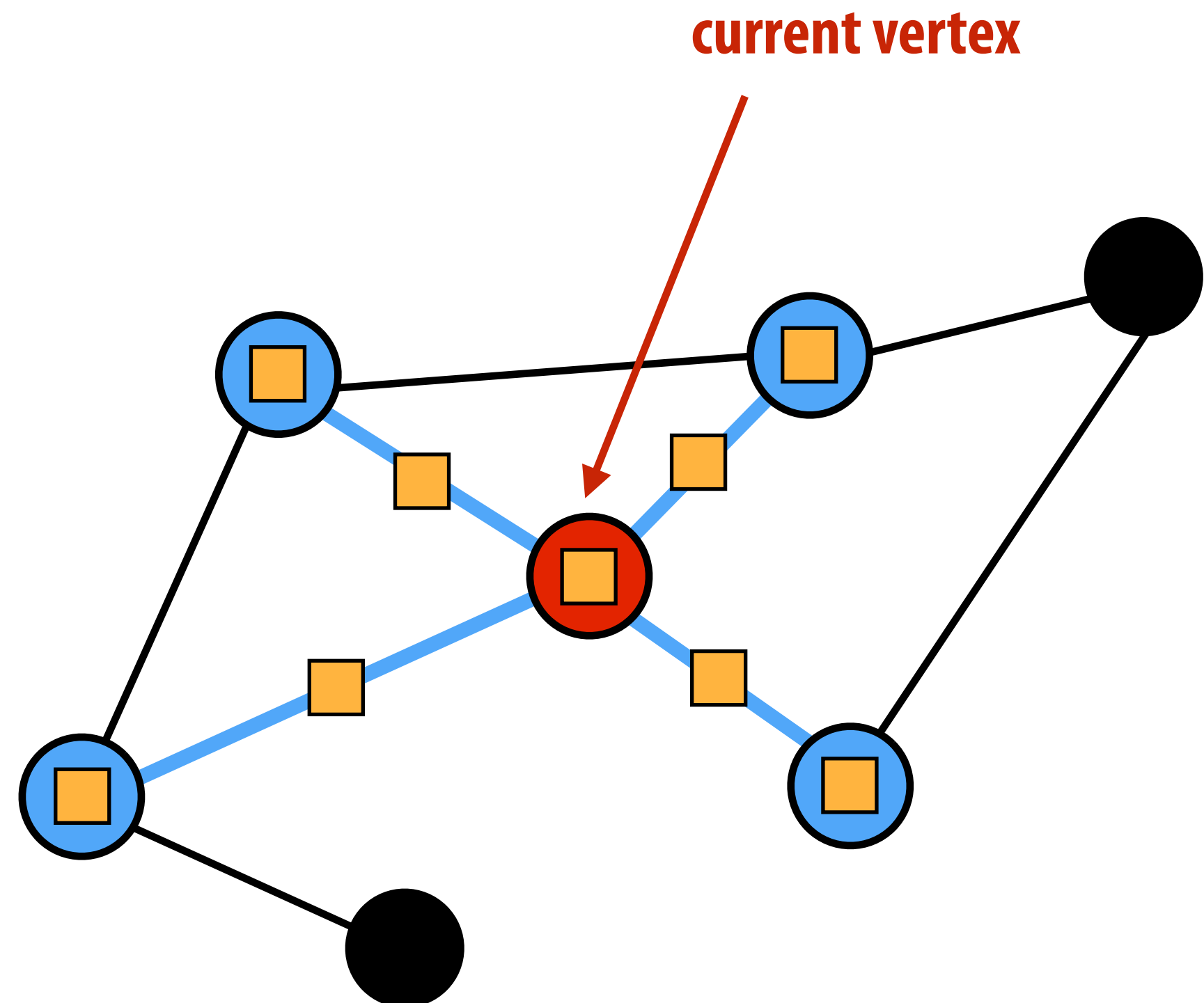
- Can think of this as per-graph data, rather than per vertex or per-edge data)

**Notice: I always first describe program state**

**And then describe what operations are available to manipulate this state**

# GraphLab operations: the “vertex program”

- Defines per-vertex operations on the vertex’s local neighborhood
- Neighborhood (aka “scope”) of vertex:
  - The current vertex
  - Adjacent edges
  - Adjacent vertices



 = vertex or edge data “in scope” of red vertex  
(graph data that can be accessed when executing  
a vertex program at the current (red) vertex)

# Simple example: PageRank \*

$$R[i] = \frac{1 - \alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

```
PageRank_vertex_program(vertex i) {  
  
    // (Gather phase) compute the sum of my neighbors rank  
    double sum = 0;  
    foreach(vertex j : in_neighbors(i)) {  
        sum = sum + j.rank / num_out_neighbors(j);  
    }  
  
    // (Apply phase) Update my rank (i)  
    i.rank = (1-0.85)/num_graph_vertices() + 0.85*sum;  
}
```

Let alpha = 0.85

**Programming in GraphLab amounts to defining how to update graph state at each vertex. The system takes responsibility for scheduling and parallelization.**

\* This is made up syntax for slide simplicity: actual syntax is C++, as we'll see on the next slide

# GraphLab: data access

- **The application's vertex program executes per-vertex**
- **The vertex program defines:**
  - **What adjacent edges are inputs to the computation**
  - **What computation to perform per edge**
  - **How to update the vertex's value**
  - **What adjacent edges are modified by the computation**
  - **How to update these output edge values**
- **Note how GraphLab requires the program to tell it all data that will be accessed, and whether it is read or write access**

# PageRank: GraphLab vertex program (C++ code)

```
struct web_page {  
    std::string pagename;  
    double pagerank;  
    web_page(): pagerank(0.0) { }  
}
```

```
typedef graphlab::distributed_graph<web_page, graphlab::empty> graph_type;
```

Graph has record of type `web_page` per vertex, and no data on edges

```
class pagerank_program : public graphlab::ivertex_program ...  
{  
public:
```

```
    // we are going to gather over all the in-edges  
    edge_dir_type gather_edges(icontext_type& context,  
                               const vertex_type& vertex) const {  
        return graphlab::IN_EDGES;  
    }
```

Define edges to gather over in "gather phase"

```
    // for each in-edge gather the weighted sum of the edge.  
    double gather(icontext_type& context, const vertex_type& vertex,  
                  edge_type& edge) const {  
        return edge.source().data().pagerank / edge.source().num_out_edges();  
    }
```

Compute value to accumulate for each edge

```
    // Use the total rank of adjacent pages to update this page  
    void apply(icontext_type& context, vertex_type& vertex,  
              const gather_type& total) {  
        double newval = total * 0.85 + 0.15;  
        vertex.data().pagerank = newval;  
    }
```

Update vertex rank

```
    // No scatter needed. Return NO_EDGES  
    edge_dir_type scatter_edges(icontext_type& context,  
                                const vertex_type& vertex) const {  
        return graphlab::NO_EDGES;  
    }  
};
```

PageRank example performs no scatter



# Running the program

```
graphlab::omni_engine<pagerank_program> engine(dc, graph, "sync");  
engine.signal_all();  
engine.start();
```

**GraphLab runtime provides “engines” that manage scheduling of vertex programs**  
`engine.signal_all()` marks all vertices for execution

**You can think of the GraphLab runtime as a work queue scheduler.**  
**And invoking a vertex program on a vertex as a task that is placed in the work queue.**

**So it’s reasonable to read the code above as: “place all vertices into the work queue”**

**Or as: “foreach vertex” run the vertex program.**

# Vertex signaling: GraphLab's mechanism for generating new work

$$R[i] = \frac{1 - \alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$

Iteratively update all  $R[i]$ 's 10 times

Uses generic "signal" primitive (could also wrap code on previous slide in a for loop)

```
struct web_page {
  std::string pagename;
  double pagerank;
  int counter;
  web_page(): pagerank(0.0), counter(0) { }
}
```

Per-vertex "counter"



```
// Use the total rank of adjacent pages to update this page
void apply(icontext_type& context, vertex_type& vertex,
           const gather_type& total) {
  double newval = total * 0.85 + 0.15;
  vertex.data().pagerank = newval;
  vertex.data().counter++;
  if (vertex.data().counter < 10)
    vertex.signal();
}
```

If counter < 10, signal to scheduler to run the vertex program on the vertex again at some point in the future



# Signal: general primitive for scheduling work

Parts of graph may converge at different rates

(iterate PageRank until convergence, but only for vertices that need it)

```
class pagerank_program: public graphlab::ivertex_program
```

```
private:
```

```
    bool perform_scatter;
```



Private variable set during apply phase,  
used during scatter phase

```
public:
```

```
    // Use the total rank of adjacent pages to update this page
```

```
void apply(icontext_type& context, vertex_type& vertex,
```

```
           const gather_type& total) {
```

```
    double newval = total * 0.85 + 0.15;
```

```
    double oldval = vertex.data().pagerank;
```

```
    vertex.data().pagerank = newval;
```

```
    perform_scatter = (std::fabs(prevval - newval) > 1E-3);
```



Check for convergence

```
}
```

```
    // Scatter now needed if algorithm has not converged
```

```
edge_dir_type scatter_edges(icontext_type& context,
```

```
                             const vertex_type& vertex) const {
```

```
    if (perform_scatter) return graphlab::OUT_EDGES;
```

```
    else return graphlab::NO_EDGES;
```

```
}
```

```
    // Make sure surrounding vertices are scheduled
```

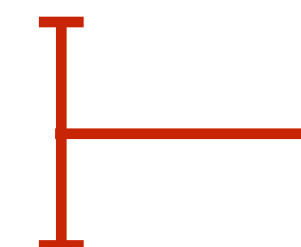
```
void scatter(icontext_type& context, const vertex_type& vertex,
```

```
            edge_type& edge) const {
```

```
    context.signal(edge.target());
```

```
}
```

```
};
```



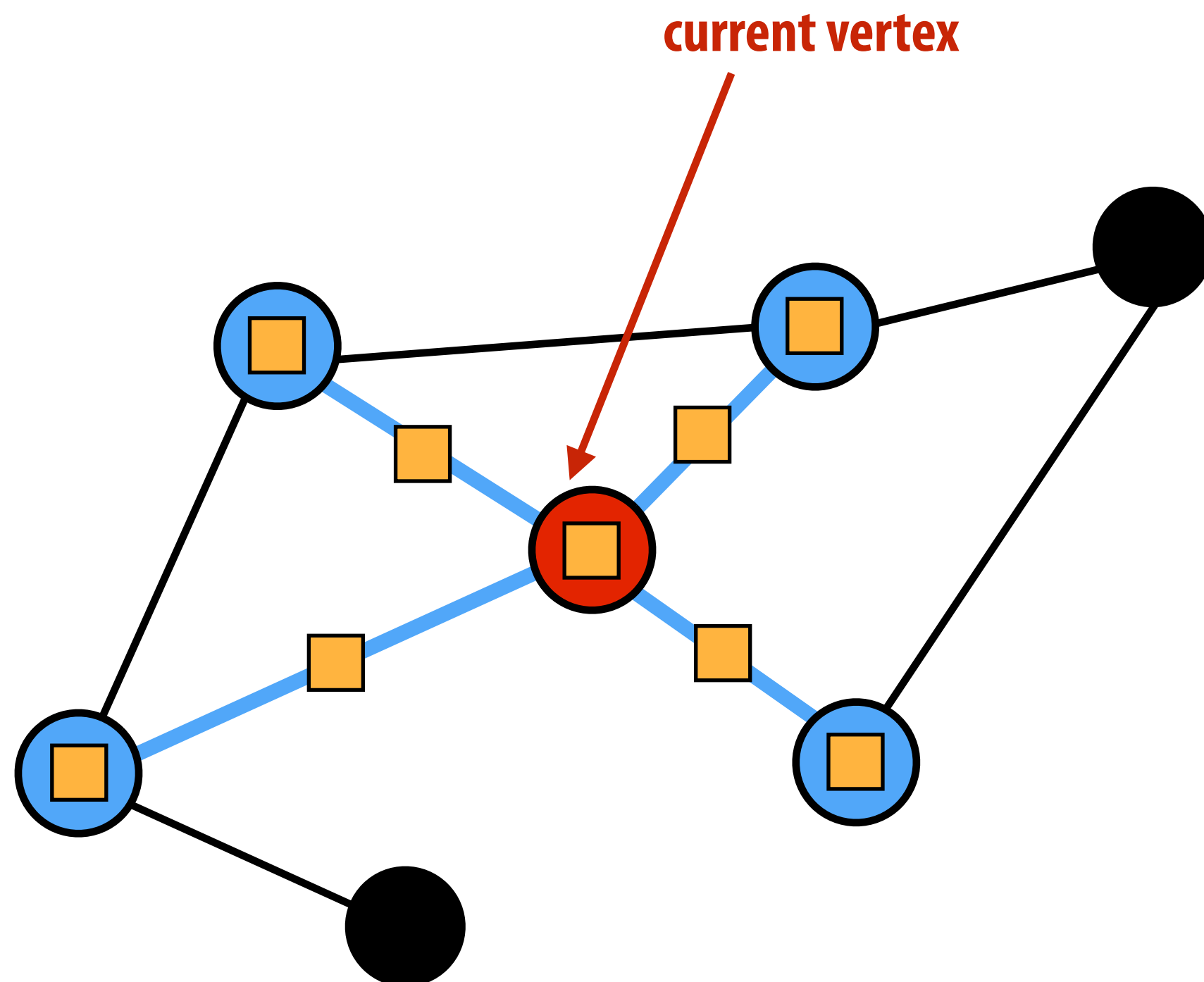
Schedule update of  
neighbor vertices

# Synchronizing parallel execution

Local neighborhood of vertex (vertex's "scope") can be read and written to by a vertex program

Programs specify what granularity of atomicity they want GraphLab to provide: this determines amount of available parallelism

- **"Full"**: implementation ensures no other execution reads or writes to data in scope of  $v$  when vertex program for  $v$  is running.
- **"Edge"**: no other execution reads or writes any data in  $v$  or in edges adjacent to  $v$
- **"Vertex"**: no other execution reads or writes to data in  $v$  ...



 = vertex or edge data in scope of red vertex

# GraphLab: work scheduling order

- **GraphLab implements several work scheduling policies**
  - **Synchronous: update all vertices simultaneously (vertex programs observe no updates from programs run on other vertices in same “round”)**
  - **Round-robin: vertex programs observe most recent updates**
  - **Dynamic: based on new work created by `signal`**
    - **Several implementations: `fifo`, `priority-based`, “`splash`” ...**
- **Application developer has flexibility for choosing atomicity guarantee and scheduling policy**
  - **Implication: choice of schedule impacts program’s correctness/output**

# Summary: GraphLab concepts

- **Program state: data on graph vertices and edges + globals**
- **Operations: per-vertex update programs and global reduction functions (reductions not discussed today)**
  - **Simple, intuitive description of work (follows mathematical formulation)**
  - **Graph restricts data access in vertex program to local neighborhood**
  - **Asynchronous execution model: application creates work dynamically by “signaling vertices” (enable lazy execution, work efficiency on real graphs)**
- **Choice of scheduler and atomicity implementation**
  - **In this domain, the order in which nodes are processed can be critical property for both performance and quality of result**
  - **Application responsible for choosing right scheduler for its needs**

# **Ligra**

**(efficient data-parallel graph processing  
for shared memory multi-cores)**



- A simple framework for parallel graph operations on shared memory multi-core machines
- Motivating example: breadth-first search

```
parents = {-1, ..., -1}
```

```
// d = dst: vertex to "update" (just encountered)
```

```
// s = src: vertex on frontier with edge to d
```

```
procedure UPDATE(s, d)
```

```
    return compare-and-swap(parents[d], -1, s);
```

```
procedure COND(i)
```

```
    return parents[i] == -1;
```

```
procedure BFS(G, r)
```

```
    parents[r] = r;
```

```
    frontier = {r};
```

```
    while (size(frontier) != 0) do:
```

```
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

**Semantics of EDGEMAP:**

**foreach vertex  $i$  in frontier, call UPDATE for all neighboring vertices  $j$  for which COND( $j$ ) is true. Add  $j$  to returned set if UPDATE( $i, j$ ) returns true**

# Implementing edgemap

- Assume vertex subset  $U$  (frontier in previous example) is represented sparsely:

- e.g., three vertex subset  $U$  of 10 vertex graph  $G=(E,V)$ :  $U \subset V = \{0, 4, 9\}$

```
procedure EDGEMAP_SPARSE(G, U, F, C):  
  result = {}  
  parallel foreach v in U do:  
    parallel foreach v2 in out_neighbors(v) do:  
      if (C(v2) == 1 and F(v,v2) == 1) then  
        add v2 to result  
  remove duplicates from result  
  return result;
```

graph

set of vertices

condition check on neighbor vertex

update function on neighbor vertex

**Cost of EDGEMAP\_SPARSE?**

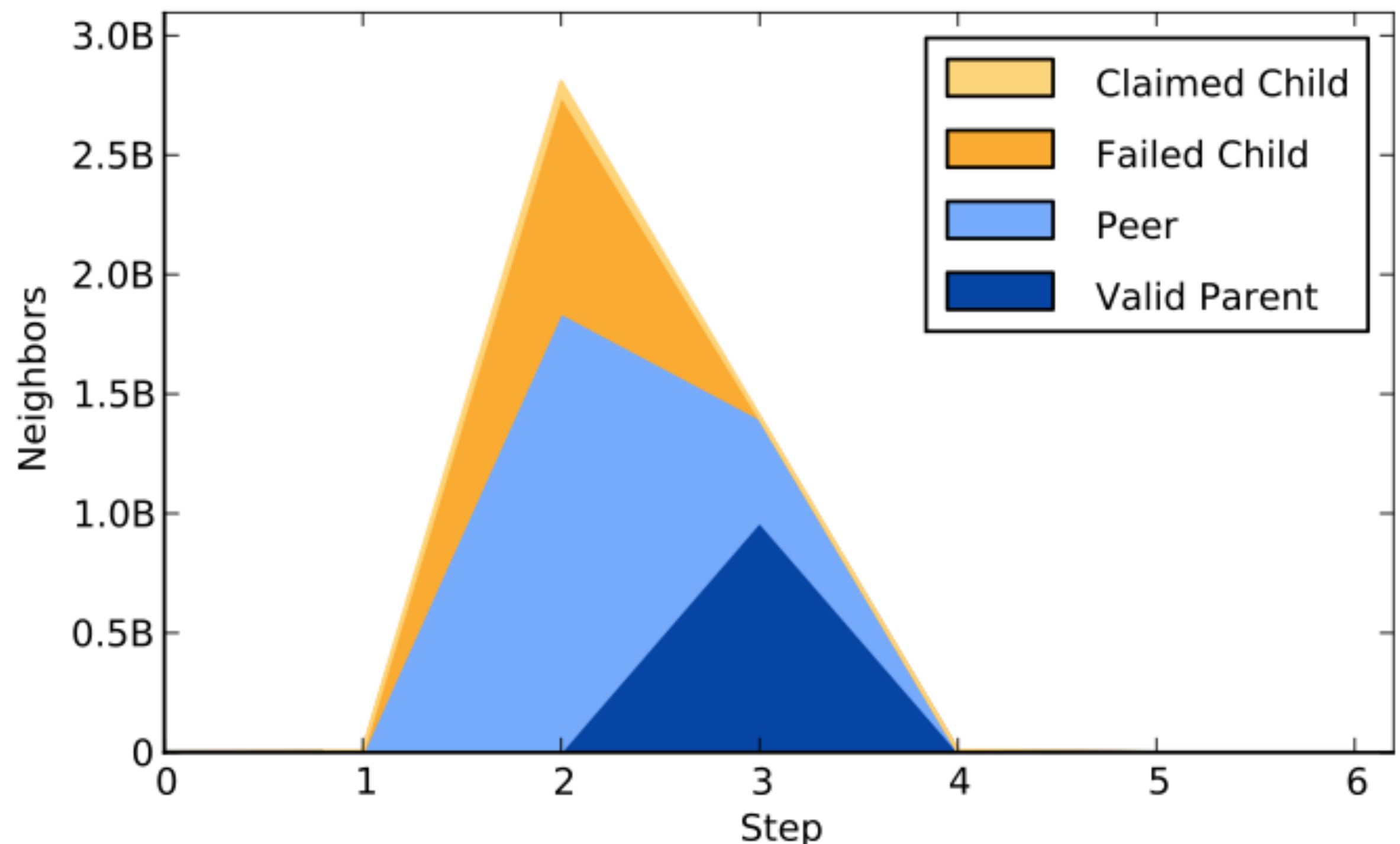
**$O(|U| + \text{sum of outgoing edges from } U)$**

```
parents = {-1, ..., -1}  
  
procedure UPDATE(s, d)  
  return compare-and-swap(parents[d], -1, s);  
  
procedure COND(i)  
  return parents[i] == -1;  
  
procedure BFS(G, r)  
  parents[r] = r;  
  frontier = {r};  
  while (size(frontier) != 0) do:  
    frontier = EDGEMAP(G, frontier, UPDATE, COND);
```

# Visiting every edge on frontier can be wasteful

- **Each step of BFS, every edge on frontier is visited**
  - **Frontier can grow quickly for social graphs (few steps to visit all nodes)**
  - **Most edge visits are wasteful! (they don't lead to a successful "update")**

- **claimed child: edge points to unvisited vertex (useful work)**
- **failed child: edge points to vertex found in this step via another edge**
- **peer: edge points to a vertex that was added to frontier in same step as current vertex**
- **valid parent: edge points to vertex found in previous step**



[Credit: Beamer et al. SC12]

# Implementing edgemap for dense vertex subsets

- Assume vertex subset (frontier in previous example) is represented densely with a bitvector:

- e.g., vertex subset  $U$  of 10 vertex graph  $G=(E,V)$ :  $U \subset V = \{1,0,0,0,1,0,0,0,0,1\}$

```
procedure EDGEMAP_SPARSE(G, U, F, C):
  result = {}
  parallel foreach v in U do:
    parallel foreach v2 in out_neighbors(v) do:
      if (C(v2) == 1 and F(v,v2) == 1) then
        add v2 to result
  remove duplicates from result
  return result;
```

```
procedure EDGEMAP_DENSE(G, U, F, C):
  result = {}
  parallel for i in {0,...,|V|-1} do:
    if (C(i) == 1) then:
      foreach v in in_neighbors(i) do:
        if v ∈ U and F(v, i) == 1 then:
          add i to result;
        if (C(i) == 0)
          break;
  return result;
```

## Cost of EDGEMAP\_DENSE?

For each unvisited vertex, quit searching as soon as some parent is found

Could be as low as  $O(|V|)$

Also no synchronization needed (“gather” results rather than “scatter”)

# Ligra on one slide

## ■ Entities:

- **Graphs**
- **Vertex subsets (represented sparsely or densely by system)**
- **EDGEMAP and VERTEXMAP functions**

```
procedure EDGEMAP(G, U, F, C):  
  if (|U| + sum of out degrees > threshold)  
    return EDGEMAP_DENSE(G, U, F, C);  
  else  
    return EDGEMAP_SPARSE(G, U, F, C);
```

Iterate over all vertices adjacent to  
vertices in set U  
Choose right algorithm for the job

```
procedure VERTEXMAP(U, F):  
  result = {}  
  parallel for u ∈ U do:  
    if (F(u) == 1) then:  
      add u to result;  
  return result;
```

Iterate over all vertices in set U

# Page rank in Ligra

```
r_cur = {1/|V|, ... 1/|V|};  
r_next = {0, ..., 0};  
diff = {}
```

```
procedure PRUPDATE(s, d):  
    atomicIncrement(&r_next[d], r_cur[s] / vertex_degree(s));
```

```
procedure PRLOCALCOMPUTE(i):  
    r_next[i] = alpha * r_next[i] + (1 - alpha) / |V|;  
    diff[i] = |r_next[i] - r_cur[i]|;  
    r_cur[i] = 0;  
    return 1;
```

```
procedure COND(i):  
    return 1;
```

```
procedure PAGERANK(G, alpha, eps):  
    frontier = {0, ... , |V|-1}  
    error = HUGE;  
    while (error > eps) do:  
        frontier = EDGEMAP(G, frontier, PRUPDATE, COND);  
        frontier = VERTEXMAP(frontier, PRLOCALCOMPUTE);  
        error = sum of per-vertex diffs // this is a parallel reduce  
        swap(r_cur, r_next);  
    return err
```

**Question: can you implement the iterate until convergence optimization we previously discussed in GraphLab?**

**(if so, what GraphLab scheduler implementation is the result equivalent to?)**

# Ligra summary

- **System abstracts graph operations as data-parallel operations over vertices and edges**
  - **Emphasizes graph traversal (potentially small subset of vertices operated on in a data parallel step)**
- **These basic operations permit a surprisingly wide space of graph algorithms:**
  - **Betweenness centrality**
  - **Connected components**
  - **Shortest paths**

See **Ligra: a Lightweight Framework for Graph Processing for Shared Memory [Shun and Blelloch 2013]**



# Ligra

Simple library with many useful examples

- Introduction
- Getting Started
- Tutorial: BFS
- Tutorial: KCore
- API
  - Vertex
  - Graph
- Running Code
- Examples

## Examples

Implementation files are provided in the apps/ directory:

- **BFS.C** (breadth-first search)
- **BFS-Bitvector.C** (breadth-first search with a bitvector to mark visited vertices)
- **BC.C** (betweenness centrality)
- **Radii.C** (graph eccentricity estimation)
- **Components.C** (connected components)
- **BellmanFord.C** (Bellman-Ford shortest paths)
- **PageRank.C**
- **PageRankDelta.C**
- **BFSCC.C** (connected components based on BFS)
- **KCore.C** (computes k-cores of the graph)

## Eccentricity Estimation

Code for eccentricity estimation is available in the apps/eccentricity/ directory:

- **kBFS-Ecc.C** (2 passes of multiple BFS's)
- **kBFS-1Phase-Ecc.C** (1 pass of multiple BFS's)
- **FM-Ecc.C** (estimation using Flajolet-Martin counters; an implementation of a variant of HADI from *TKDD '11*)
- **LogLog-Ecc.C** (estimation using LogLog counters; an implementation of a variant of HyperANF from *WWW '11*)
- **RV.C** (parallel implementation of the algorithm by Roditty and Vassilevska Williams from *STOC '13*)
- **CLRSTV.C** (parallel implementation of a variant of the algorithm by Chechik, Larkin, Roditty, Schoenebeck, Tarjan, and Vassilevska Williams from *SODA '14*)
- **kBFS-Exact.C** (exact algorithm using multiple BFS's)
- **TK.C** (a parallel implementation of the exact algorithm by Takes and Kosters from *Algorithms '13*)
- **Simple-Approx-Ecc.C** (simple 2-approximation algorithm)

Follow the same instructions as above for compilation, but from the apps/eccentricity/ directory.

For kBFS-Ecc.C, kBFS-1Phase-Ecc.C, FM-Ecc.C, LogLog-Ecc.C, and kBFS-Exact.C, the "-r" flag followed by an integer indicates the maximum number of words to associate with each vertex. For all implementations, the "-s" flag should be used as the current implementations are designed for undirected graphs. To output the eccentricity estimates to a file, use the "-out" flag followed by the name of the output file. The file format is one integer per line, with the eccentricity estimate for vertex  $i$  on line  $i$ .

# **Elements of good domain-specific programming system design**

# #1: good systems identify the most important cases, and provide most benefit in these situations

- **Structure of code mimics the natural structure of problems in the domain**
  - **Halide: pixel-wise view of filters:  $\text{pixel}(x,y)$  computed as expression of these input pixel values**
  - **Graph processing algorithms are designed in terms of per-vertex operations**
- **Efficient expression: common operations are easy and intuitive to express**
- **Efficient implementation: the most important optimizations in the domain are performed by the system for the programmer**
  - **My experience: a parallel programming system with “convenient” abstractions that precludes best-known implementation strategies will almost always fail**

# #2: good systems are usually simple systems

- **They have a small number of key primitives and operations**
  - **Ligra: only two operations! (vertexmap and edgemap)**
  - **GraphLab: run computation per vertex, trigger new work by signaling**
    - **But GraphLab gets messy with all the scheduling options**
  - **Halide: a few scheduling primitives for describing loop nests**
  - **Hadoop: map + reduce**
- **Allows compiler/runtime to focus on optimizing these primitives**
  - **Provide parallel implementations, utilize appropriate hardware**
- **Common question that good architects ask: “do we really need that?”**  
**(can this concept be reduced to a primitive we already have?)**
  - **For every domain-specific primitive in the system: there better be a strong performance or expressivity justification for its existence**

# #3: good primitives compose

- **Composition of primitives allows for wide application scope, even if scope remains limited to a domain**
  - e.g., frameworks discussed today support a wide variety of graph algorithms
  - Halide's loop ordering + loop interleaving schedule primitives allow for expression of wide range of schedules
- **Composition often allows optimization to generalizable**
  - If system can optimize A and optimize B, then it can optimize programs that combine A and B
- **Sign that a feature should not be added (or added in a different way):**
  - The new feature does not compose with all existing features in the system
- **Sign of a good design:**
  - System ultimately is used for applications original designers never anticipated

# **Optimizing graph computations**

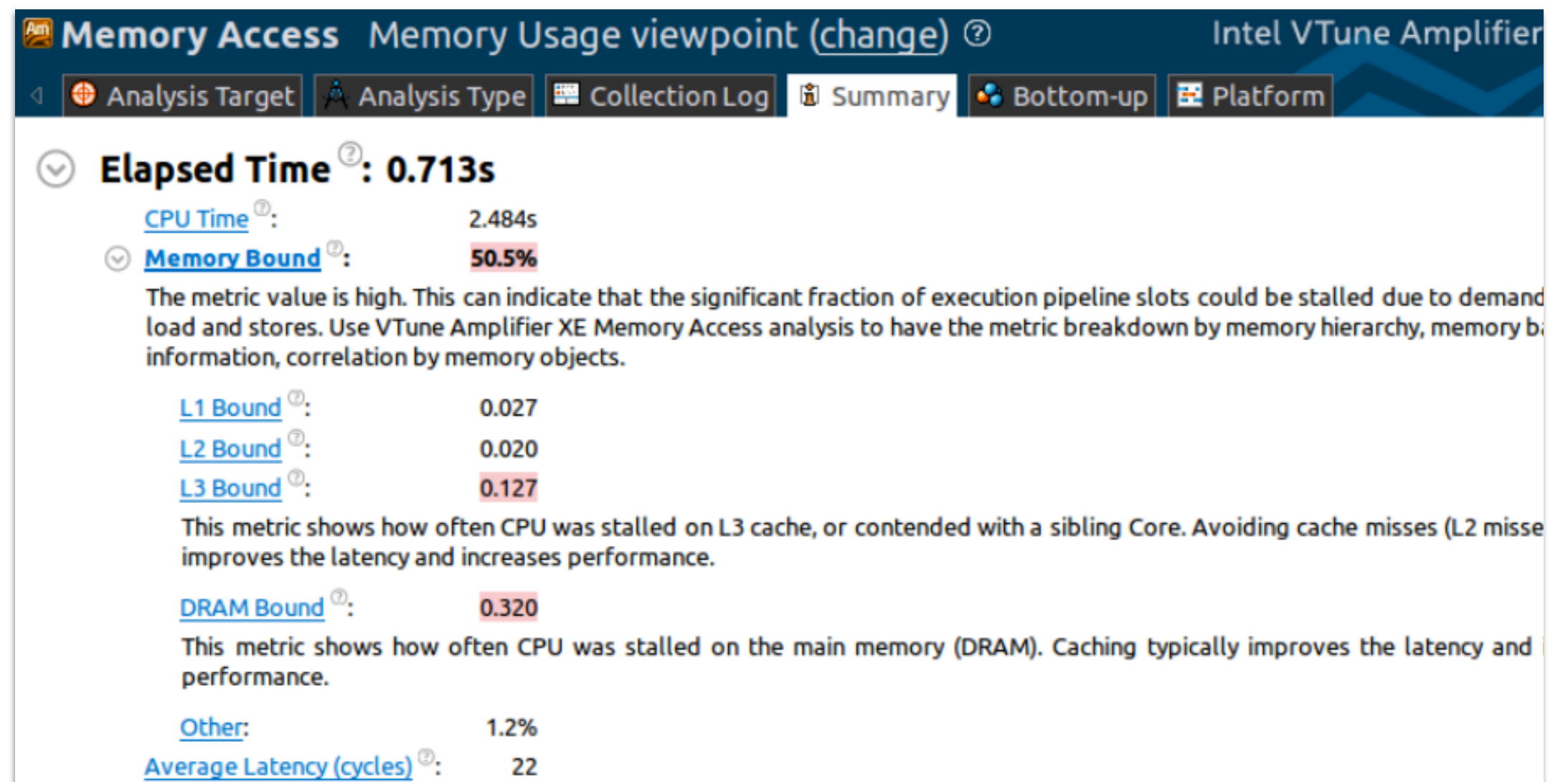
**(now we are talking about implementation)**

# Wait a minute...

- So far in this lecture, we've discussed issues such as parallelism, synchronization ...
- But you may recall from your assignment that graph processing is typically has low arithmetic intensity

Walking over edges accesses information from "random" graph vertices

VTune profiling results from Asst 4: Memory bandwidth bound!



Or just consider PageRank: ~ 1 multiply-accumulate per iteration of summation loop

$$R[i] = \frac{1 - \alpha}{N} + \alpha \sum_{j \text{ links to } i} \frac{R[j]}{\text{Outlinks}[j]}$$



# Two ideas to increase the performance of operations on large graphs \*

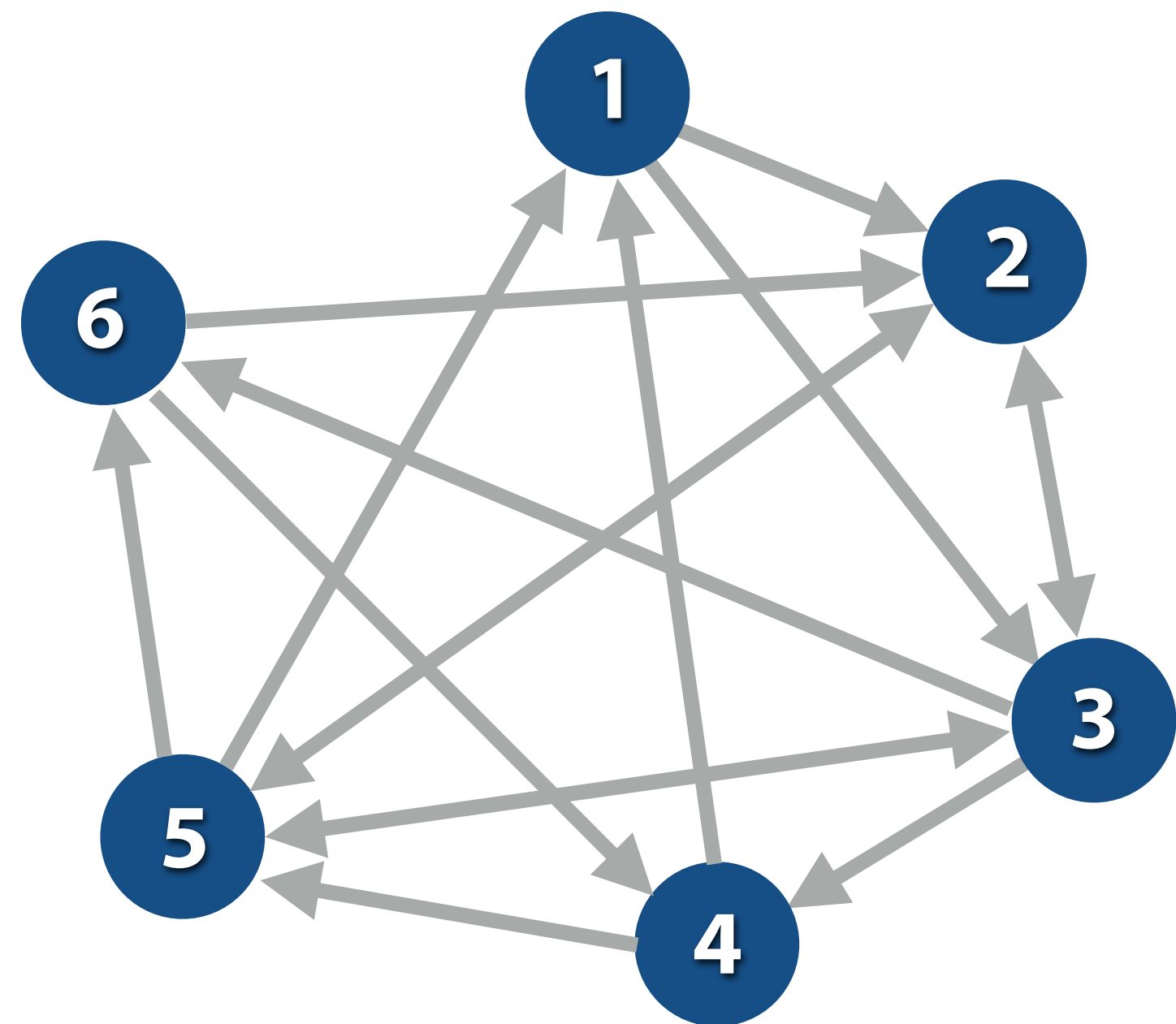
1. Reorganize graph structure to increase locality
2. Compress the graph

\* Both optimizations might be performed by a framework without application knowledge

# Recall: directed graph representation

<b>Vertex Id</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>
<b>Outgoing Edges</b>	2 3	3 5	2 4 5 6	1 5	1 2 3 6	2 4

<b>Vertex Id</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>
<b>Incoming Edges</b>	4 5	1 3 5 6	1 2 5	3 6	2 3 4	3 6



# Memory footprint challenge of large graphs

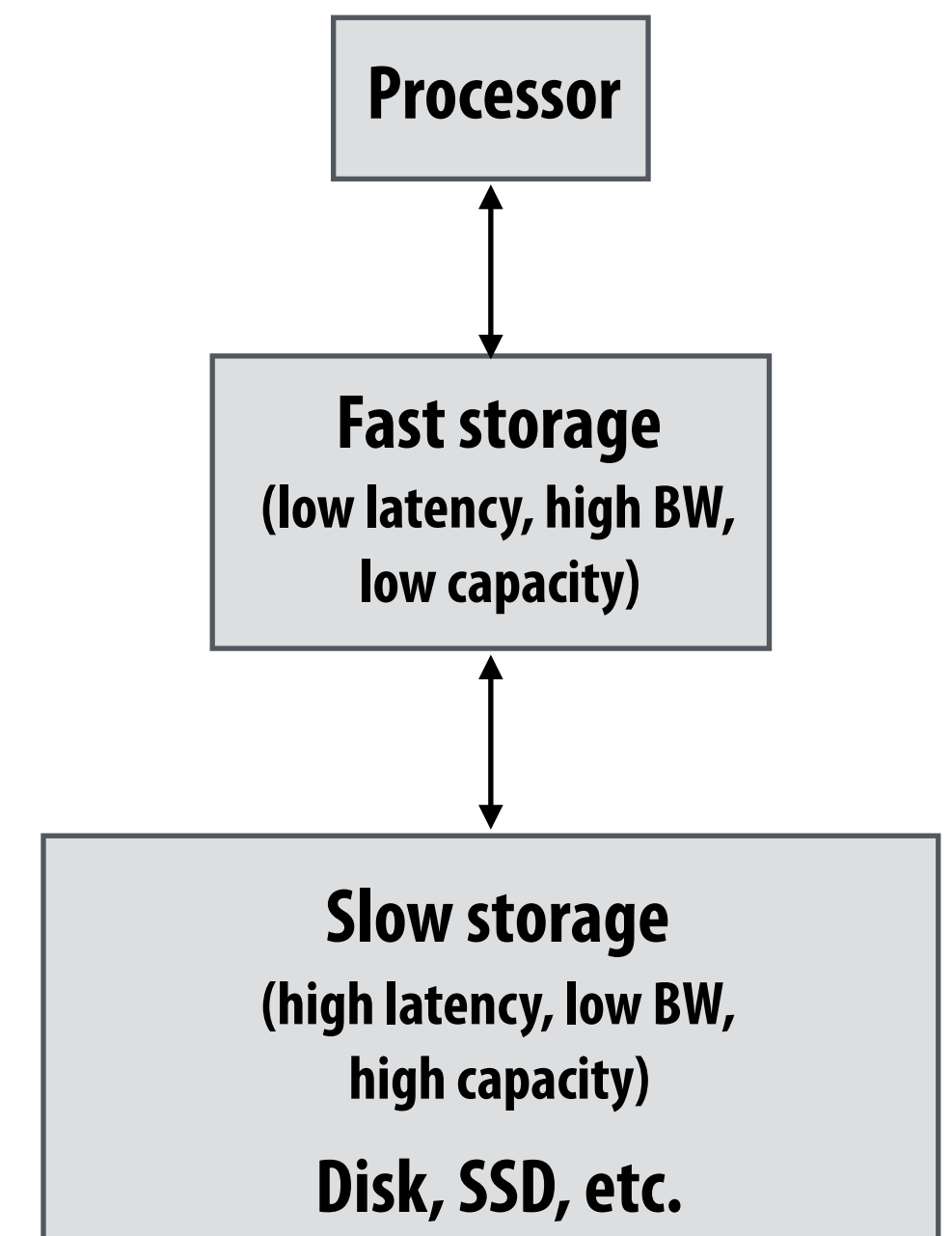
- **Challenge: cannot fit all edges in memory for large graphs**
  - Consider representation of graph from your programming assignment:
    - Each edge represented twice in graph structure (as incoming/outgoing edge)
    - 8 bytes per edge to represent adjacency
  - May also need to store per-edge values (e.g., 4 bytes for a per-edge weight)
  - 1 billion edges (modest): ~12 GB of memory for edge information
  - Algorithm may need multiple copies of per-edge structures (current, prev data, etc.)
- **Could employ cluster of machines to store graph in memory**
  - Rather than store graph on disk
- **Would prefer to process large graphs on a single machine**
  - Managing clusters of machines is difficult
  - Partitioning graphs is expensive (also needs a lot of memory) and difficult

# “Streaming” graph computations

- Graph operations make “random” access to graph data (edges adjacent to vertex  $v$  may be distributed arbitrarily throughout storage)
  - Single pass over graph’s edges might make billions of fine-grained accesses to disk

- Streaming data access pattern

- Make large, predictable data accesses to slow storage (achieve high bandwidth data transfer)
- Load data from slow storage into fast storage\*, then reuse it as much as possible before discarding it (achieve high arithmetic intensity)
- **Can we modify the graph data structure so that data access requires only a small number of efficient bulk loads/stores from slow storage?**



\* By fast storage, in this context I mean DRAM. However, techniques for streaming from disk into memory would also apply to streaming from memory into a processor’s cache

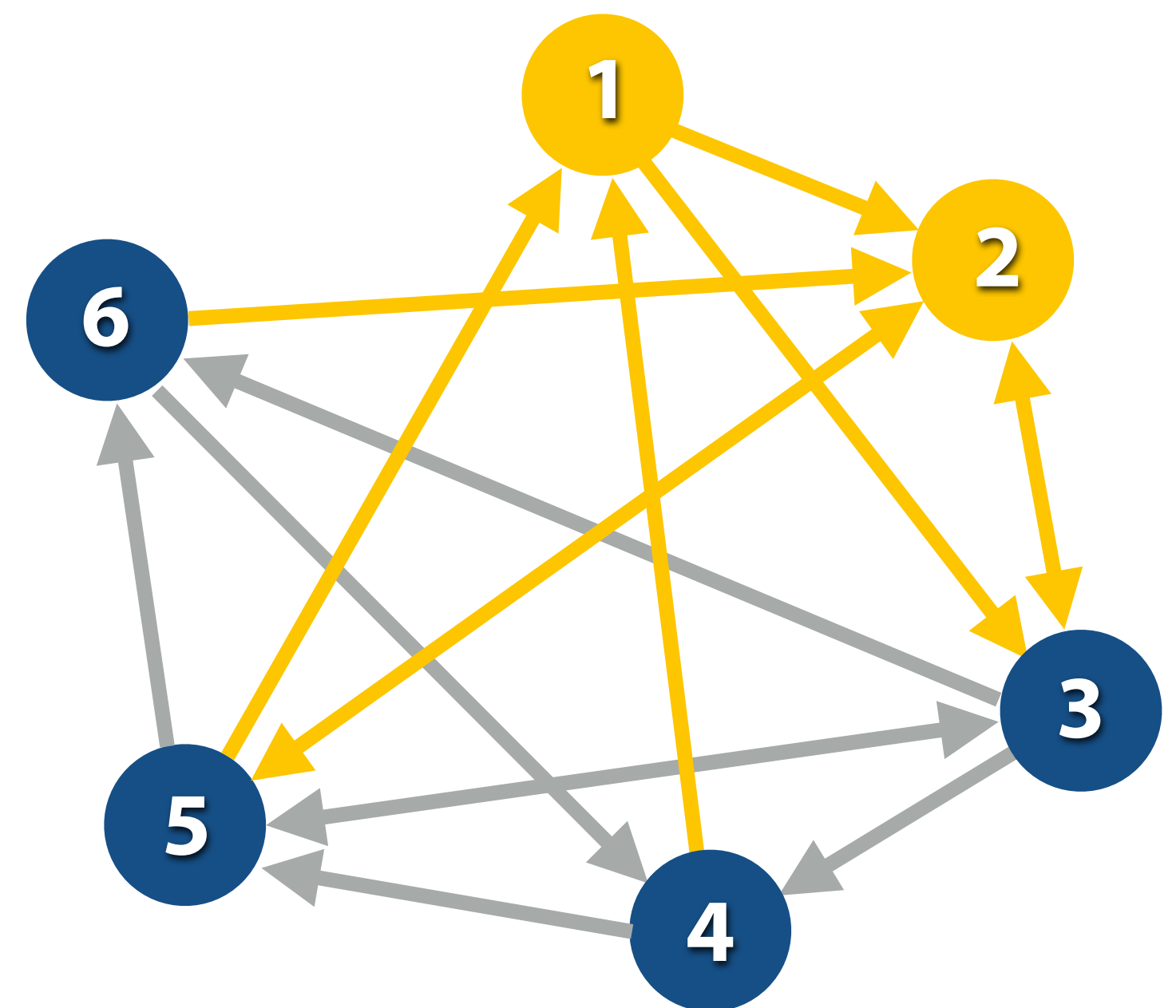
# Sharded graph representation

GraphChi: Large-scale graph computation on just a PC  
[Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)			Shard 2: vertices (3-4)			Shard 3: vertices (5-6)		
src	dst	value	src	dst	value	src	dst	value
1	2	0.3	1	3	0.4	2	5	0.6
3	2	0.2	2	3	0.9	3	5	0.9
4	1	0.8	3	4	0.15	4	6	0.85
5	1	0.25	5	3	0.2	5	6	0.3
	2	0.6	6	4	0.9	6	5	0.2
6	2	0.1						

Yellow = data required to process subgraph containing vertices in shard 1



**Notice: to construct subgraph containing vertices in shard 1 and their incoming and outgoing edges, only need to load contiguous information from other P-1 shards**

**Writes to updated outgoing edges require P-1 bulk writes**

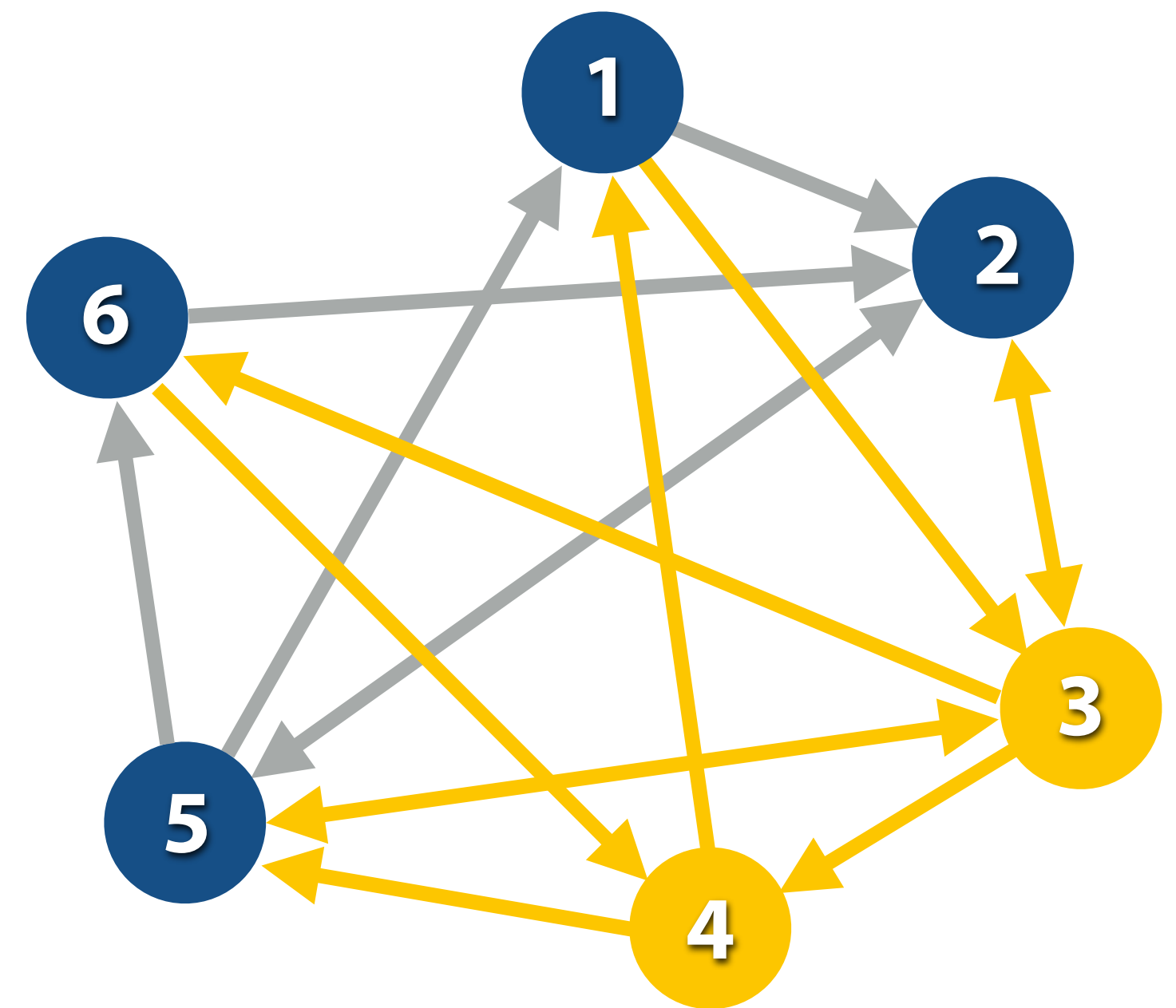
# Sharded graph representation

GraphChi: Large-scale graph computation on just a PC  
[Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)			Shard 2: vertices (3-4)			Shard 3: vertices (5-6)		
src	dst	value	src	dst	value	src	dst	value
1	2	0.3	1	3	0.4	2	5	0.6
3	2	0.2	2	3	0.9	3	5	0.9
4	1	0.8	3	4	0.15	6	0.85	
5	1	0.25	5	3	0.2	4	5	0.3
	2	0.6	6	4	0.9	5	6	0.2
6	2	0.1						

Yellow = data required to process subgraph containing vertices in shard 2



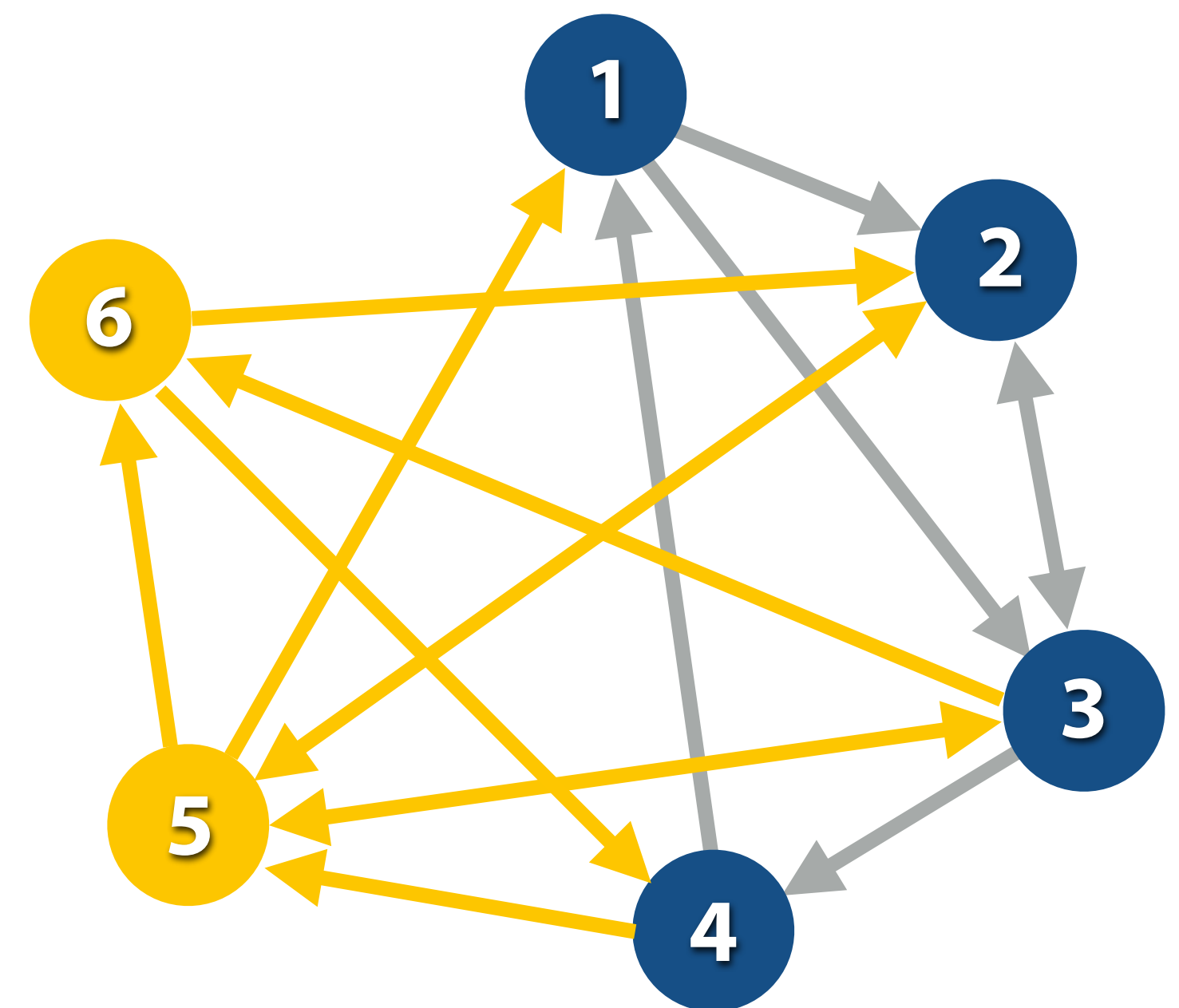
# Sharded graph representation

GraphChi: Large-scale graph computation on just a PC  
[Kryola et al. 2013]

- Partition graph vertices into intervals (sized so that subgraph for interval fits in memory)
- Vertices and only incoming edges to these vertices are stored together in a shard
- Sort edges in a shard by source vertex id

Shard 1: vertices (1-2)			Shard 2: vertices (3-4)			Shard 3: vertices (5-6)		
src	dst	value	src	dst	value	src	dst	value
1	2	0.3	1	3	0.4	2	5	0.6
3	2	0.2	2	3	0.9	3	5	0.9
4	1	0.8	3	4	0.15	6	0.85	
5	1	0.25	5	3	0.2	4	5	0.3
	2	0.6	6	4	0.9	5	6	0.2
6	2	0.1						

Yellow = data required to process subgraph containing vertices in shard 3



**Observe: due to sorting of incoming edges, iterating over all intervals results in contiguous sliding window over the shards**

# Putting it all together: looping over all graph edges

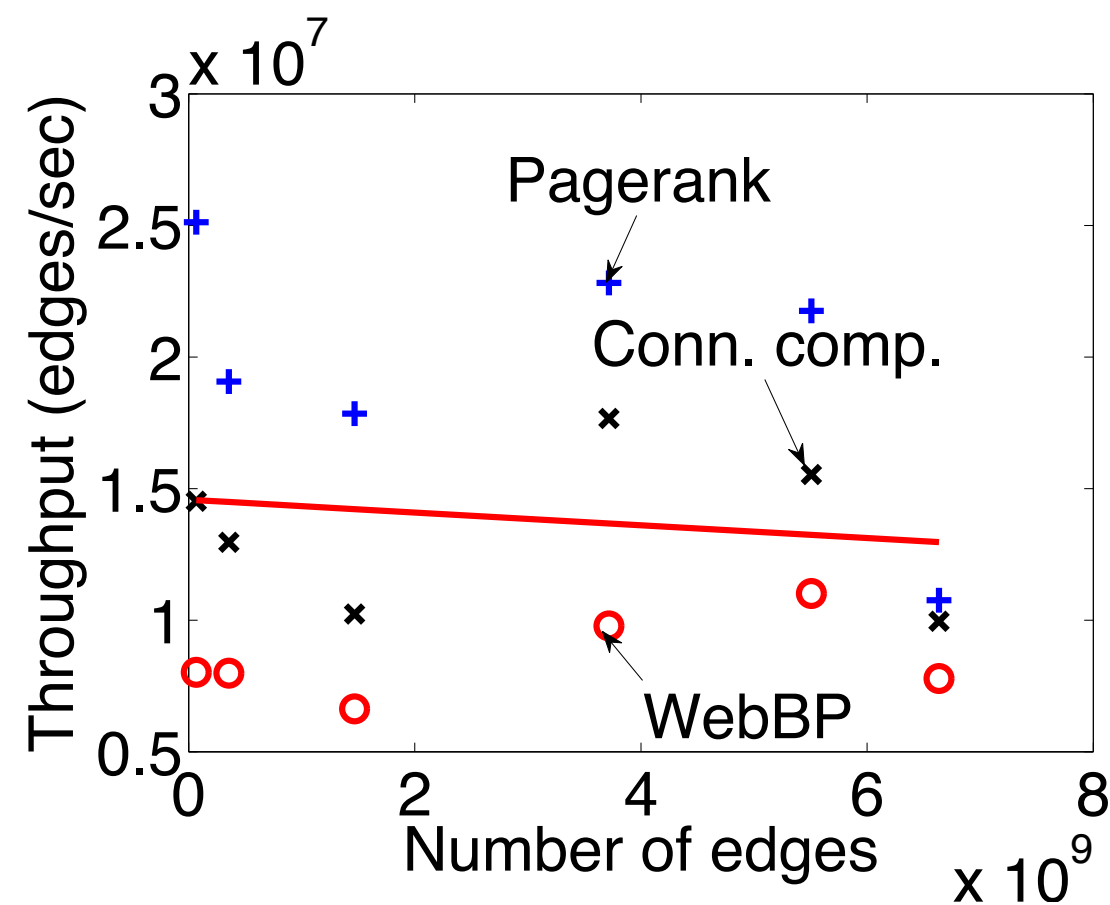
**For each partition  $i$  of vertices:**

- **Load shard  $i$  (contains all incoming edges)**
- **For each other shard  $s$** 
  - **Load section of  $s$  containing data for edges leaving  $i$  and entering  $s$**
- **Construct subgraph in memory**
- **Do processing on subgraph**

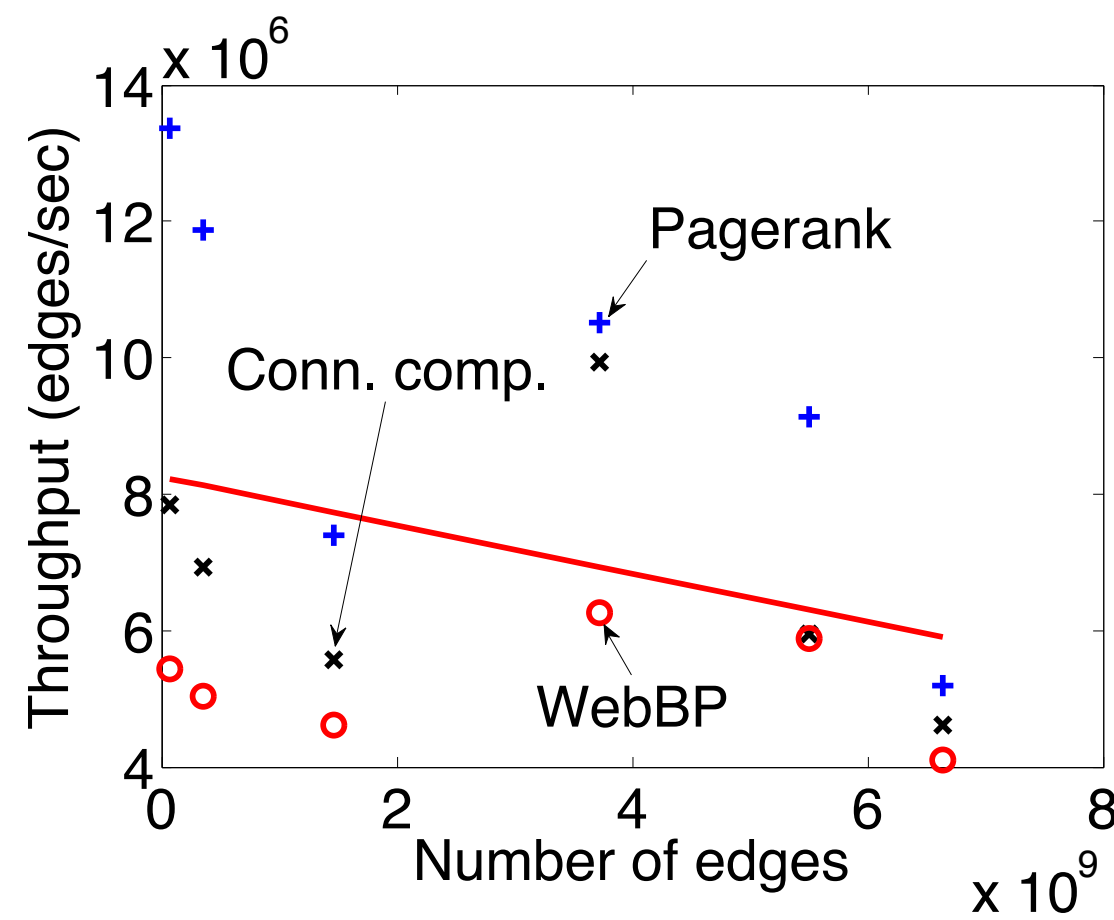
**Note: a good implementation could hide disk I/O by prefetching data for next iteration of loop**



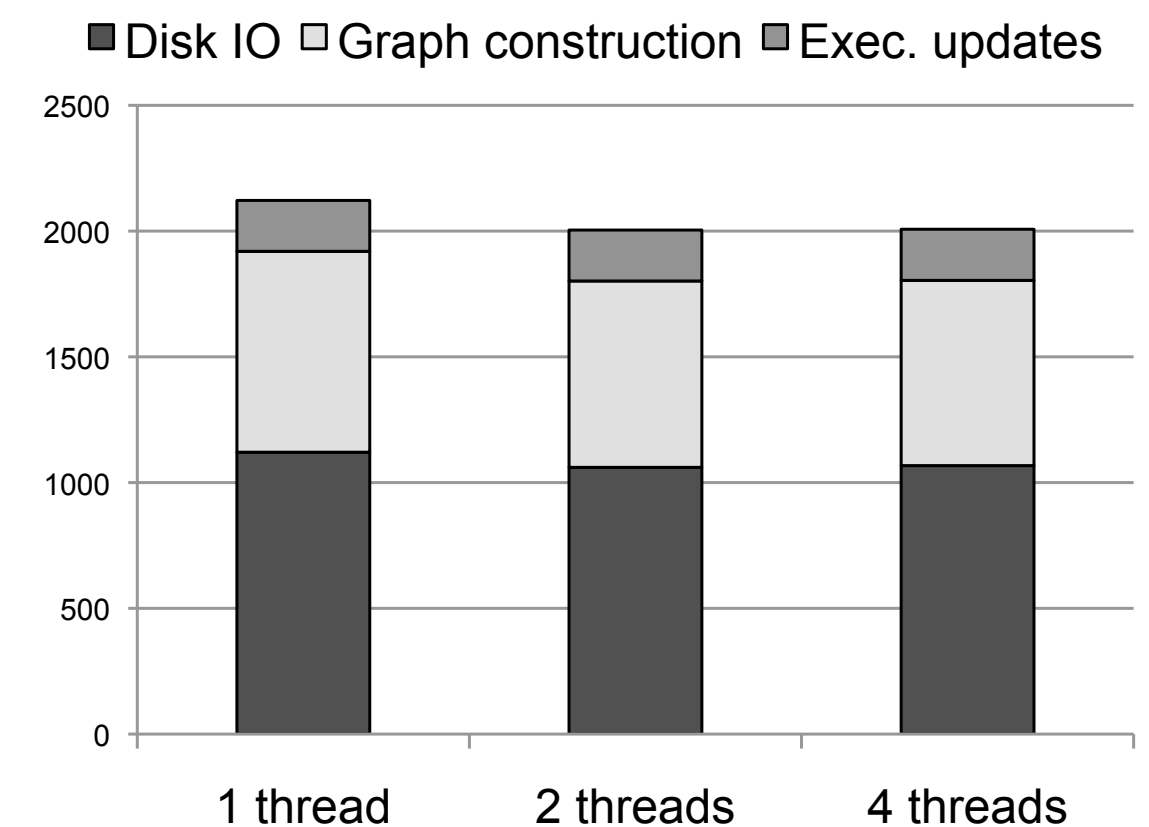
# Performance on a Mac mini (8 GB RAM)



(a) Performance: SSD



(b) Performance : Hard drive



(c) Runtime breakdown

**Throughput (edges/sec) remains stable as graph size is increased**

- **Desirable property: throughput (edges/sec) largely invariant of dataset size**

# Graph compression

- **Recall: graph operations are often bandwidth bound**
- **Implication: using additional CPU instructions to reduce BW requirements can benefit overall performance (the processor would be waiting on memory anyway, so use it to decompress data!)**
- **Idea: store graph compressed in memory, decompress on-the-fly when operation wants to read data**

# Compressing an edge list

Vertex Id 32

Outgoing Edges 1001 10 5 30 6 1025 200000 1010 1024 100000 1030 275000

---

## 1. Sort edges for each vertex

5 6 10 30 1001 1010 1024 1025 1030 100000 200000 275000

## 2. Compute differences

5 6 10 30 1001 1010 1024 1025 1030 100000 200000 275000  
0 1 4 20 971 9 14 1 5 98070 100000 75000

## 3. Group into sections requiring same number of bytes

relative to vertex index

5 6 10 30 1001 1010 1024 1025 1030 100000 200000 275000  
-27 1 4 20 971 9 14 1 5 98070 100000 75000

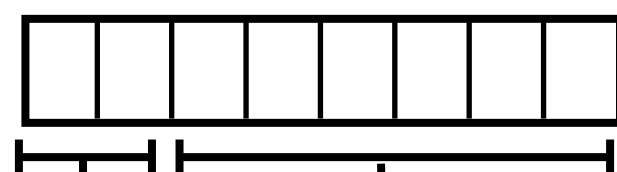
1 byte 2 bytes 1 byte 4 bytes

## 4. Encode deltas

Uncompressed encoding: 12 edges x 4 bytes = 48 bytes

Compressed encoding: 26 bytes

1-byte group header



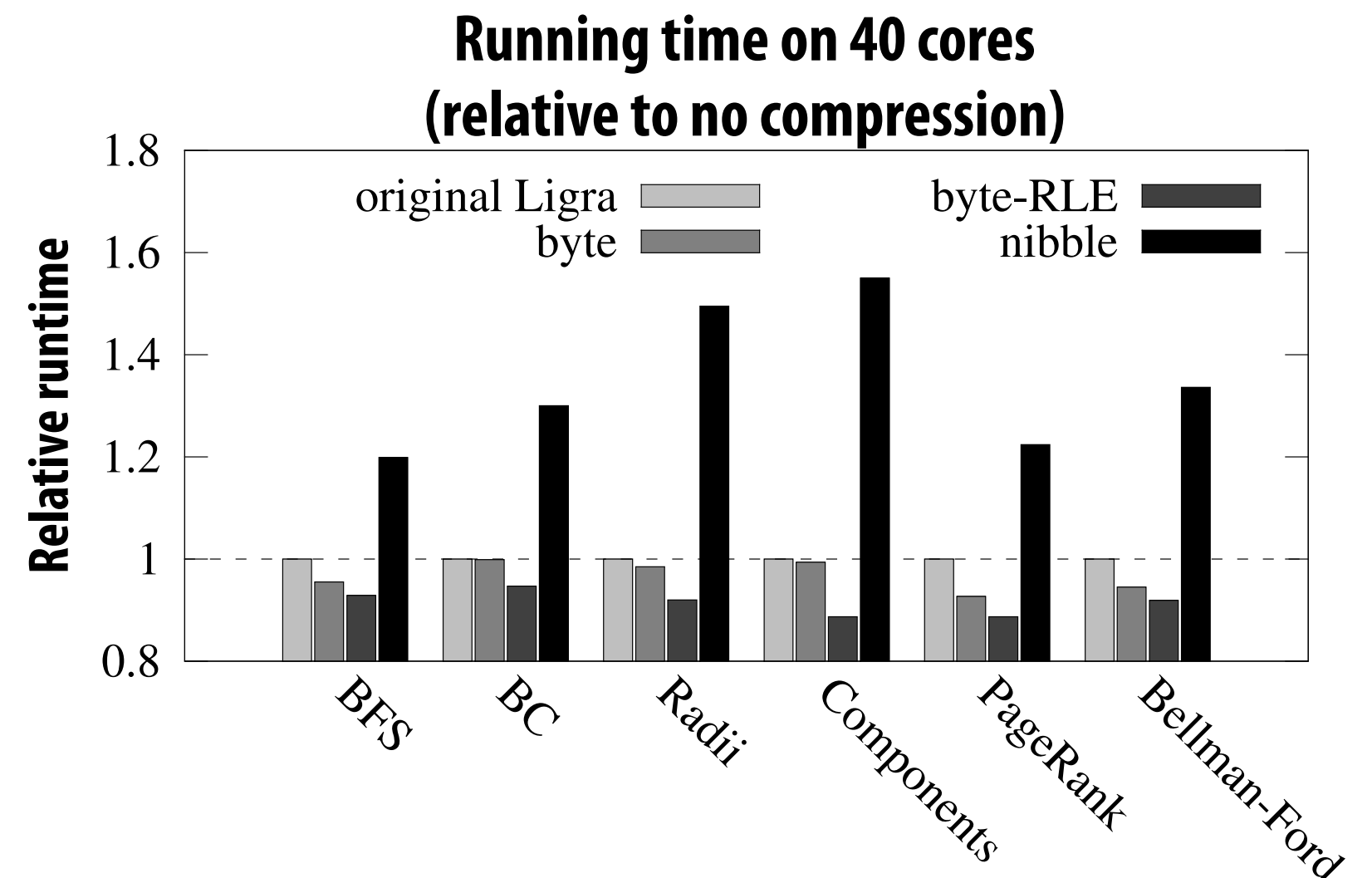
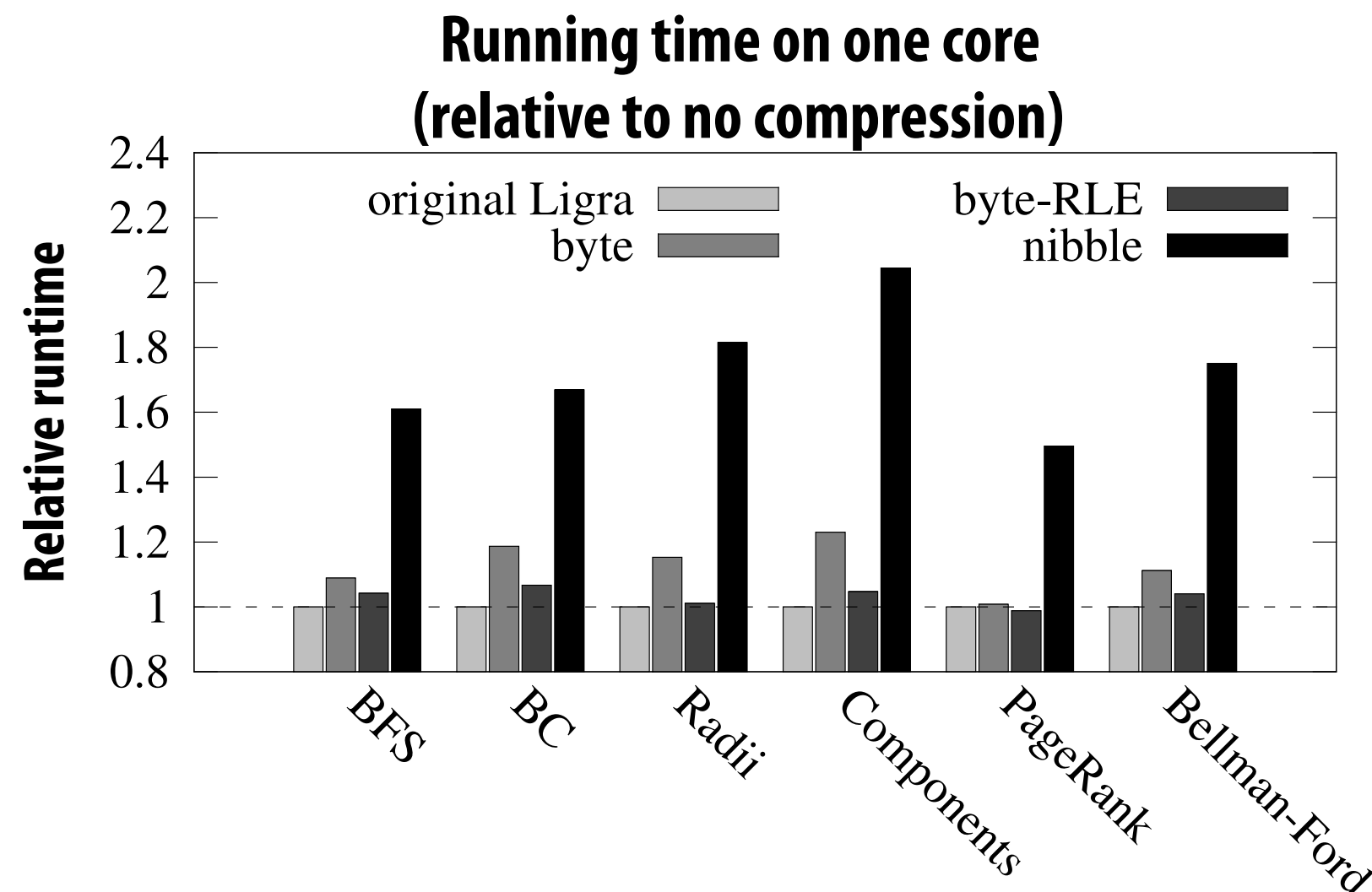
6 bits: number of edges in group

2 bits: encoding width (1, 2, 4 bytes)

[ONE\_BYTE, 4], -27, 1, 4, 20 (5 bytes)  
[TWO\_BYTE, 1], 971 (3 bytes)  
[ONE\_BYTE, 4], 9, 14, 1, 5 (5 bytes)  
[FOUR\_BYTE, 3], 98070, 100000, 75000 (13 bytes)

# Performance impact of graph compression

[Shun et al. DDC 2015]



- **Benefit of graph compression increases with higher core count, since computation is increasingly bandwidth bound**
- **Performance improves even if graphs already fit in memory**
  - **Added benefit is that compression enables larger graphs to fit in memory**

\* Different data points on graphs are different compression schemes  
(byte-RLE is the scheme on the previous slide)

# Summary

- **Analyzing large graphs is a workload of high interest**
- **High performance execution requires**
  - **Parallelism (complexity emerges from need to synchronize updates to shared vertices or edges)**
  - **Locality optimizations (restructure graph for efficient I/O)**
  - **Graph compression (reduce amount memory BW or disk I/O)**
- **Graph-processing frameworks handle many of these details, while presenting the application programmer with domain-specific abstractions that make it easy to express graph analysis operations**

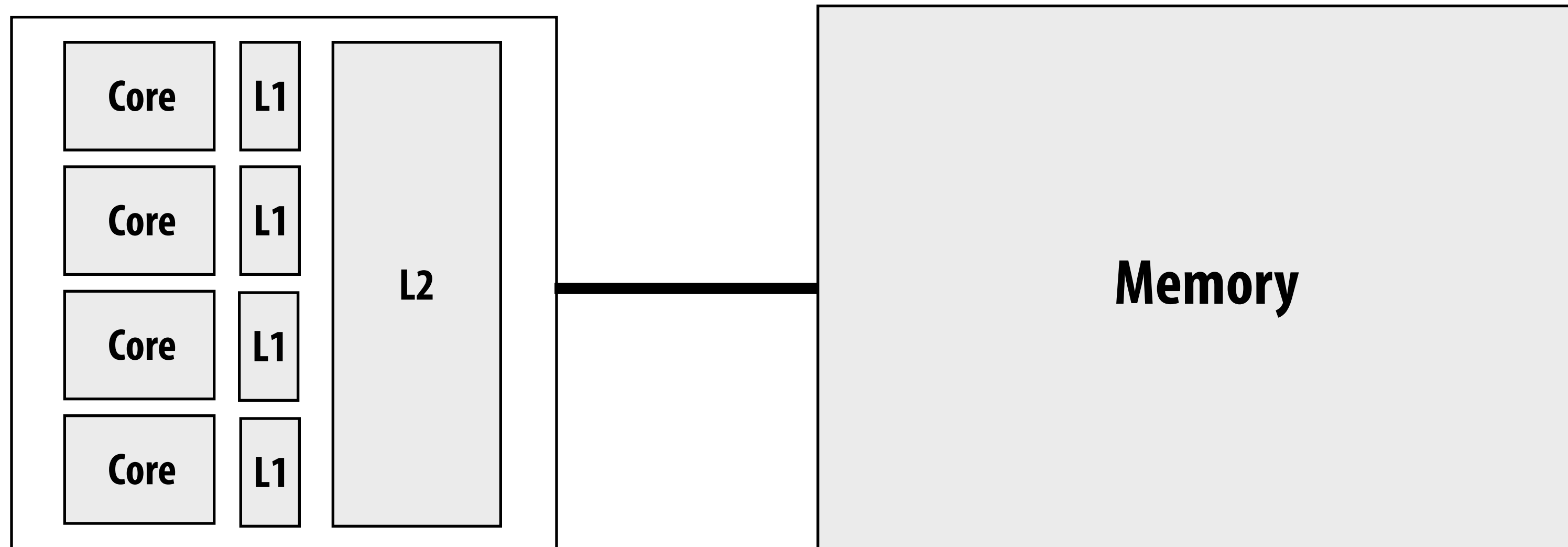
# How memory works

# Memory bandwidth limits

- **So far in this course, we've stressed the need for reducing bandwidth costs...**

# 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 on-chip local storage
  - Computations featuring data access locality can reuse data in this storage
- Common software optimization technique: reorder computation so that cached data is accessed many times before it is evicted (“blocking”, “loop fusion”, etc.)
- 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];  
}
```

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

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

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

```
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);
```

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];  
}
```

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

```
// compute E = D + (A + B) * C
```

```
fused(n, A, B, C, D, E);
```

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

```

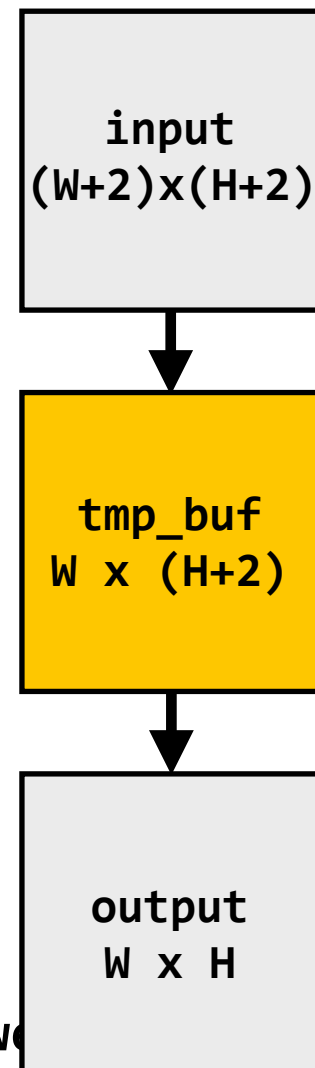
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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];

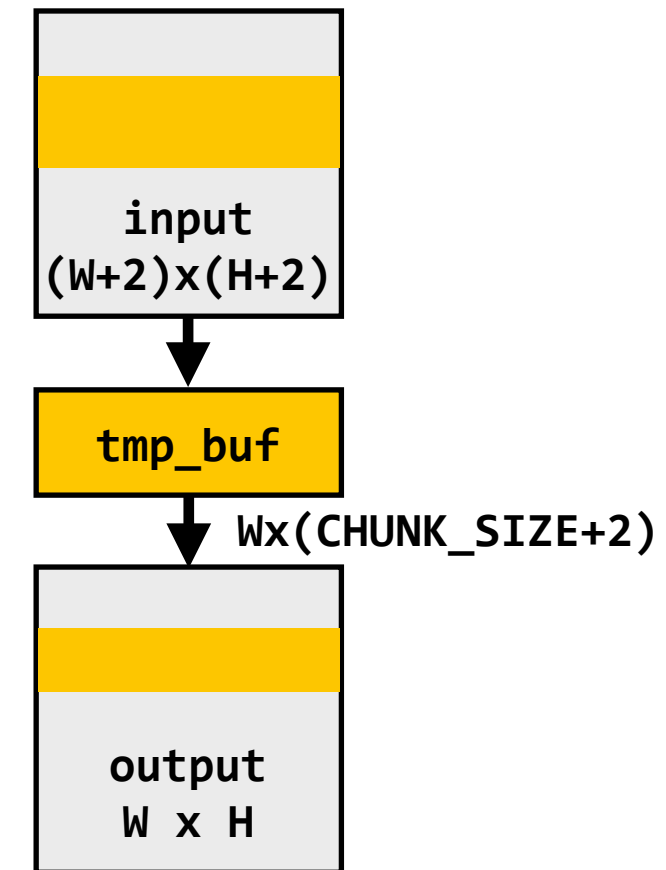
float weights[] = {1.0/3, 1.0/3, 1.0/3};

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;
    }
}

```



# Example 3: restructuring loops for locality

**Recall Apache Spark:**

**Programs are sequences of operations on collections (called RDDs)**

```
var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLowerCase());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

**Actual execution order of computation for the above lineage is similar to this...**

```
int count = 0;
while (inputFile.eof()) {
    string line = inputFile.readLine();
    string lower = line.toLowerCase();
    if (isMobileClient(lower))
        count++;
}
```

# 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 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!



# Moving data is costly!

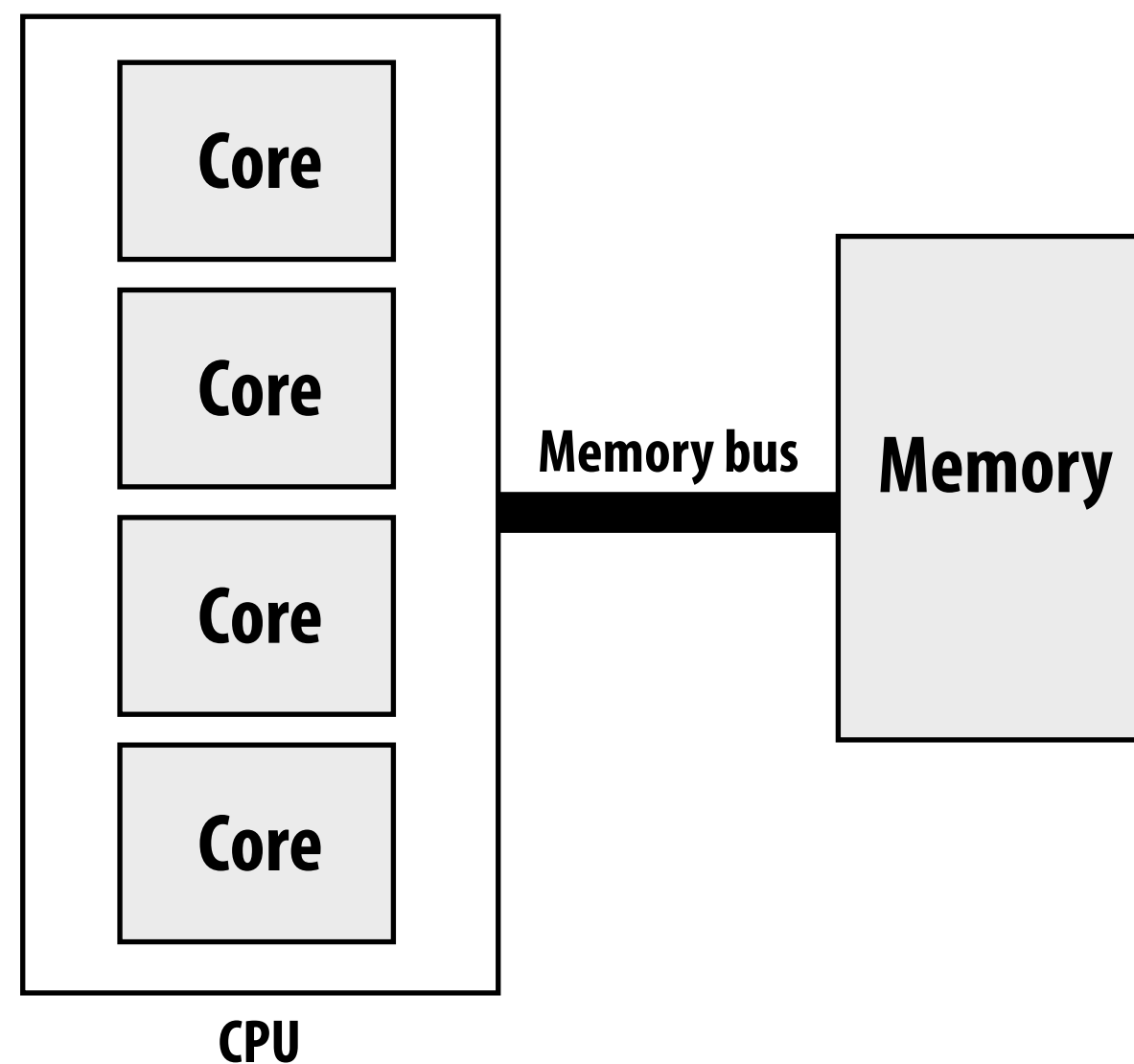
## Data movement limits performance

Many processing elements...

= higher overall rate of memory requests

= need for more memory bandwidth

(result: bandwidth-limited execution)

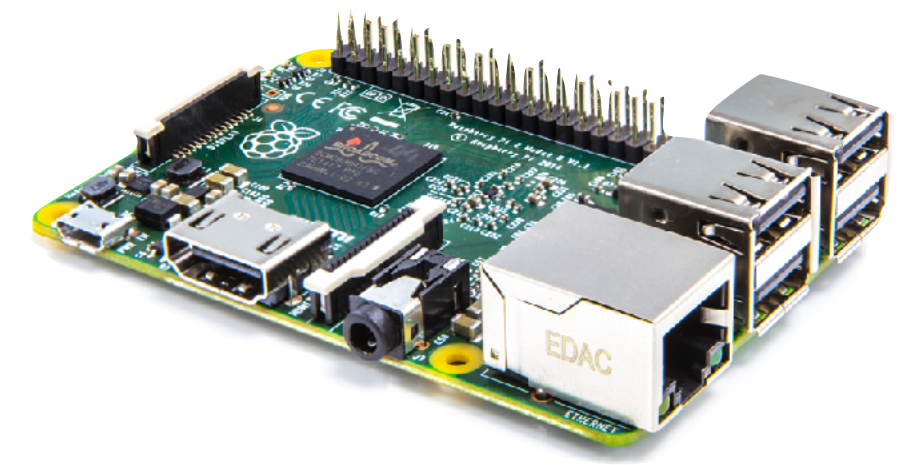


## Data movement has high energy cost

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

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

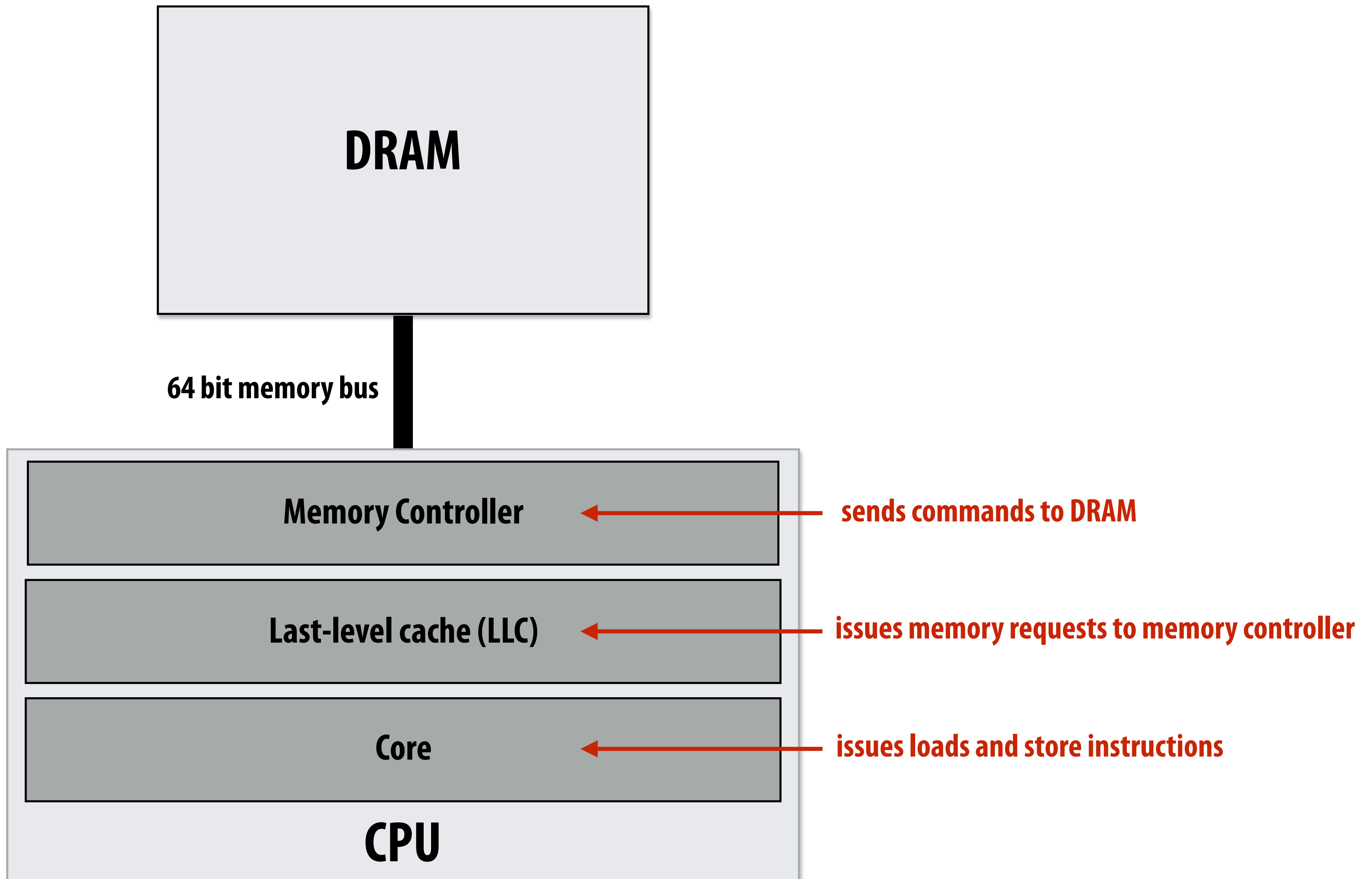
~ 640 pJ to load 32 bits from LPDDR memory



# **Accessing DRAM**

**(a basic tutorial on how DRAM works)**

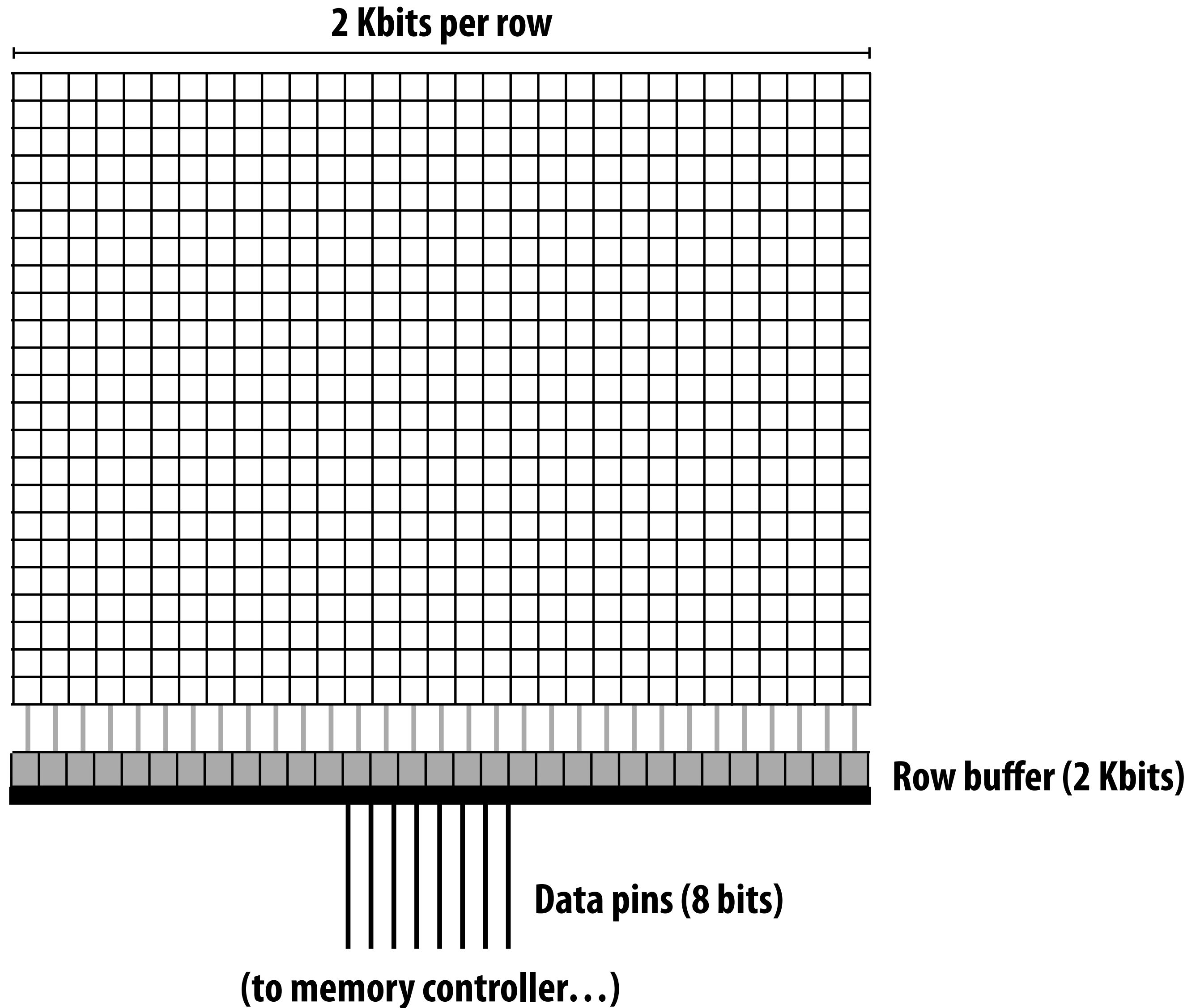
# The memory system



# DRAM array

1 transistor + capacitor per "bit"

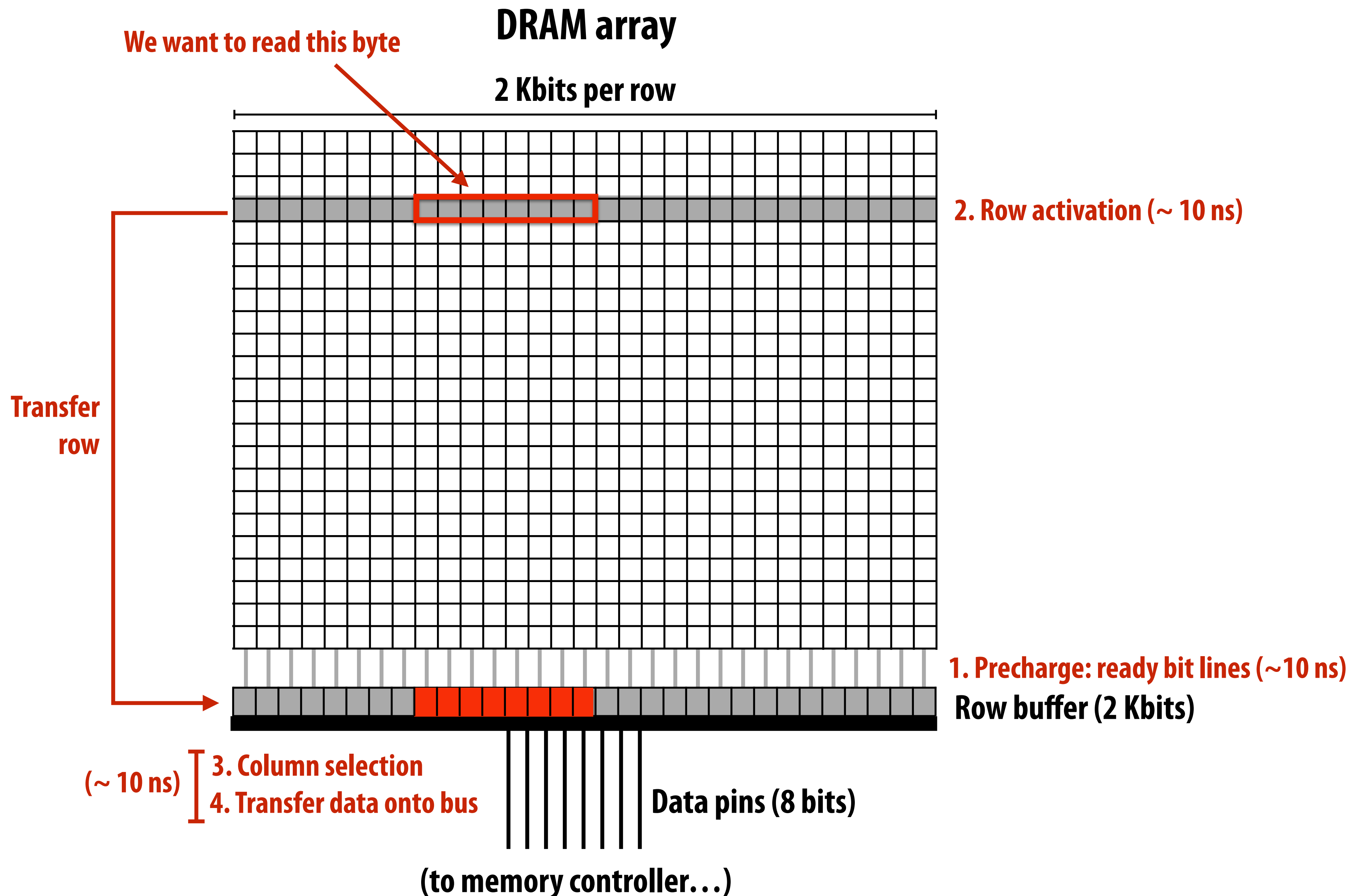
(Recall: a capacitor stores charge)





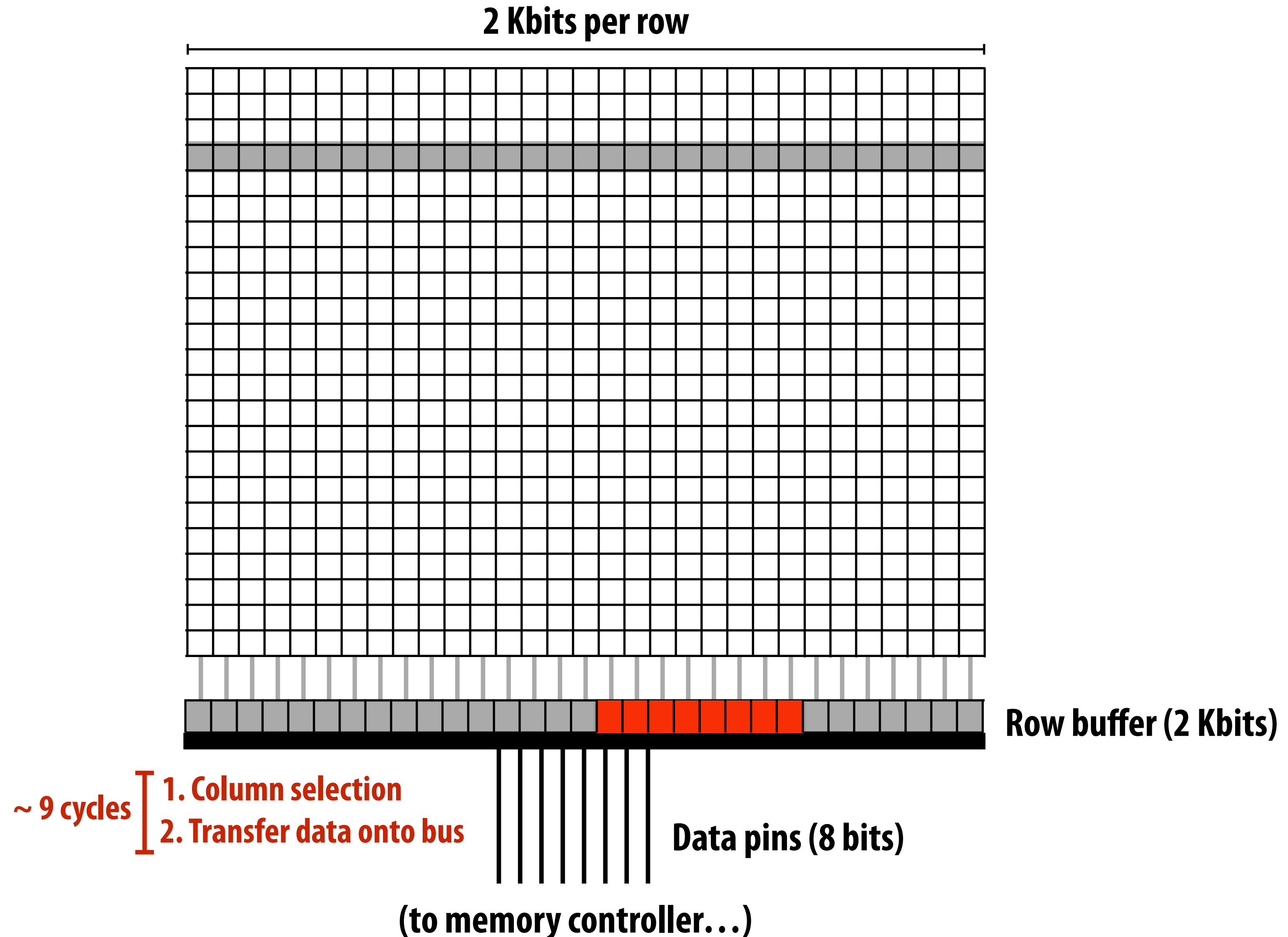
# DRAM operation (load one byte)

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

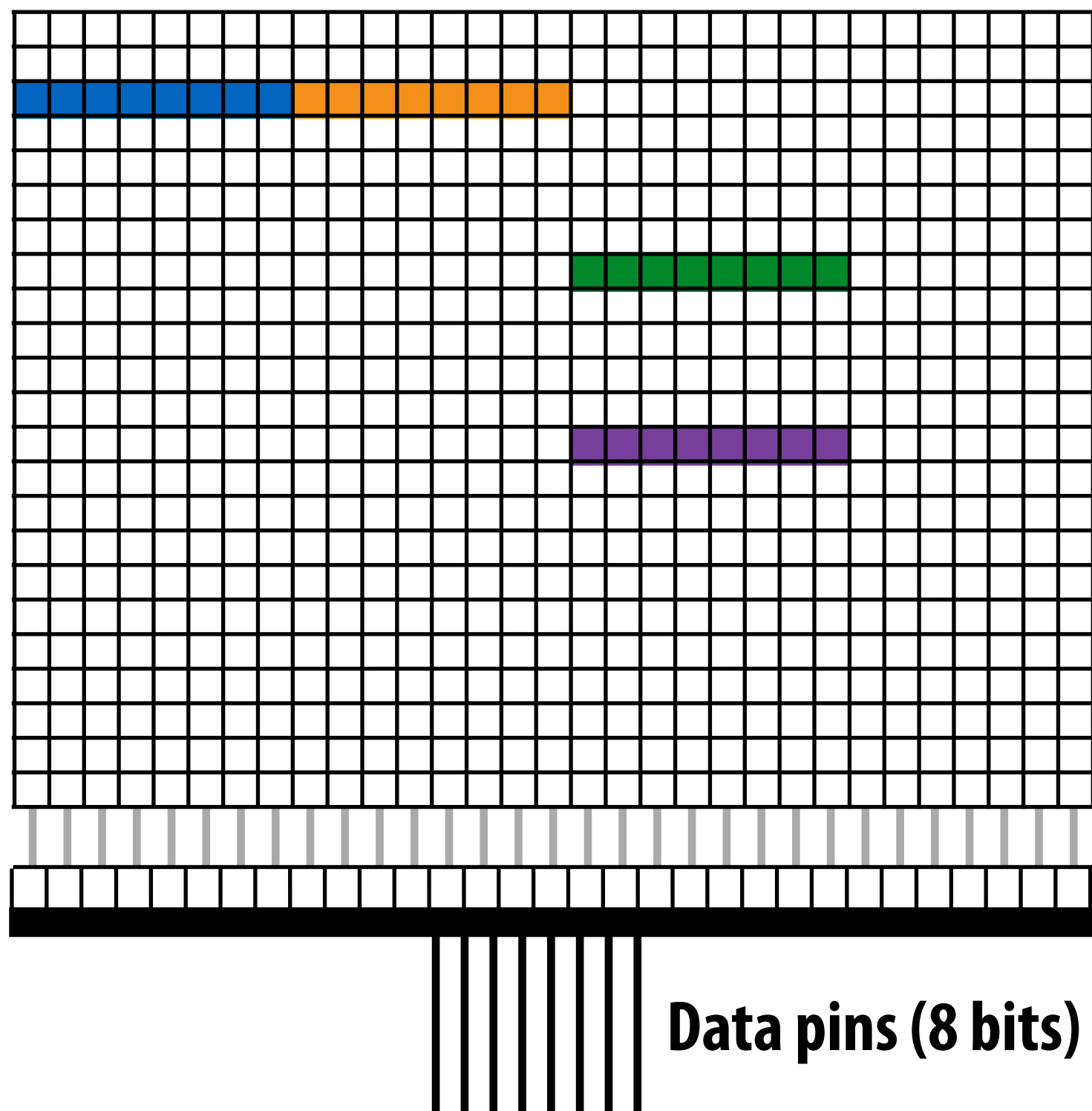
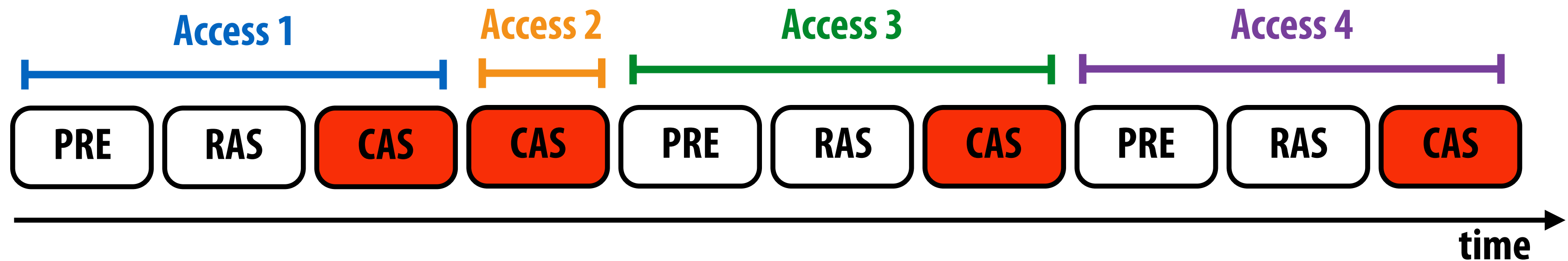


# DRAM access latency is not fixed

- **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?**

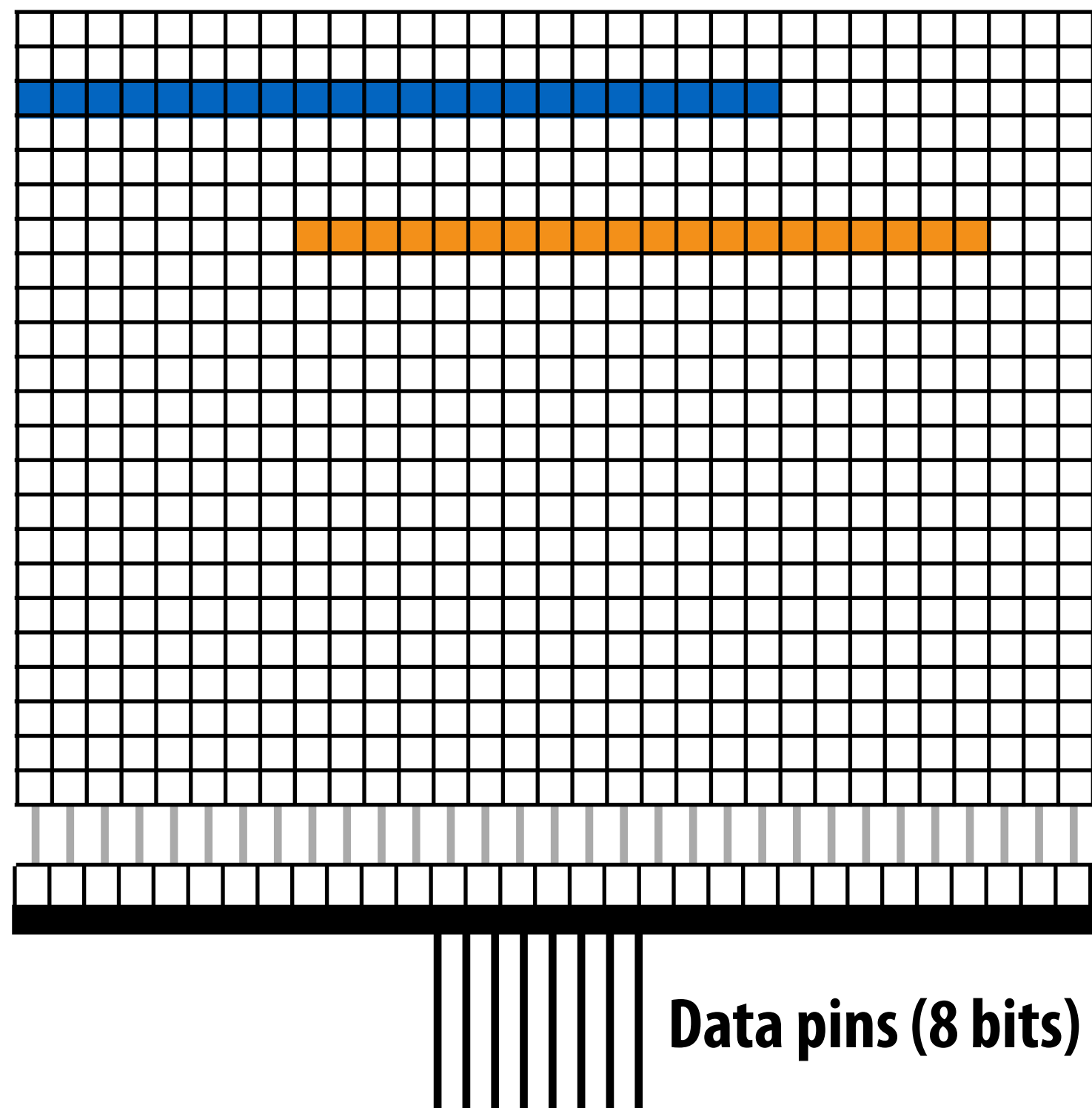
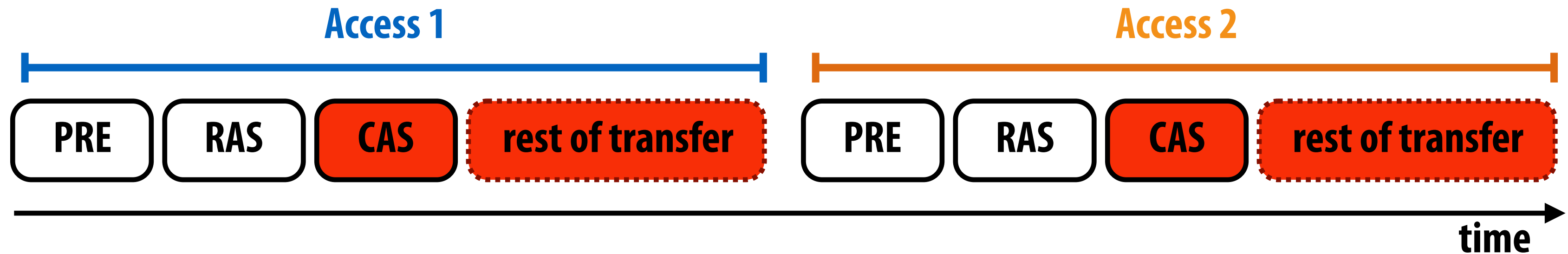
# Problem: low pin utilization due to latency of access



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

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

# DRAM burst mode

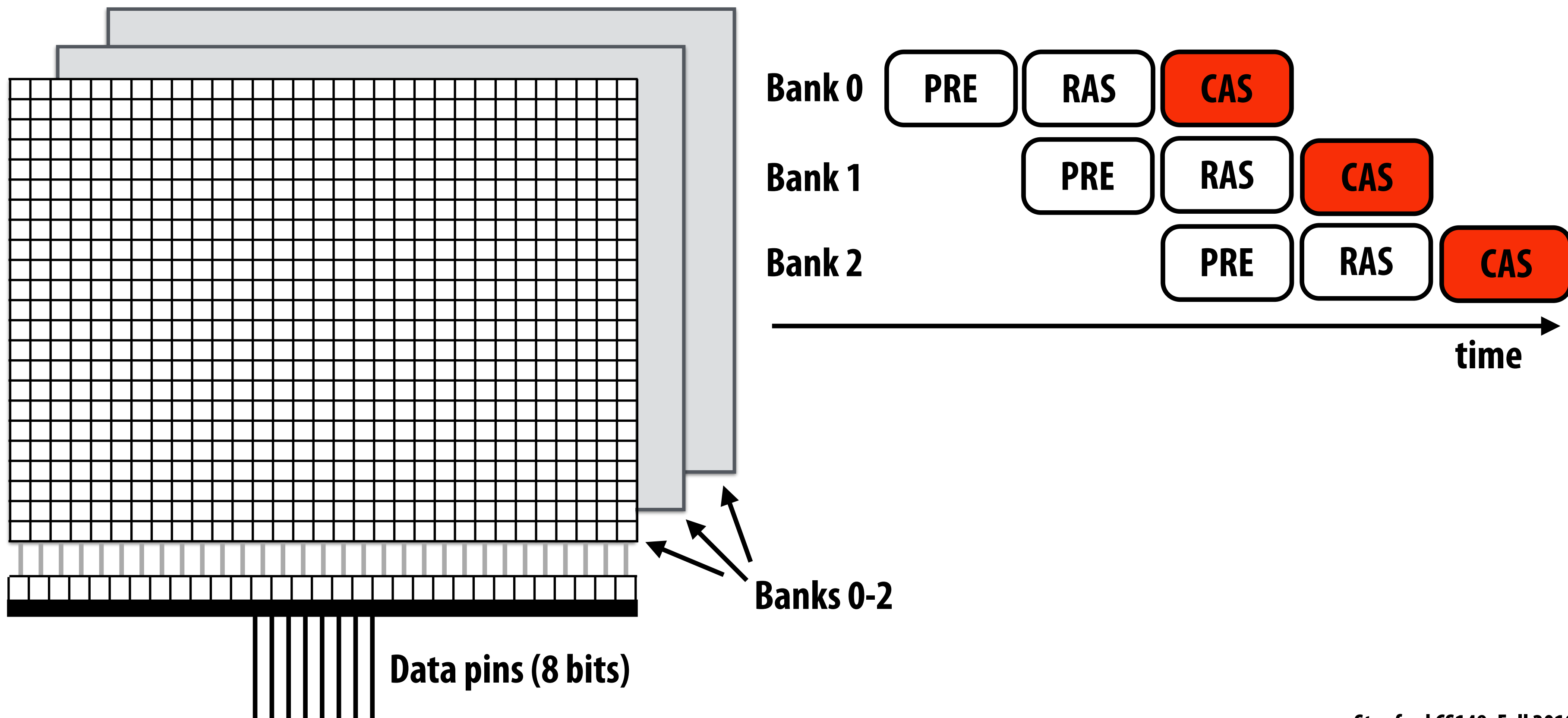


**Idea: amortize latency over larger transfers**

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

# DRAM chip consists of multiple banks

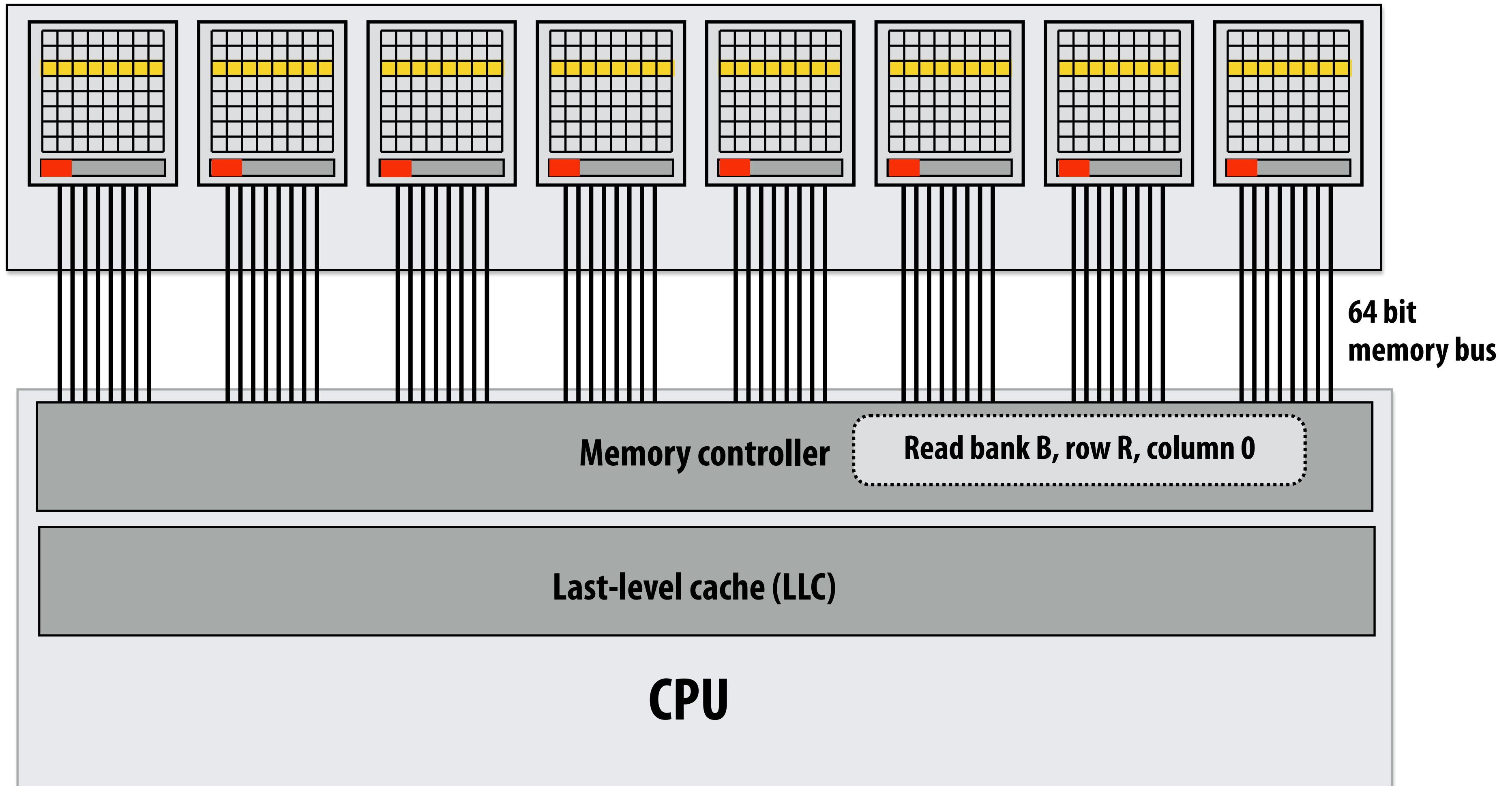
- All banks share same pins (only one transfer at a time)
- Banks allow for pipelining of memory requests
  - Precharge/activate rows/send column address to one bank while transferring data from another
  - Achieves high data pin utilization



# Organize multiple chips into a DIMM

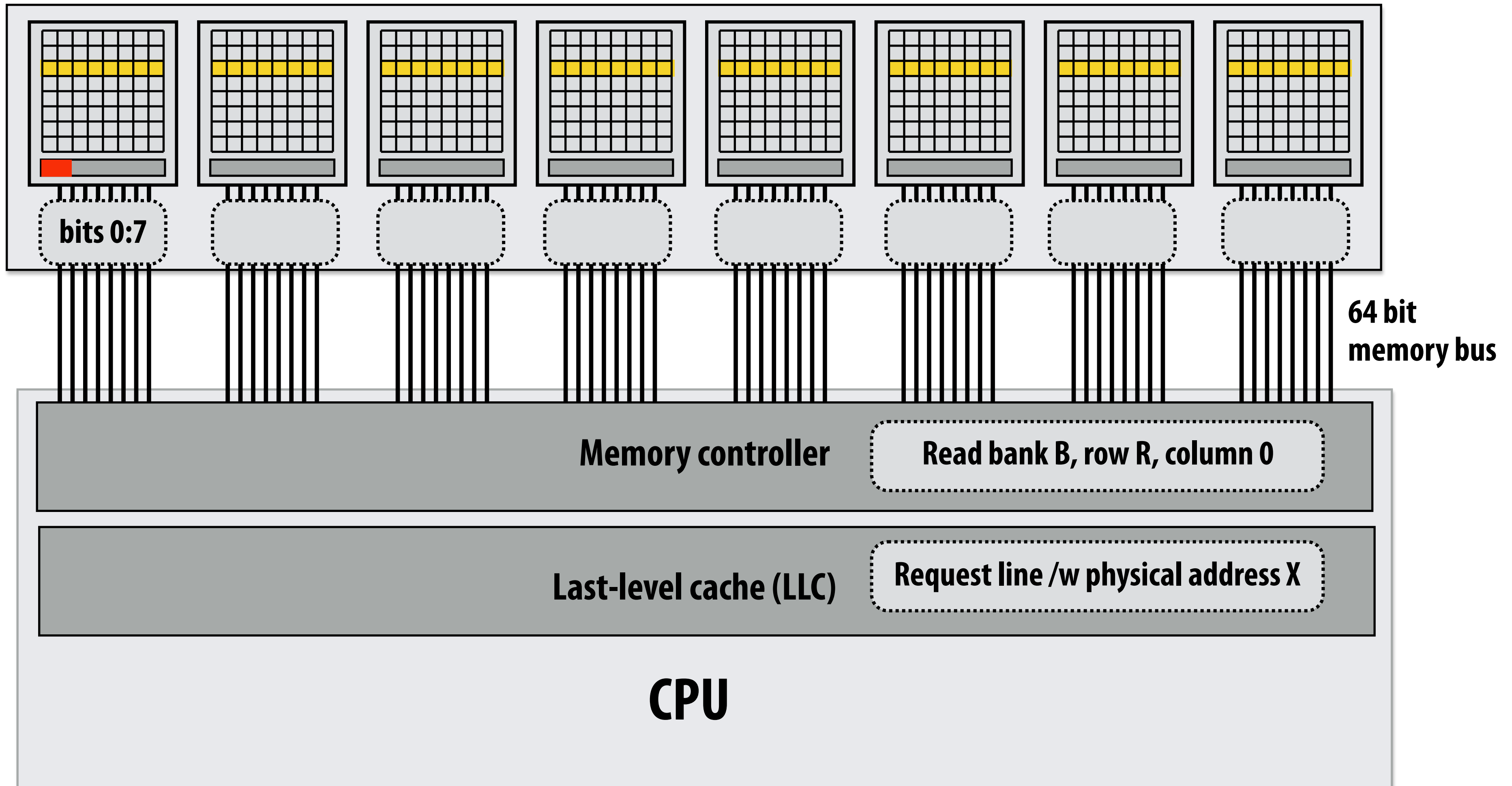
**Example: Eight DRAM chips (64-bit memory bus)**

**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)

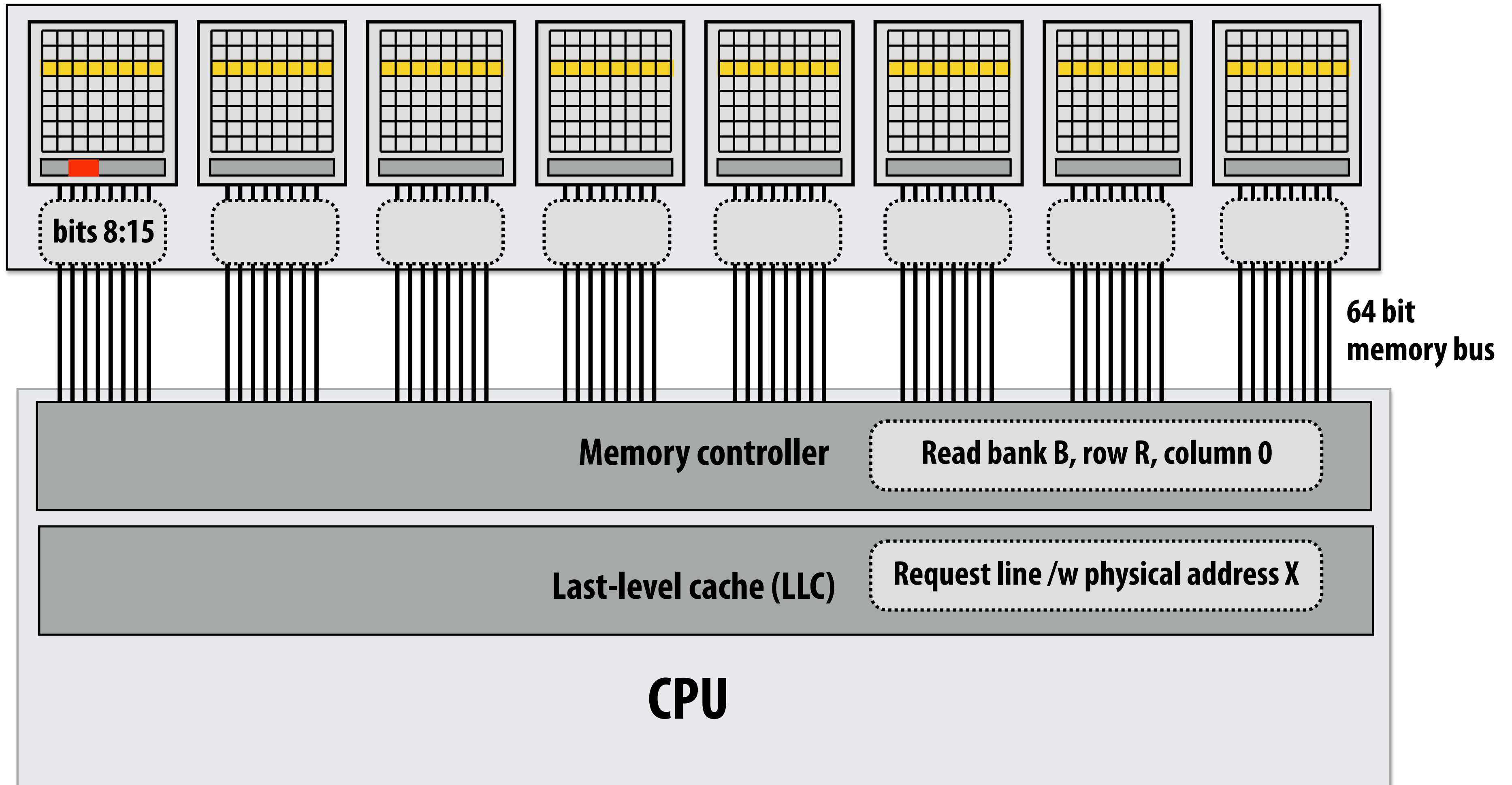
Assume: consecutive physical addresses mapped to same row of same chip  
Memory controller converts physical address to DRAM bank, row, column





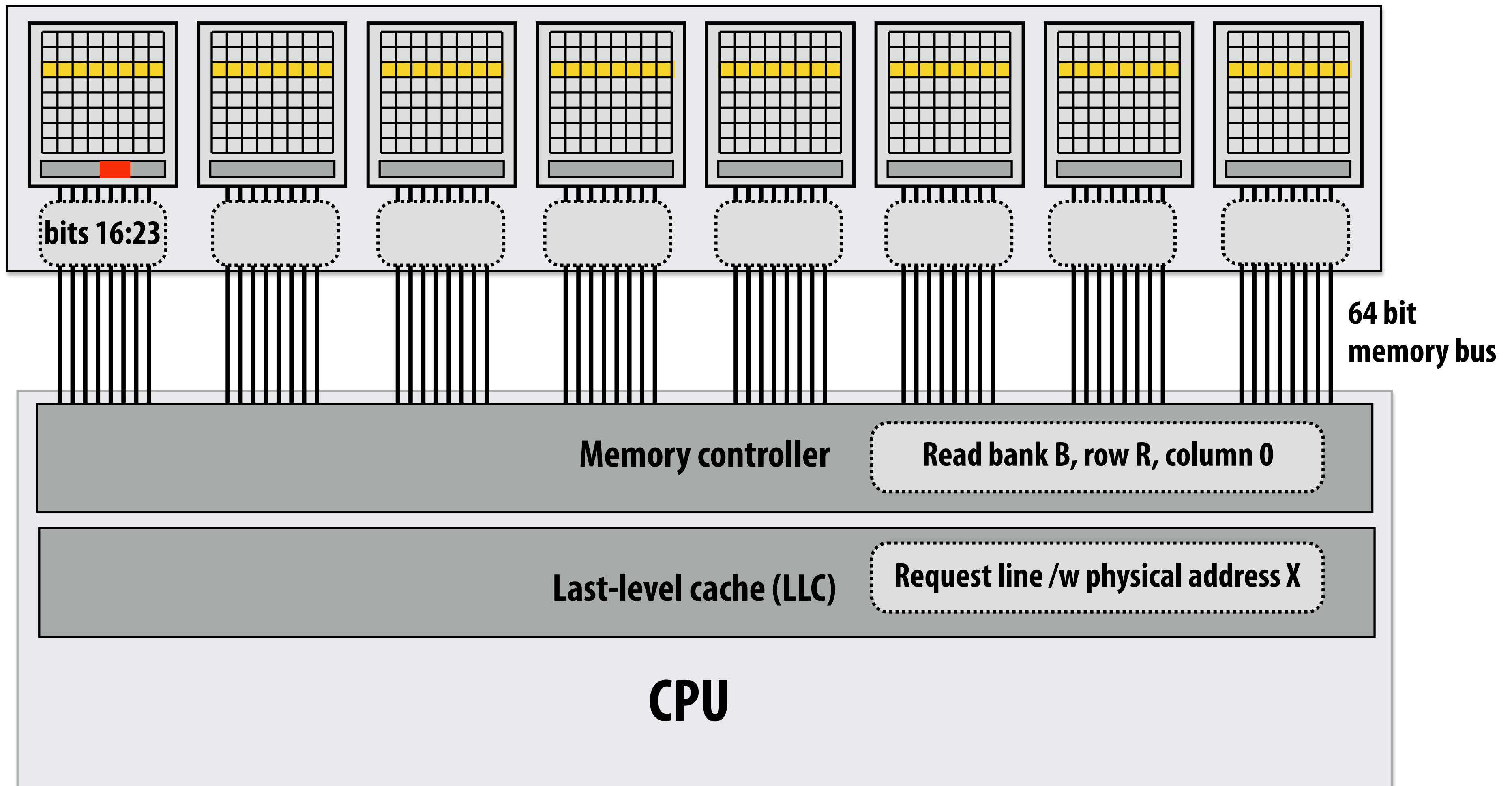
# Reading one 64-byte (512 bit) cache line (the wrong way)

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

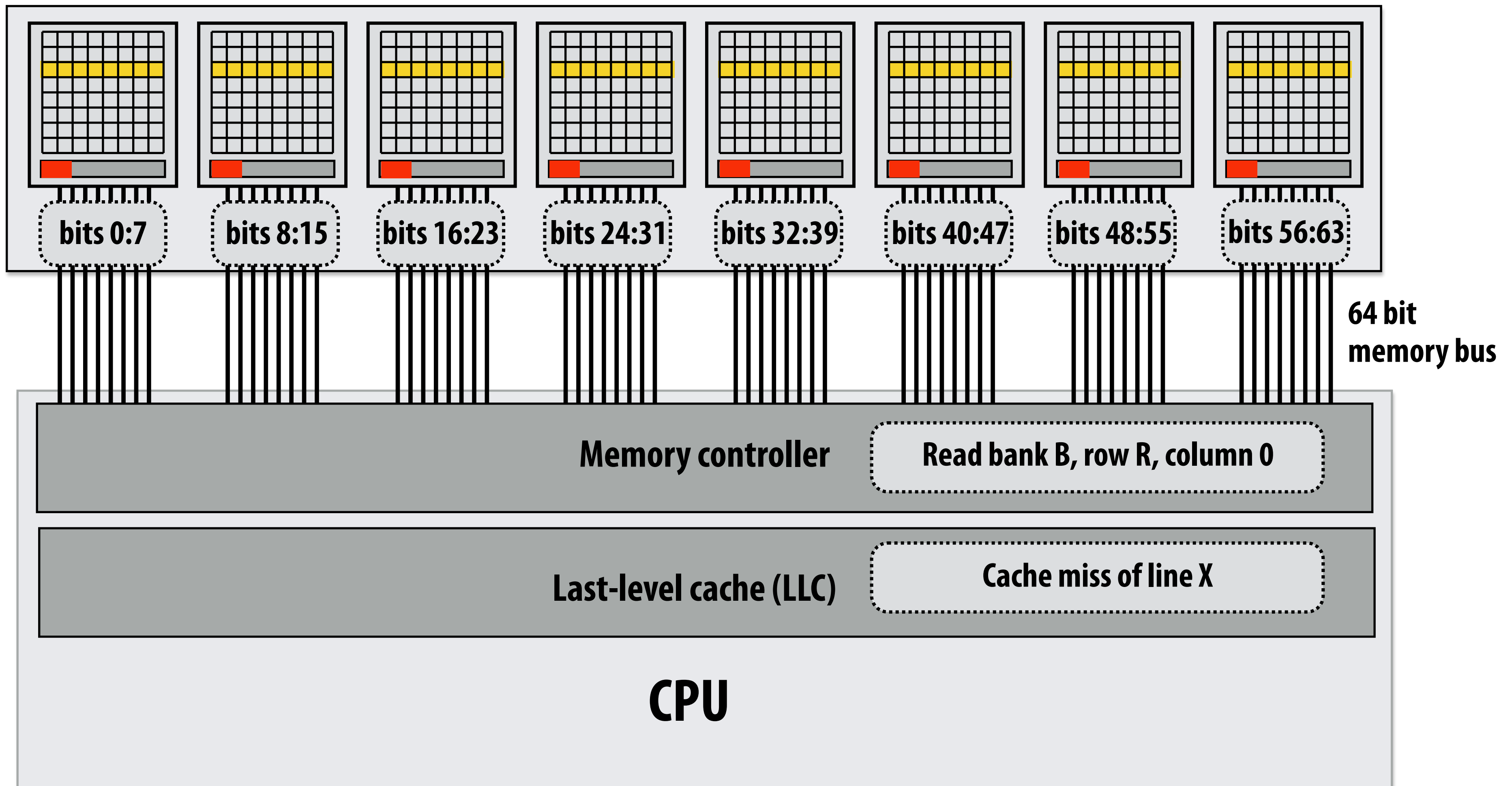


# Reading one 64-byte (512 bit) cache line

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

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

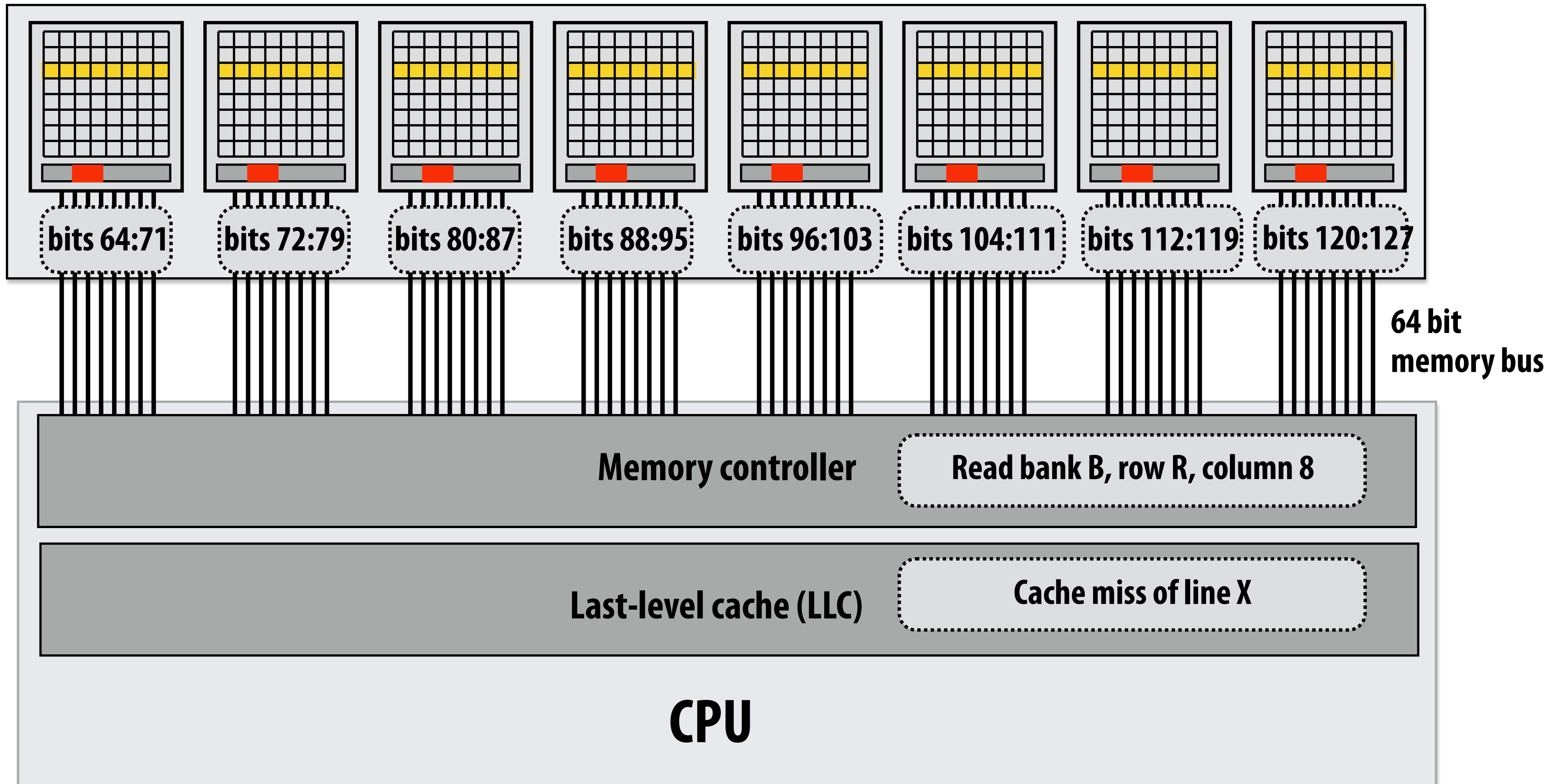
DRAM chips transmit first 64 bits in parallel



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

DRAM controller requests data from new column \*

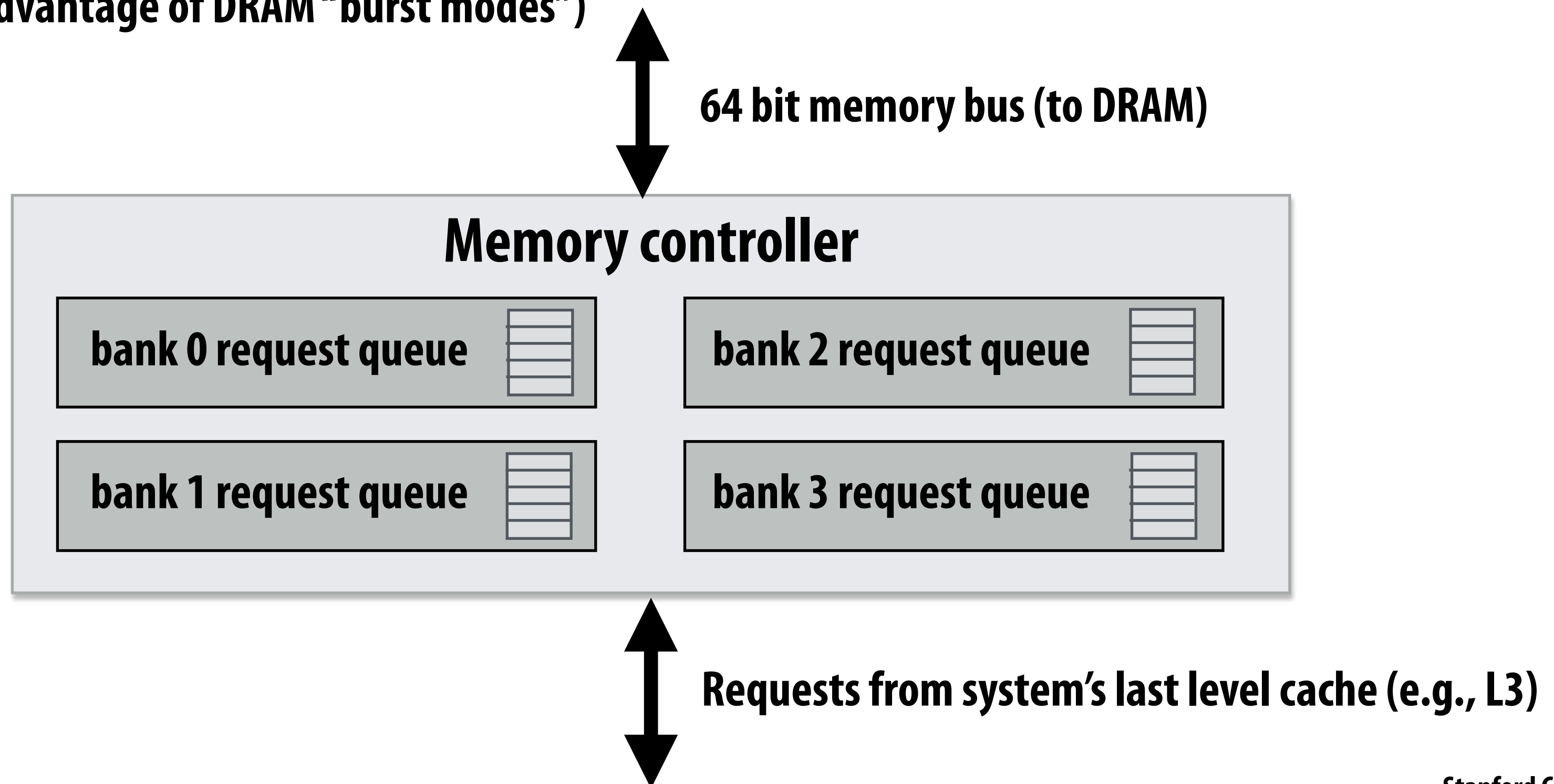
DRAM chips transmit next 64 bits in parallel



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

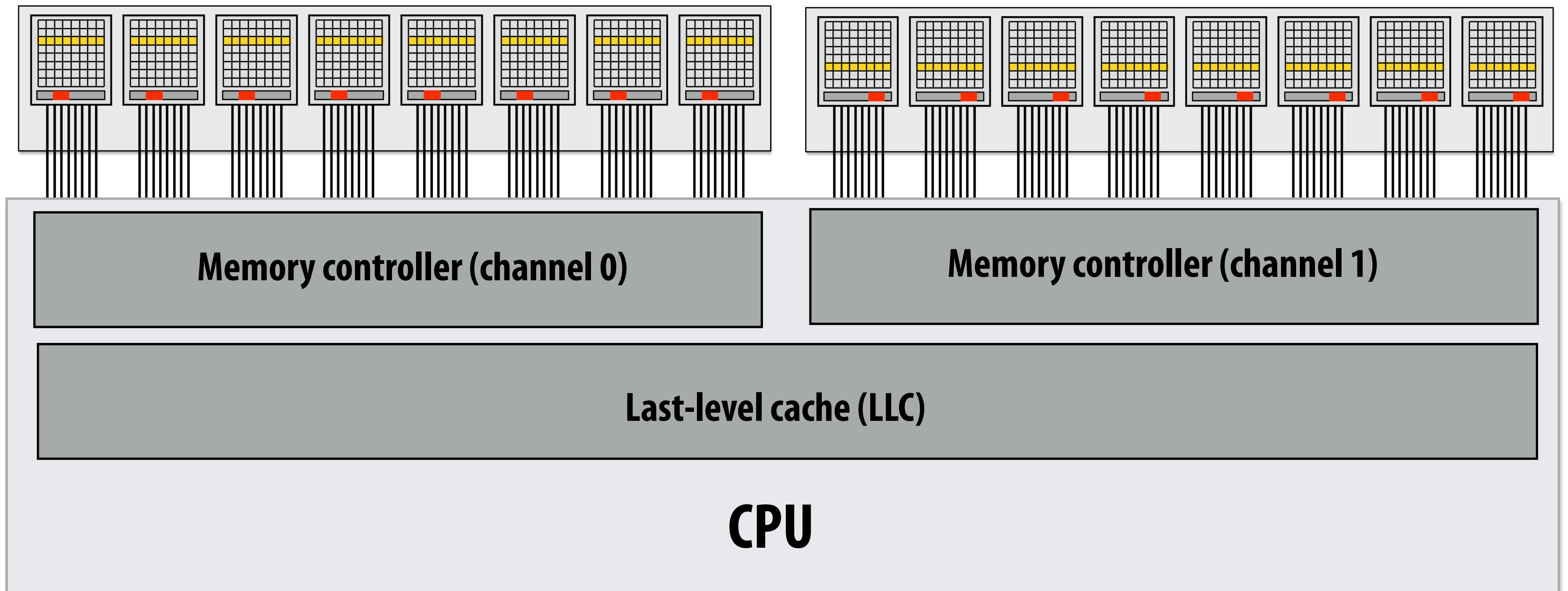
# Memory controller is a memory request scheduler

- **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

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

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

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

## 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

## DDR4 2400

- **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**

**\* DDR stands for “double data rate”**

# 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:**

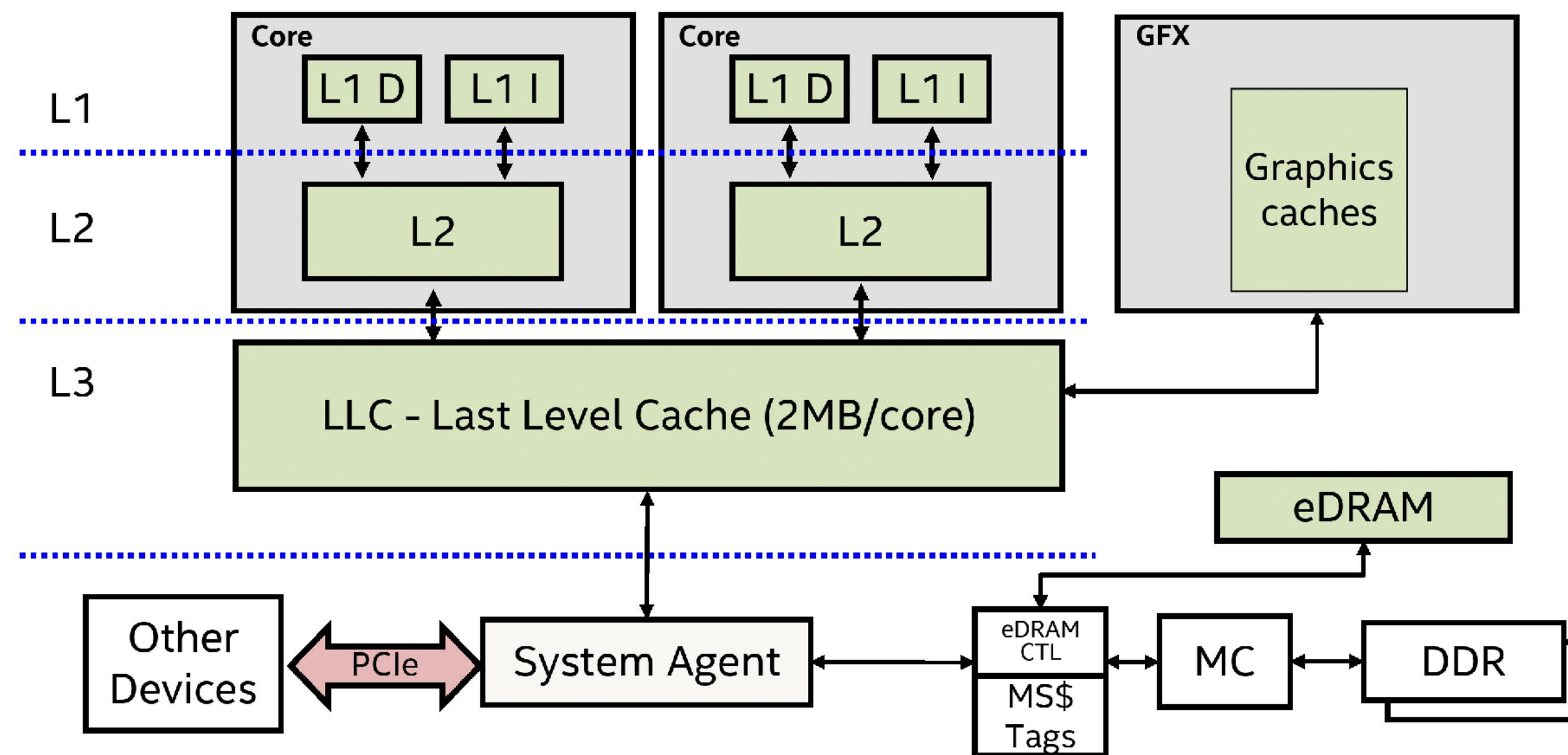
**Decrease distance data must move by  
locating memory closer to processors**

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

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

Some Intel Broadwell/Skylake processors featured 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

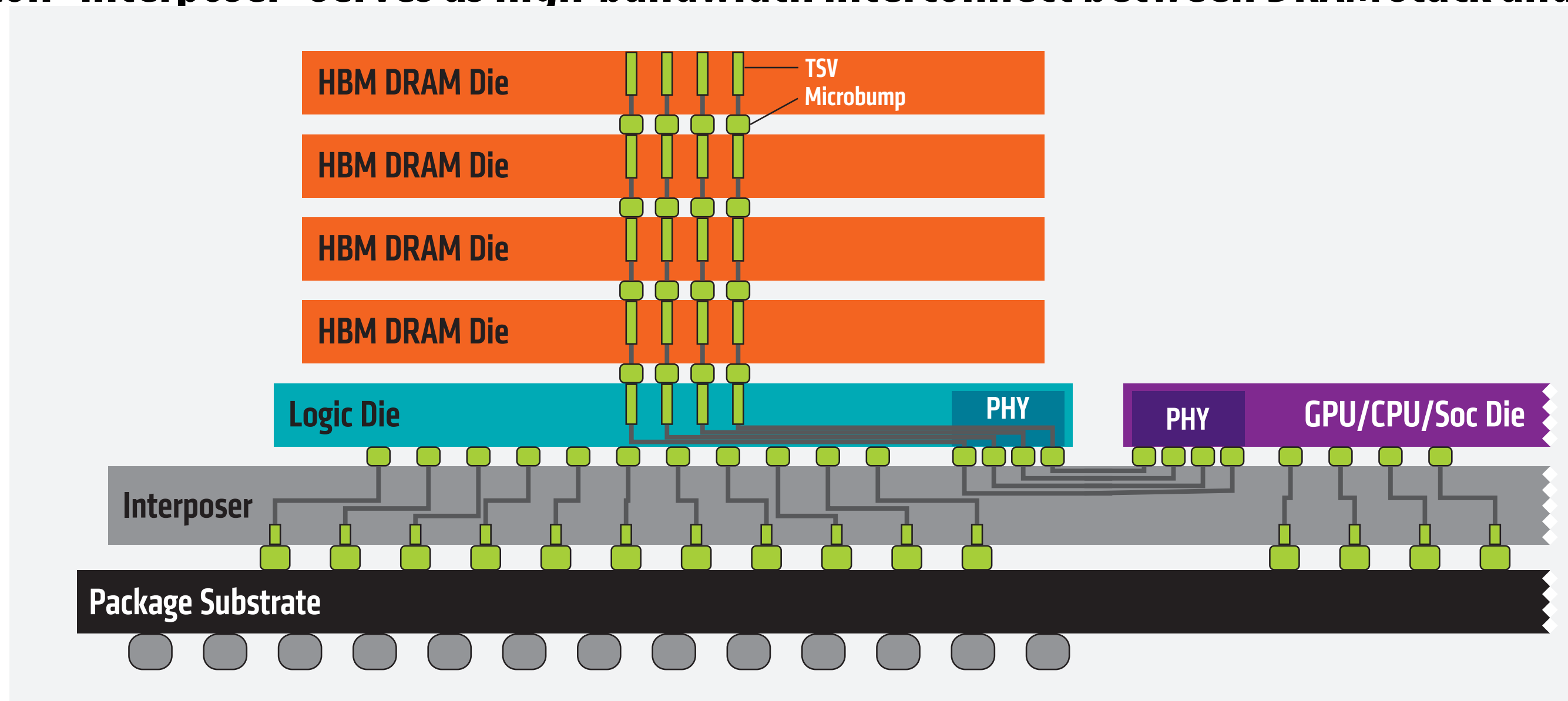
Attractive in mobile SoC setting



# 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

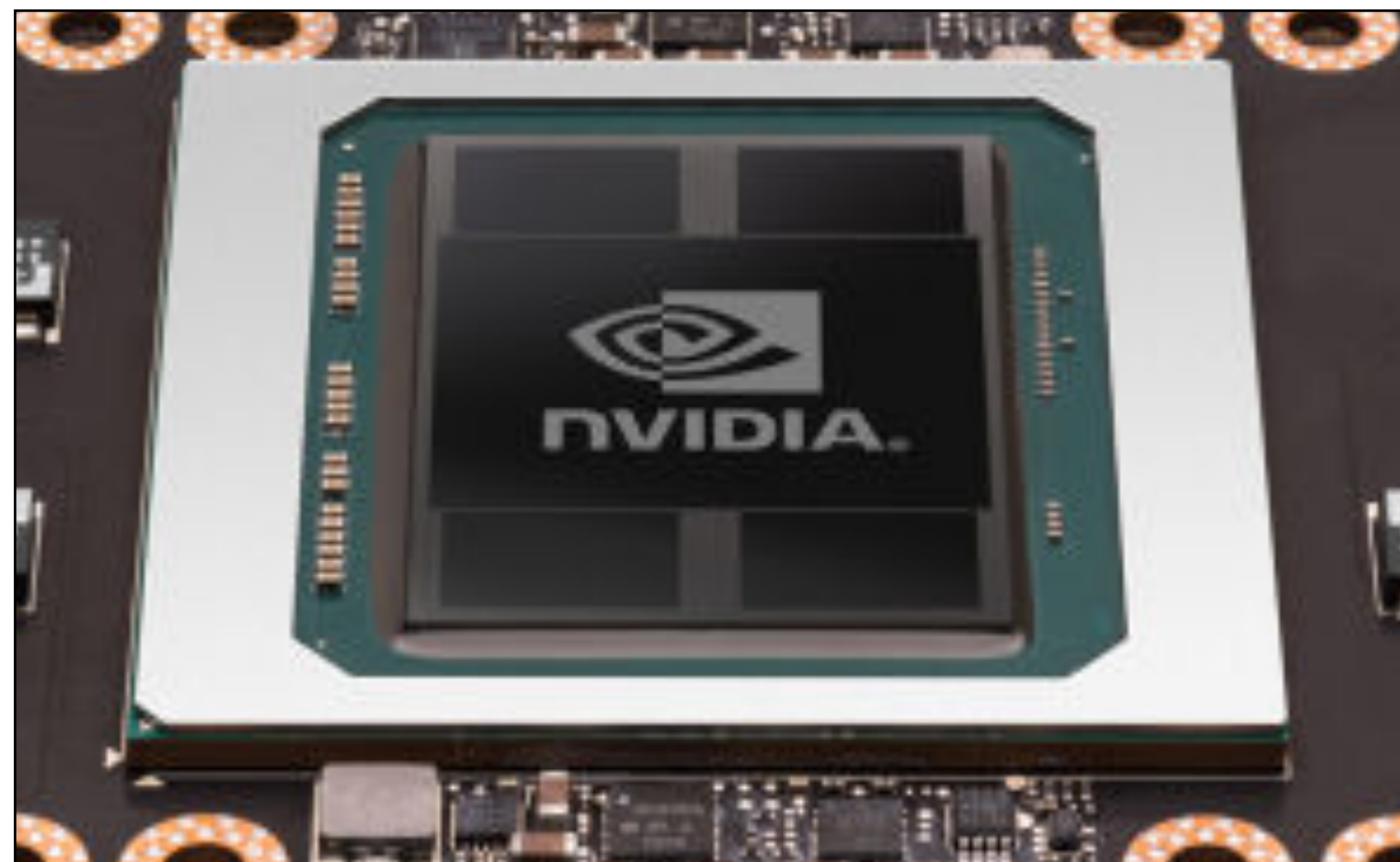
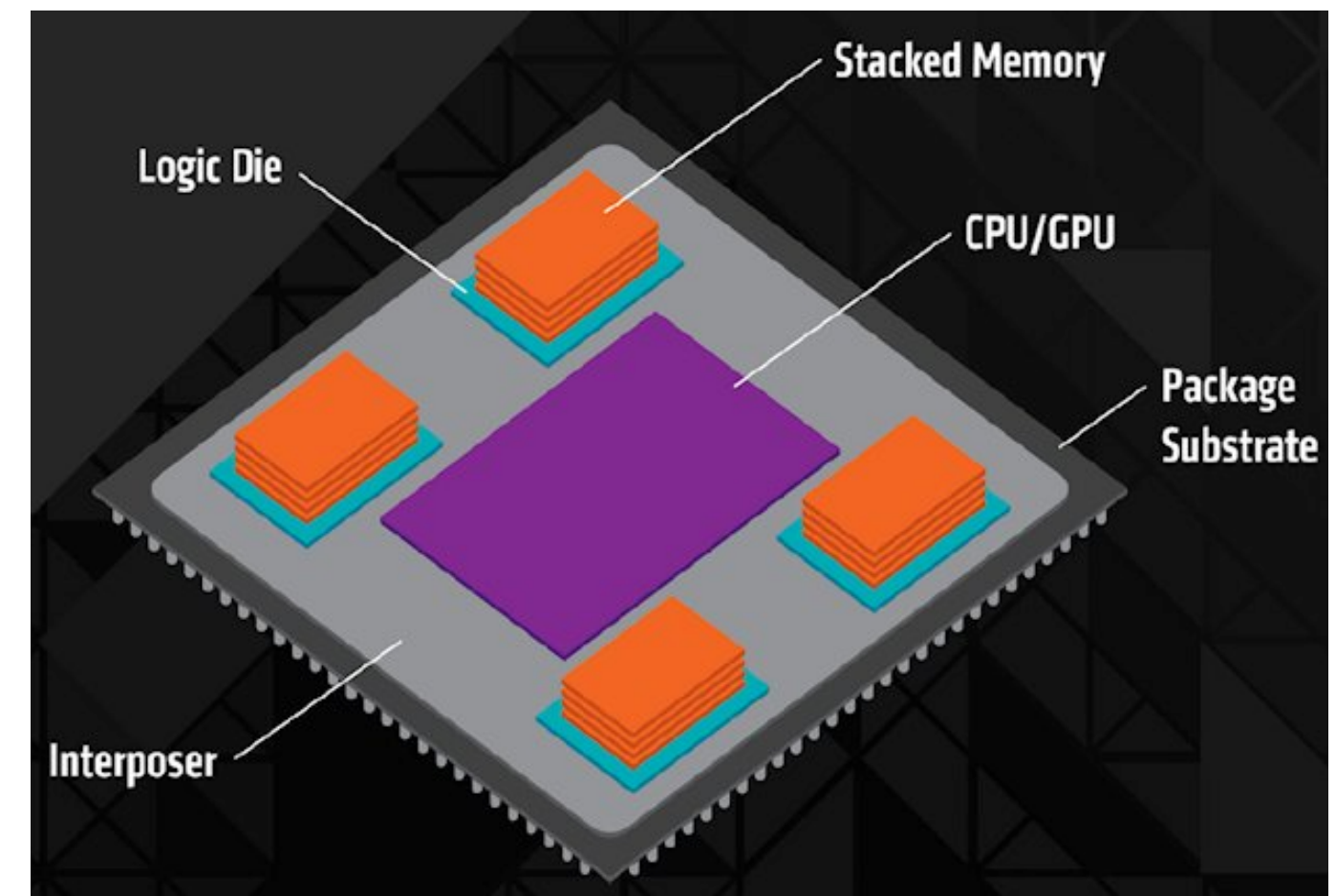
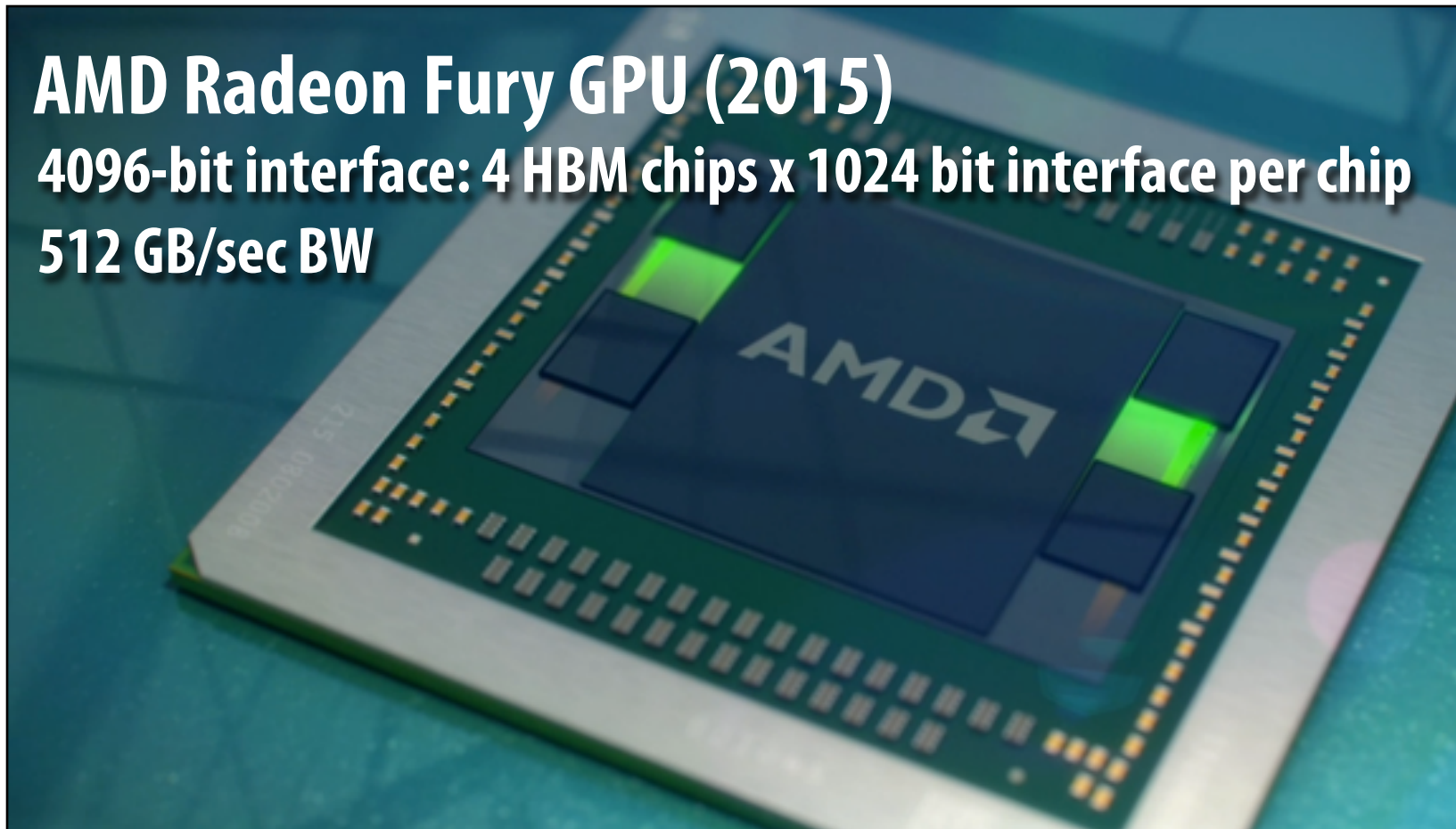


# GPUs are adopting HBM technologies

## AMD Radeon Fury GPU (2015)

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

512 GB/sec BW



## 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



# 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, 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 (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)