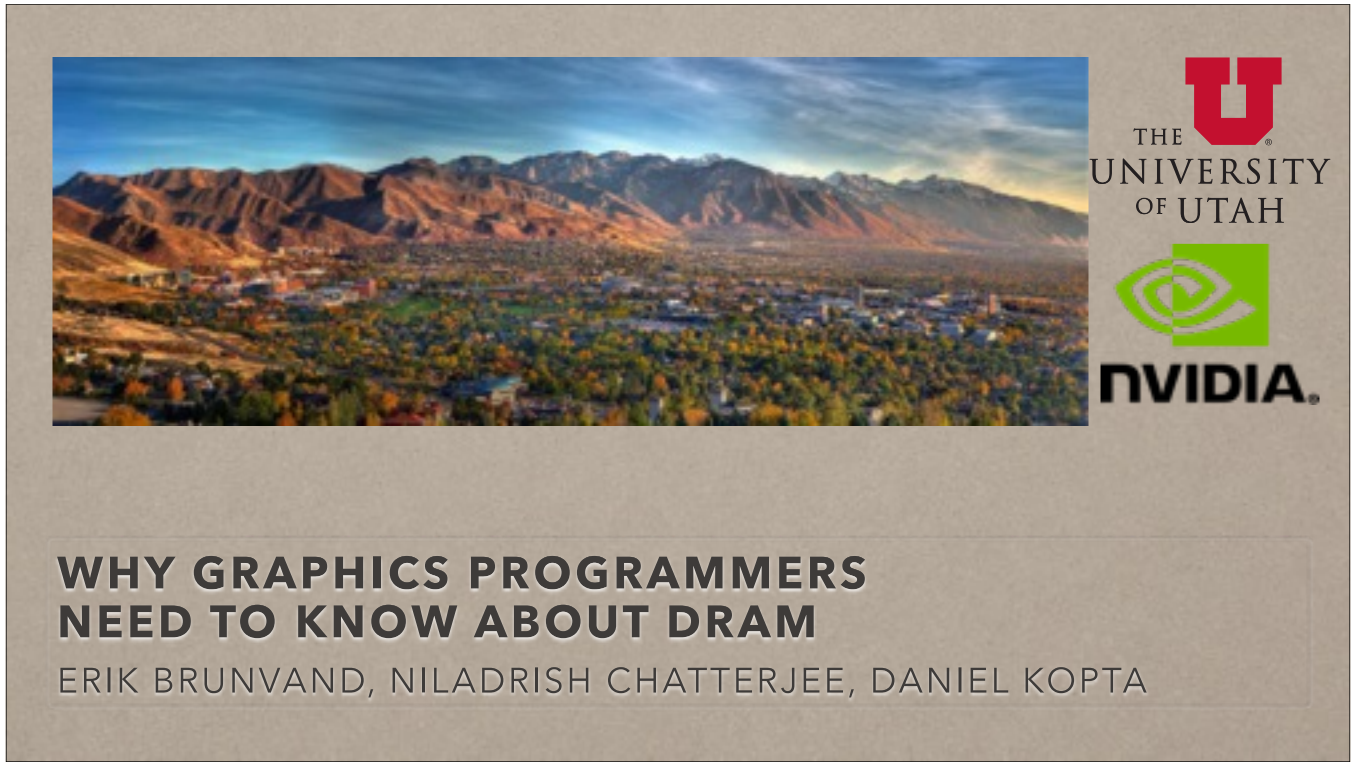


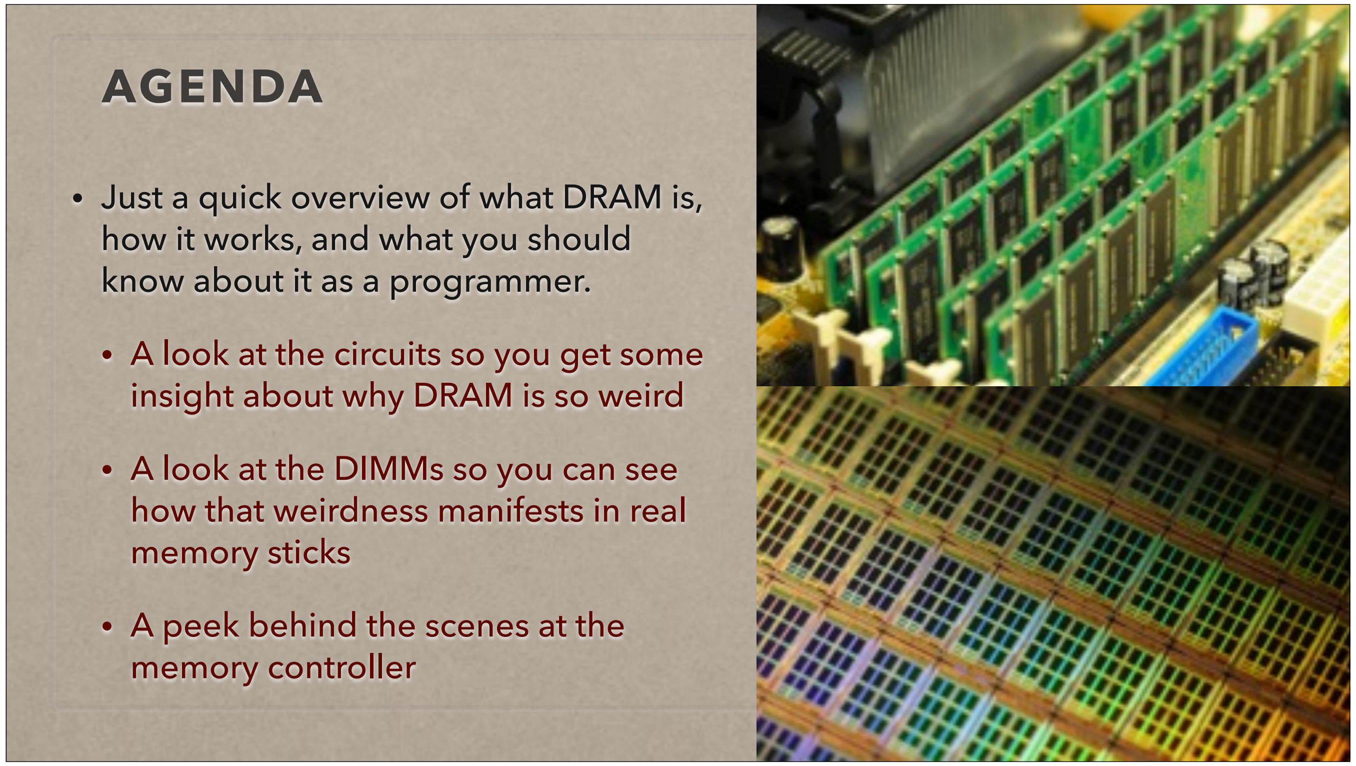
THE UNIVERSITY OF UTAH

NVIDIA

WHY GRAPHICS PROGRAMMERS NEED TO KNOW ABOUT DRAM

ERIK BRUNVAND, NILADRISH CHATTERJEE, DANIEL KOPTA

Welcome!



AGENDA

- Just a quick overview of what DRAM is, how it works, and what you should know about it as a programmer.
  - A look at the circuits so you get some insight about why DRAM is so weird
  - A look at the DIMMs so you can see how that weirdness manifests in real memory sticks
  - A peek behind the scenes at the memory controller

Welcome! This course is all about main memory behavior - DRAM in particular.

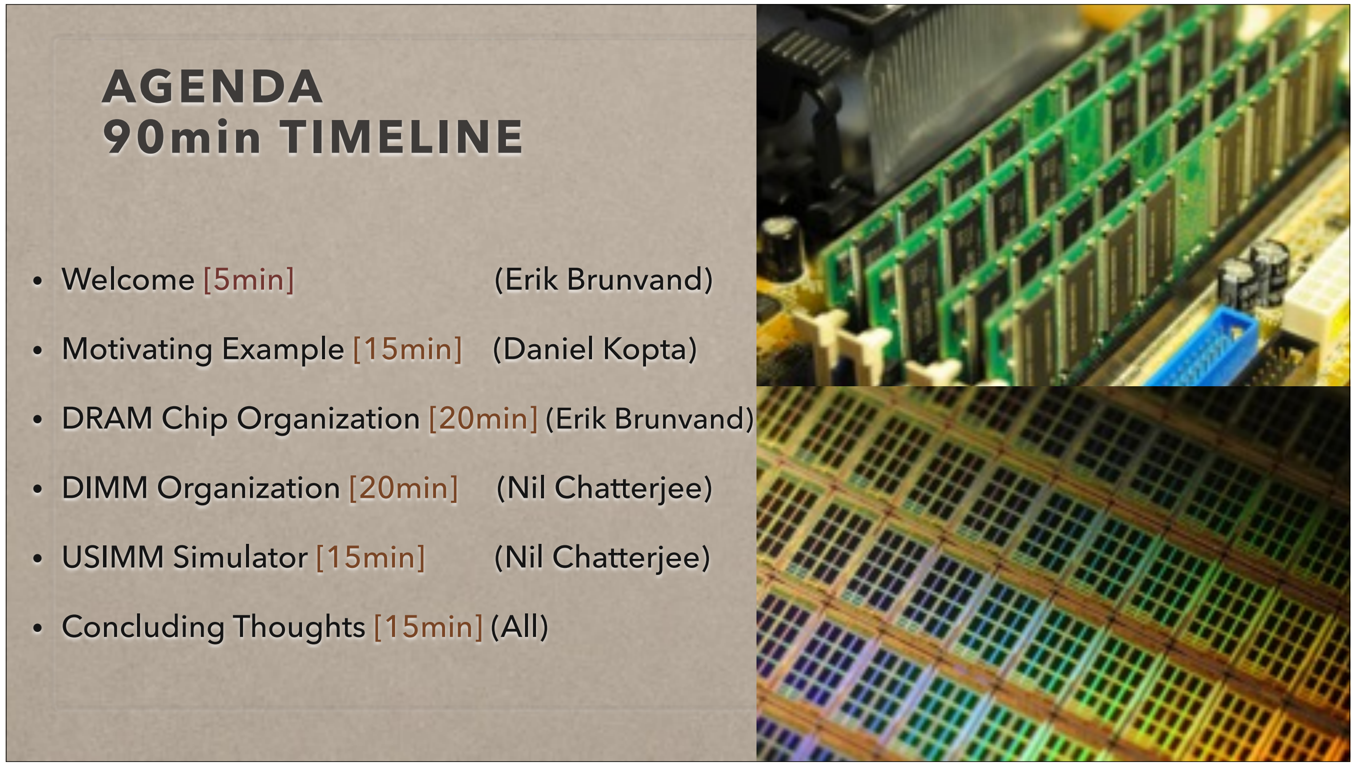
You may not think that main memory performance is important - it's caches that are really important, right?

Of course caches are important. But, even if your cache hit rates are high, and you are more concerned about throughput than latency, main memory is still interesting.

- For one thing, DRAM uses much more power than you probably think.

- It's also slower than you might think. So, small changes in the required bandwidth can have a large impact on your application's speed and power performance.

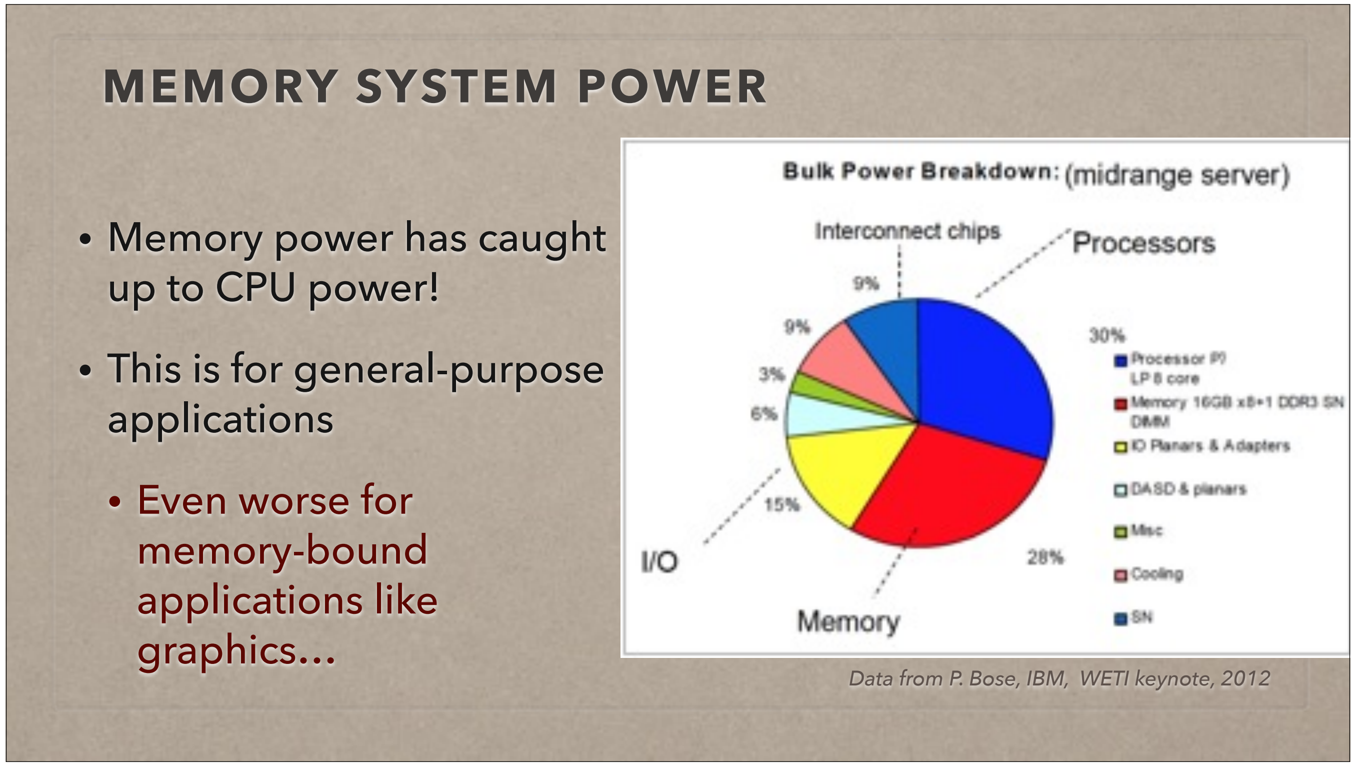
It turns out that DRAM is really complex to access, so if you don't have at least a little understanding of what's happening "under the hood", you won't know how to work with it.



AGENDA
90min TIMELINE

- Welcome [5min] (Erik Brunvand)
- Motivating Example [15min] (Daniel Kopta)
- DRAM Chip Organization [20min] (Erik Brunvand)
- DIMM Organization [20min] (Nil Chatterjee)
- USIMM Simulator [15min] (Nil Chatterjee)
- Concluding Thoughts [15min] (All)

- Course outline:
  - Erik Brunvand will start (Associate Professor, University of Utah) with some high-level discussion of why DRAM and main memory are interesting things for a programmer to consider.
  - Daniel Kopta, PhD student at the University of Utah, will introduce a motivating example that shows how a proposed parallel hardware architecture designed for Ray Tracing reacts to DRAM. Even for a system with high cache hit rates, changes in how DRAM is used can have a large impact on system performance.
  - Brunvand will describe some details of DRAM circuit and chip behavior. This sets the stage for much of the interesting behavior that we'll talk about later. It all comes from the circuit structures required by DRAM
  - Nil Chatterjee, from NVIDIA Architecture Research will then describe how those DRAM chips are assembled into higher-level structures (DIMMs, Ranks, Banks, etc.) He will also describe how the Memory Controller on a CPU and on a GPU communicate/negotiate with that main memory hierarchy.
  - Chatterjee will also talk about a highly accurate main memory simulator: USIMM. This simulator provides an accurate model of main memory timing and power, taking DIMM, rank, bank, and chips into account.
  - We will conclude with some wrap-up thoughts about how programmers can or can't take the influence of DRAM behavior into account in their programs.



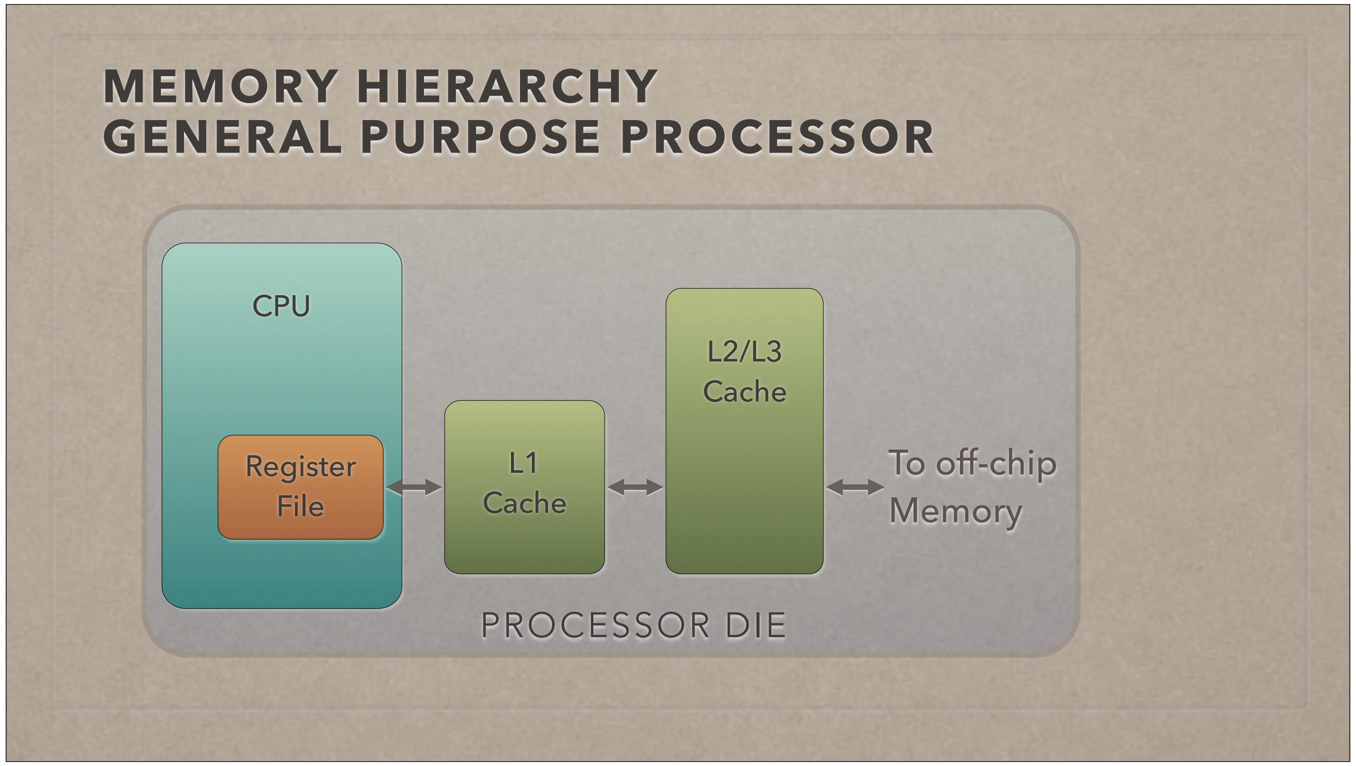
MEMORY SYSTEM POWER

- Memory power has caught up to CPU power!
- This is for general-purpose applications
  - Even worse for memory-bound applications like graphics…

Data from P. Bose, IBM, WETI keynote, 2012

One big motivator - memory power has caught up with processor power!

- And, power is a huge issue these days.
- Many studies have shown that DRAM consumes 30-50% of the total power in a datacenter machine. That’s huge!
- This even worse for memory-bound applications like graphics. We’ll see an example shortly.



MEMORY HIERARCHY
GENERAL PURPOSE PROCESSOR

CPU

Register File

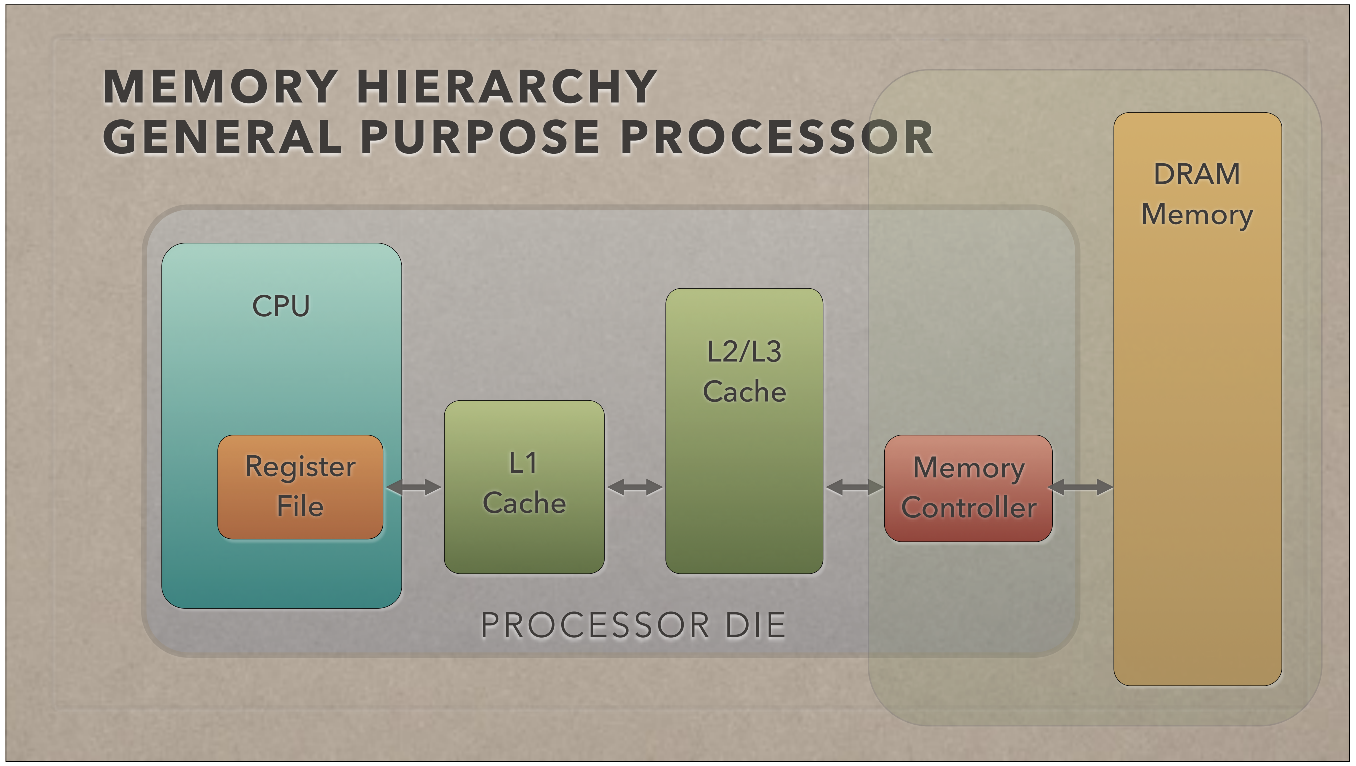
L1 Cache

L2/L3 Cache

To off-chip Memory

PROCESSOR DIE

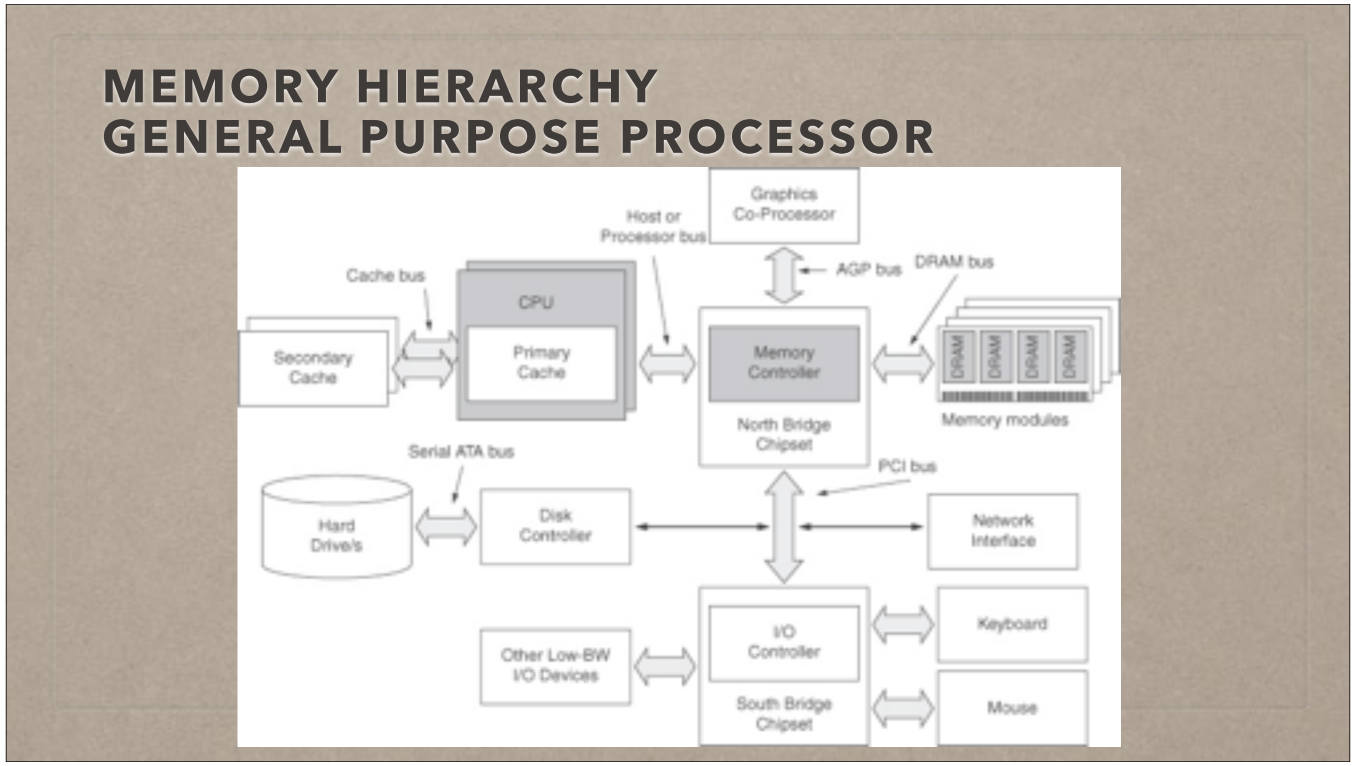
This is a general view of a generic on-chip memory hierarchy. If possible, you want to keep all required data within this portion of the hierarchy! But, we’re not always so lucky.



MEMORY HIERARCHY
GENERAL PURPOSE PROCESSOR

This is the focus of this course - off-chip main memory (DRAM), and the on-chip memory controller that deals with it.

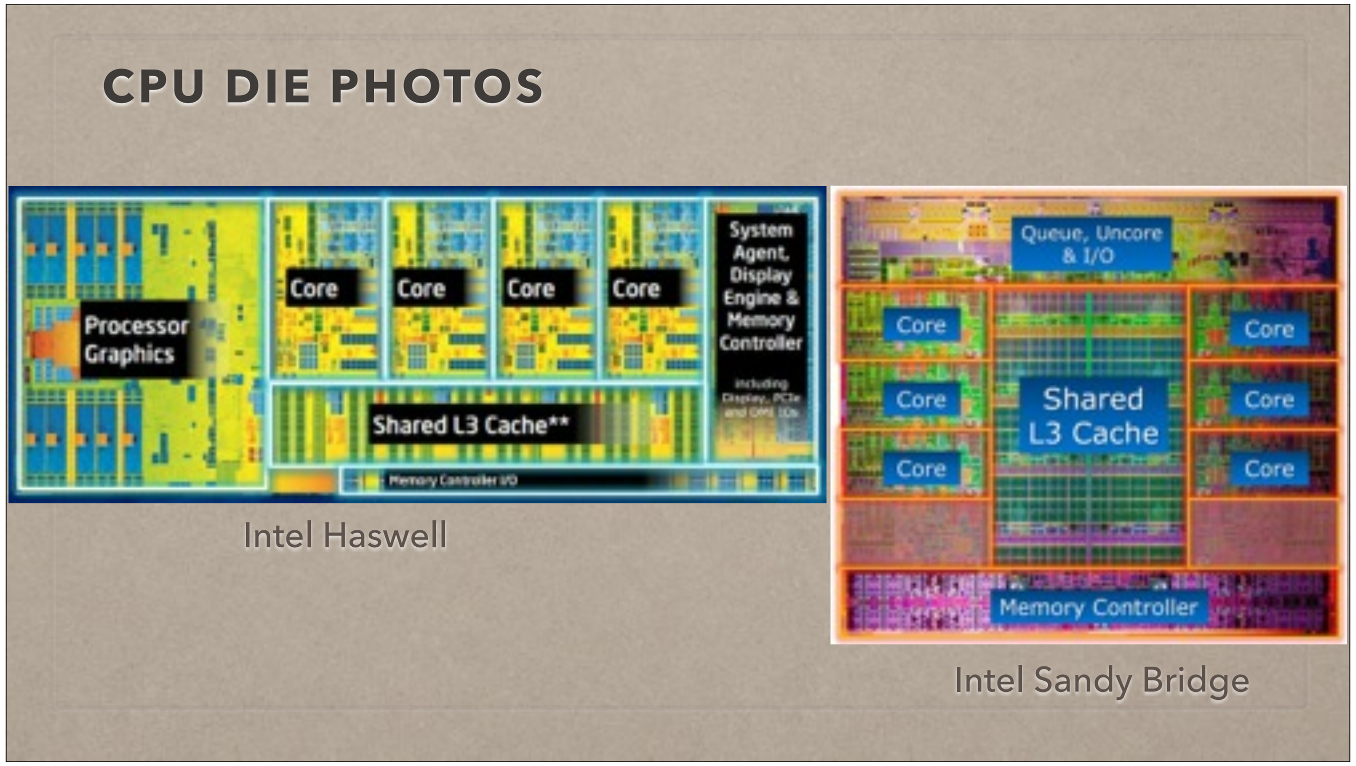
Note that this doesn’t have anything to do with the virtual memory system. We’re talking about how to access main memory once the physical address is known. Turns out that even once you know the physical address, getting that data from the DRAM is not trivial!



MEMORY HIERARCHY
GENERAL PURPOSE PROCESSOR

This is the focus of this course - off-chip main memory (DRAM), and the on-chip memory controller that deals with it.

Note that this doesn't have anything to do with the virtual memory system. We're talking about how to access main memory once the physical address is known. Turns out that even once you know the physical address, getting that data from the DRAM is not trivial!

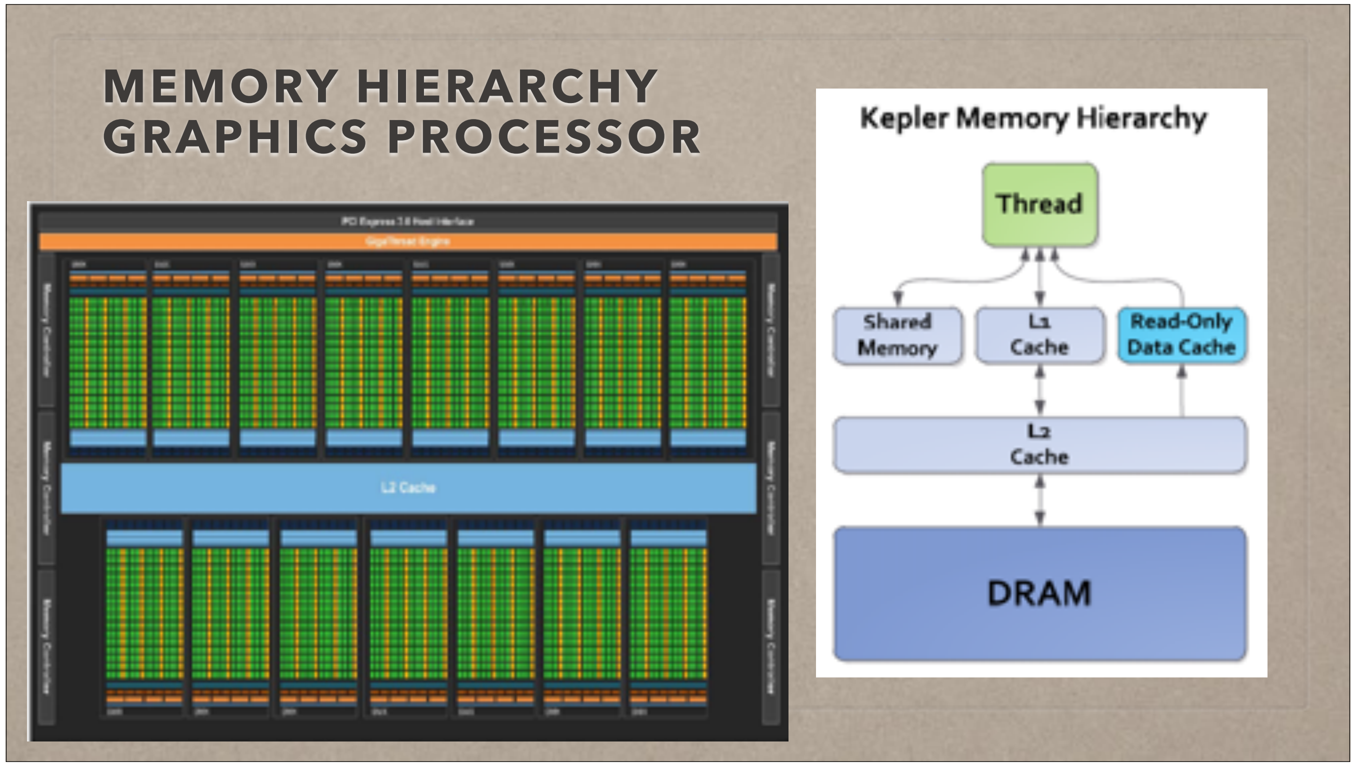


CPU DIE PHOTOS

Intel Haswell

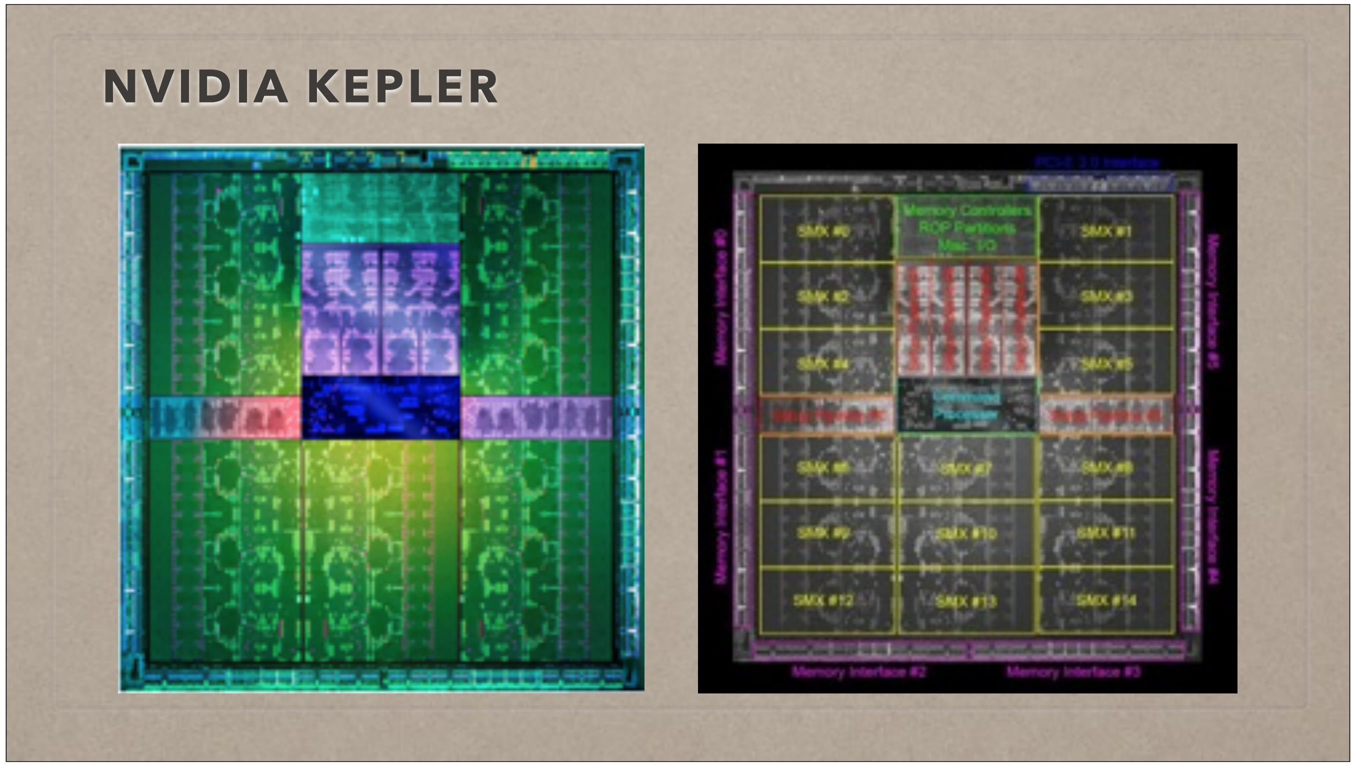
Intel Sandy Bridge

Some quick chip photos - the important thing here is that the memory controller is not a small thing. It’s a complex beast that includes a number of queues for incoming and outgoing data, and some complex state machines that deal with the complexities of the DRAM interface, the DRAM internal structure, and things like refresh.



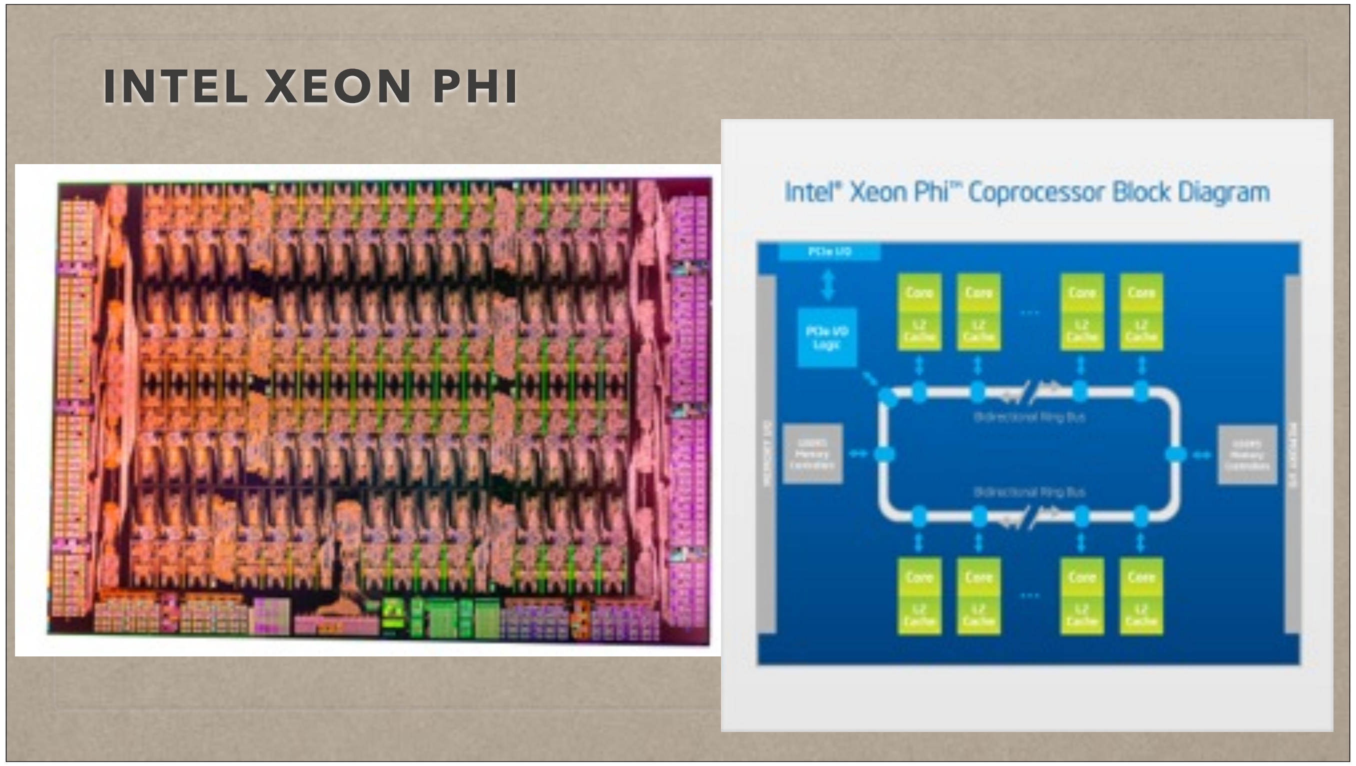
MEMORY HIERARCHY
GRAPHICS PROCESSOR

GPUs aren't immune to this - modern high-end GPUs have a very similar memory hierarchy. It's a little more specialized for the behavior of graphics data, but it still has to deal with those complex DRAM beasts.



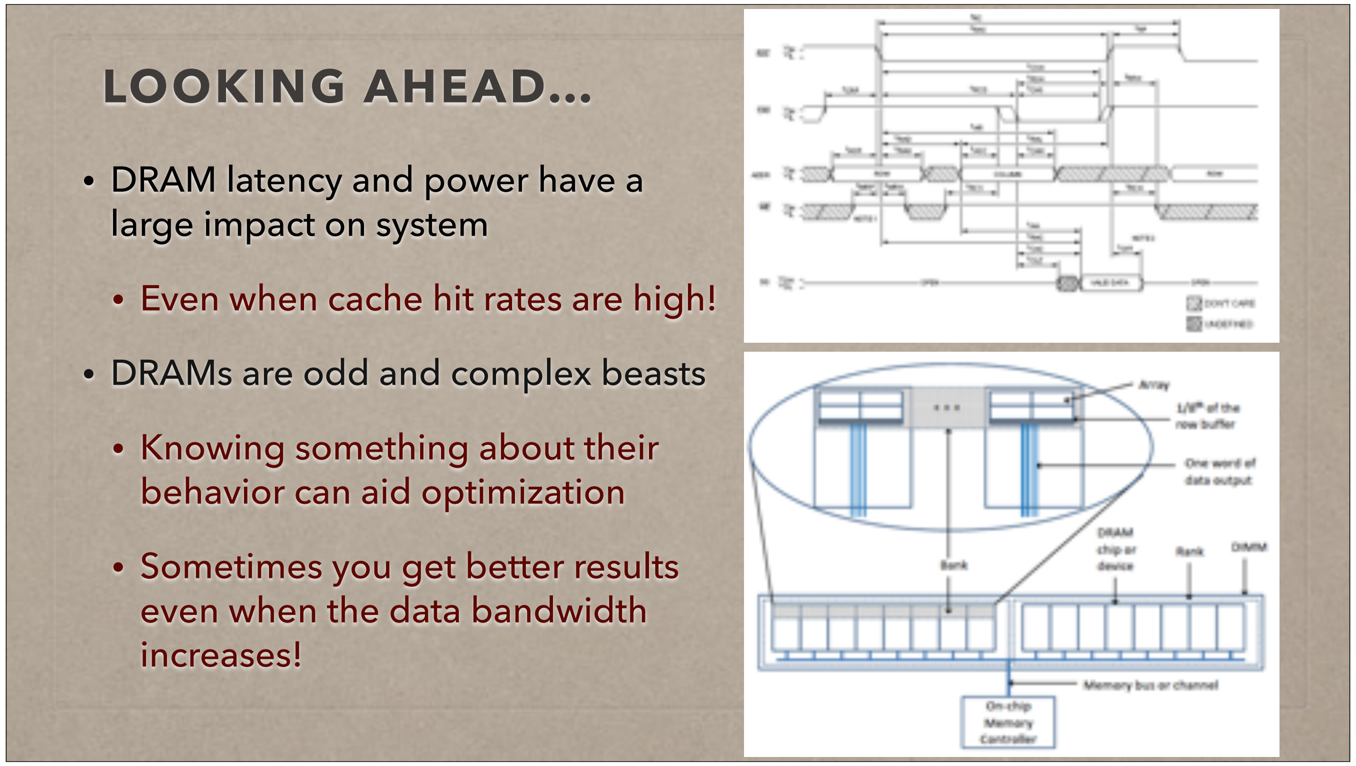
NVIDIA KEPLER

The NVIDA Kepler die shows where the memory controllers and physical memory interfaces live on that die.



INTEL XEON PHI

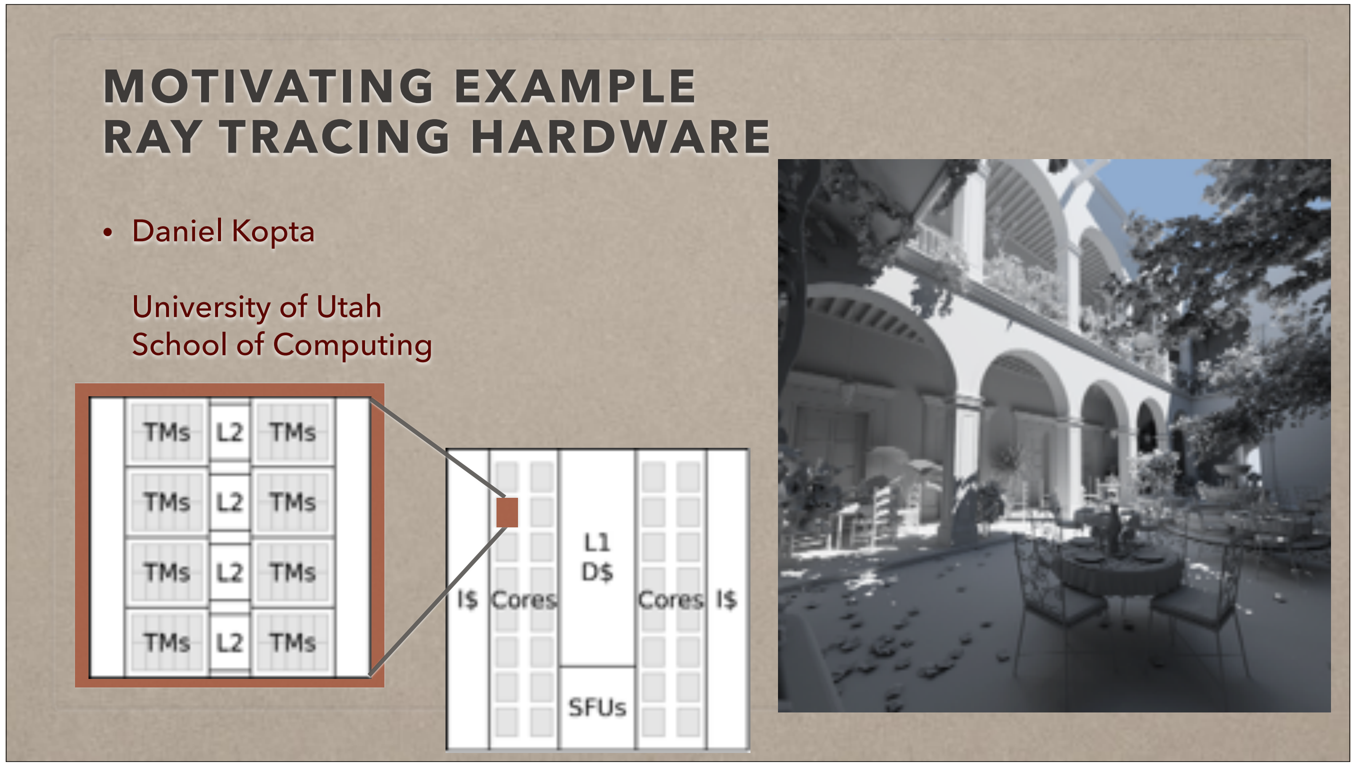
Xeon Phi also has significant memory controllers instantiated as “first class citizens” on their high-speed ring interconnect.



LOOKING AHEAD…

- DRAM latency and power have a large impact on system
  - Even when cache hit rates are high!
- DRAMs are odd and complex beasts
  - Knowing something about their behavior can aid optimization
  - Sometimes you get better results even when the data bandwidth increases!

Looking ahead - we'll look at the hidden extra hierarchy in a DRAM main memory system, and especially the hidden cache known as the "Row Buffer." Hopefully you'll at least learn something about how complex DRAM and main memory subsystems are. You should also learn something about how you can modify your applications to be more efficient in their dealing with main memory. This can have a large impact on performance!



MOTIVATING EXAMPLE
RAY TRACING HARDWARE

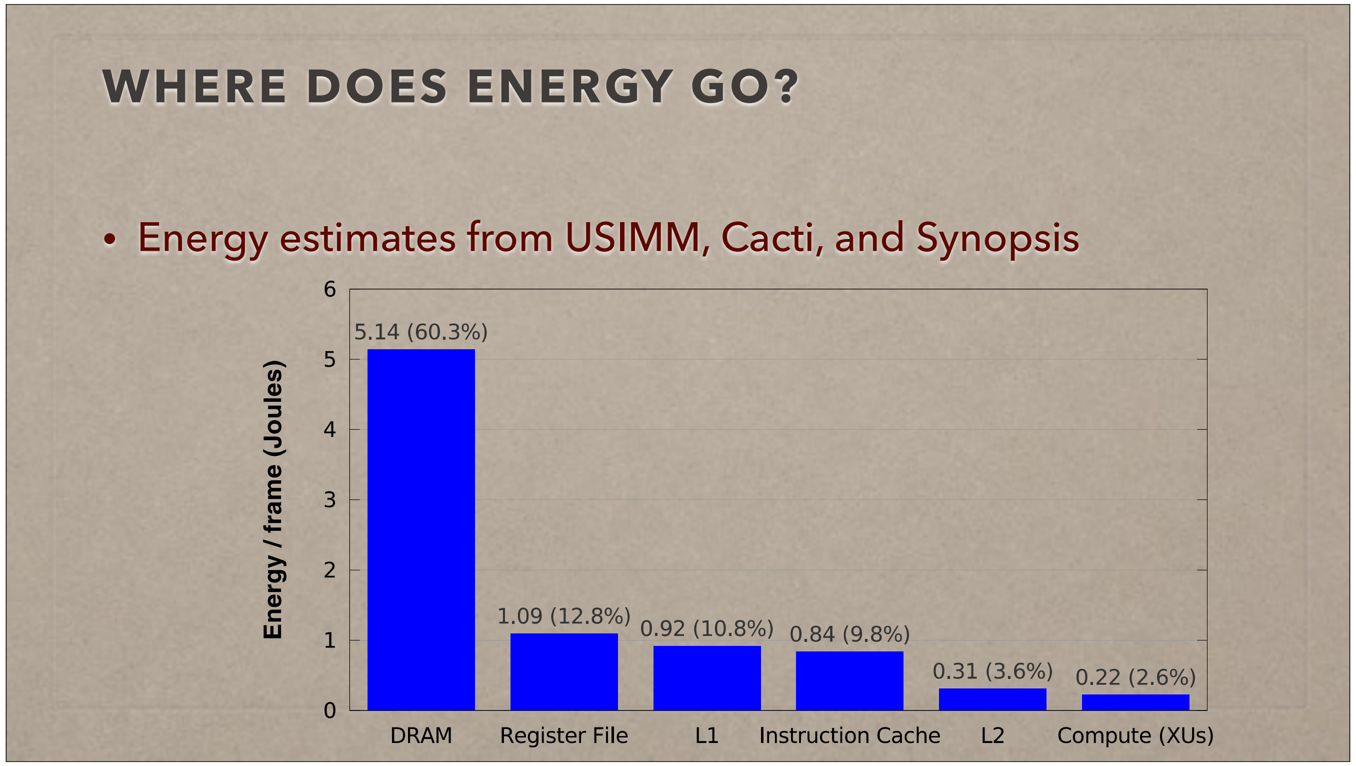
- Daniel Kopta

  University of Utah
  School of Computing

As a motivating example, we'll use a study of a proposed parallel system designed for ray tracing. This system is based on our TRaX (Threaded Ray eXecution) simulated architecture from the University of Utah. While this example is ray tracing specific, the lessons learned should apply to a wide range of bandwidth oriented graphics apps

The underlying architecture is a tiled parallel architecture with lots of independent thread processors - not tied together in SIMD clusters.

- The most basic unit is a Thread Processor (TP) - a simple in-order integer processor with its own RF, integer functional units, and PC.
- Multiple TPs are clustered into a Thread Multiprocessor (TM). Inside a TM, the TPs share multi-banked I$ and D$, and also larger functional units such as FPUs, integer multiply, and FP inv/sqrt.
- Multiple TMs are tiled on the chip where groups of TMs share multi-banked L2 caches.
- The L2 caches have external DRAM interfaces for cache refill.

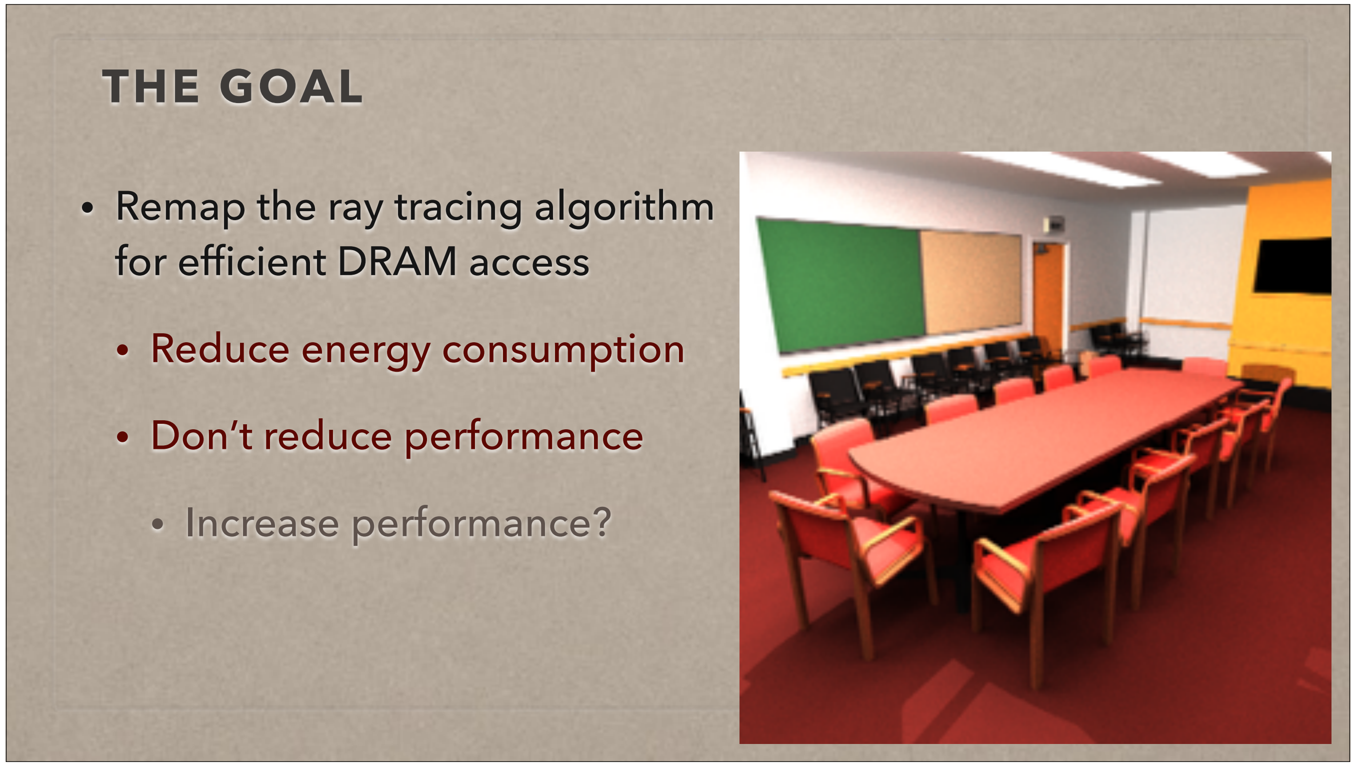


WHERE DOES ENERGY GO?

- Energy estimates from USIMM, Cacti, and Synopsis

Looking at the energy performance of the baseline system running a set of path tracing benchmarks - here's where the energy is being used. Clearly the memory is the primary issue with energy usage!

- Off-chip DRAM is the largest component: ~60% of energy used
- On-chip memory (RF, L1-D$, I$, and L2) takes another ~30%

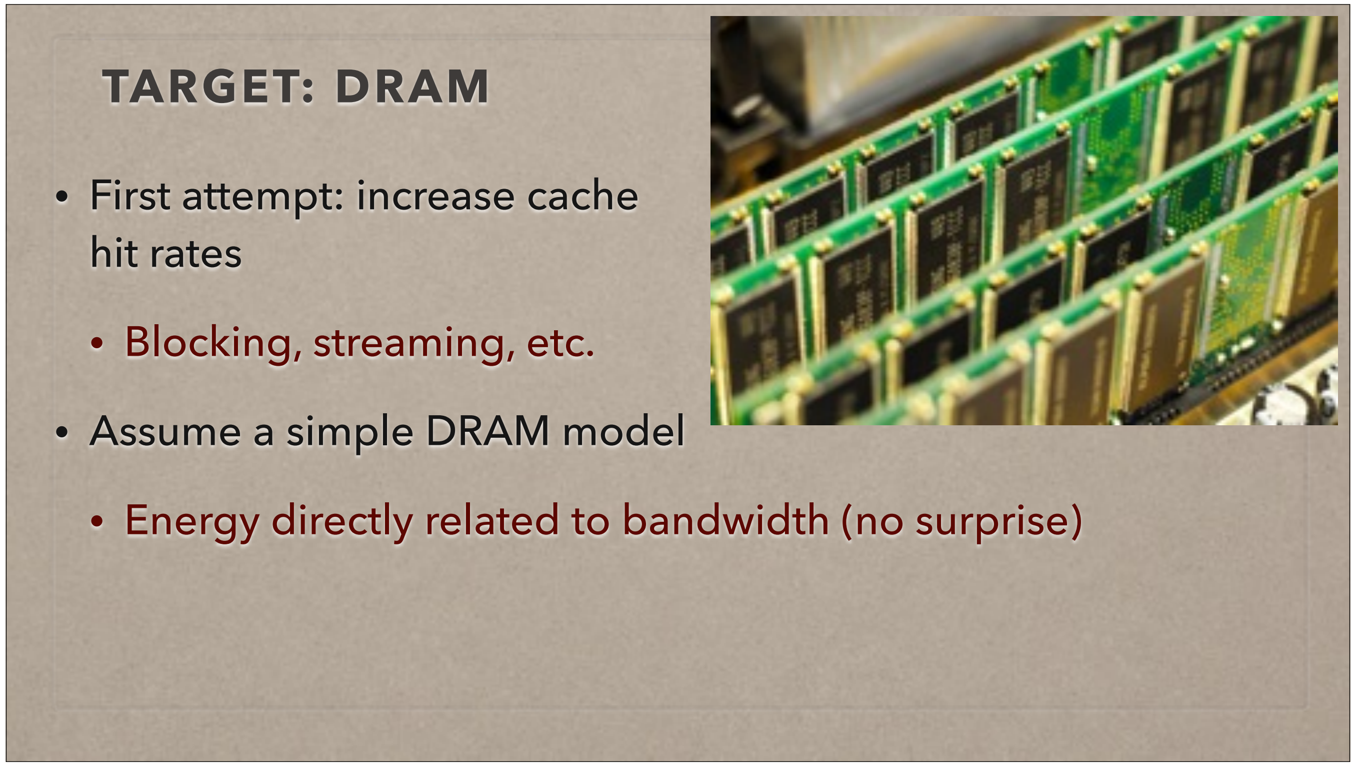


THE GOAL

- Remap the ray tracing algorithm for efficient DRAM access
  - Reduce energy consumption
  - Don’t reduce performance
    - Increase performance?

So - the question is - what can we do about that?

How can we reduce energy consumption without impacting FPS?



TARGET: DRAM

- First attempt: increase cache hit rates
  - Blocking, streaming, etc.
- Assume a simple DRAM model
  - Energy directly related to bandwidth (no surprise)

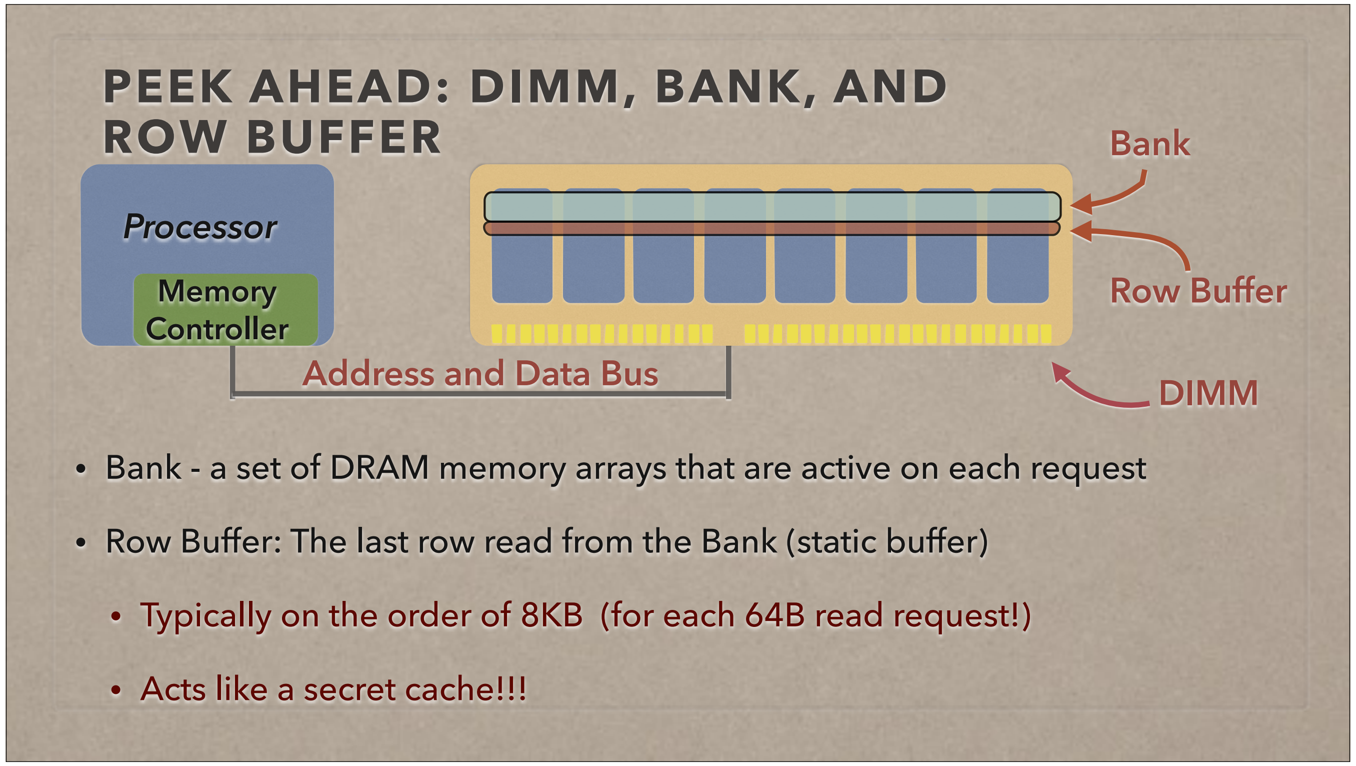
To reduce DRAM energy consumption, the obvious thing to try is to reduce the number of accesses to it (bandwidth consumed). This will certainly be true if we work under the simple assumption that DRAM energy consumption is directly related to bandwidth. Aside from reducing the amount of data required by a program (perhaps through data compression), another option is to improve cache hit rates. While this does not reduce the total data requirements, it does reduce the amount of it that must come from DRAM.

There are many techniques in ray tracing that either directly aim to increase cache hit rates, or achieve this as a side effect [Gribble et al. 2008, Ramani et al. 2009, Aila et al. 2010, Bigler et al. 2006, Boulos et al. 2007, Günther et al. 2007, Overbeck et al. 2008].

References:
[Gribble et al. 2008] GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In Symposium on Interactive Ray Tracing (IRT08).
[Ramani et al. 2009] RAMANI, K., AND GRIBBLE, C. 2009. StreamRay: A stream filtering architecture for coherent ray tracing. In ASPLOS '09.
[Aila et al. 2010] AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In Proc. High Performance Graphics.
[Bigler et al. 2006] BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In Symposium on Interactive Ray Tracing (IRT06).
[Boulos et al. 2007] BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based Whitted and Distribution Ray Tracing. In Proc. Graphics Inter- face.
[Günther et al. 2007] GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In Symposium on Interactive Ray Tracing (IRT07), 113–118.
[Overbeck et al. 2008] OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time whitted ray tracing. In Symposium on Interactive Ray Tracing (IRT08), 41–48.

TARGET: DRAM

- First attempt: increase cache hit rates
  - Blocking, streaming, etc.
- Now add a detailed DRAM model
  - Energy can decrease even when DRAM bandwidth goes up!
  - Clearly there's something interesting going on here…

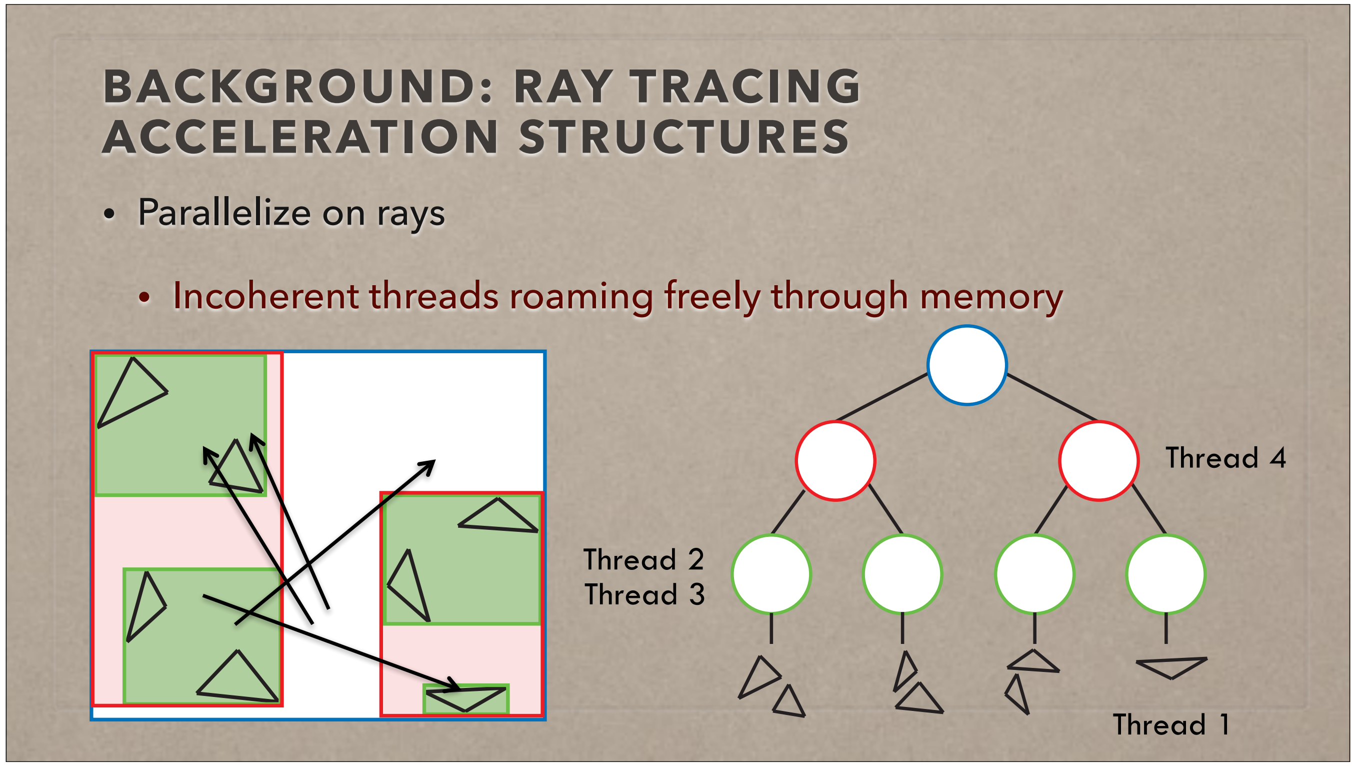


PEEK AHEAD: DIMM, BANK, AND ROW BUFFER

- Bank - a set of DRAM memory arrays that are active on each request
- Row Buffer: The last row read from the Bank (static buffer)
  - Typically on the order of 8KB (for each 64B read request!)
  - Acts like a secret cache!!!

As a peek ahead we should look at perhaps the most important feature of off chip DRAM from an energy perspective - the Row Buffer.

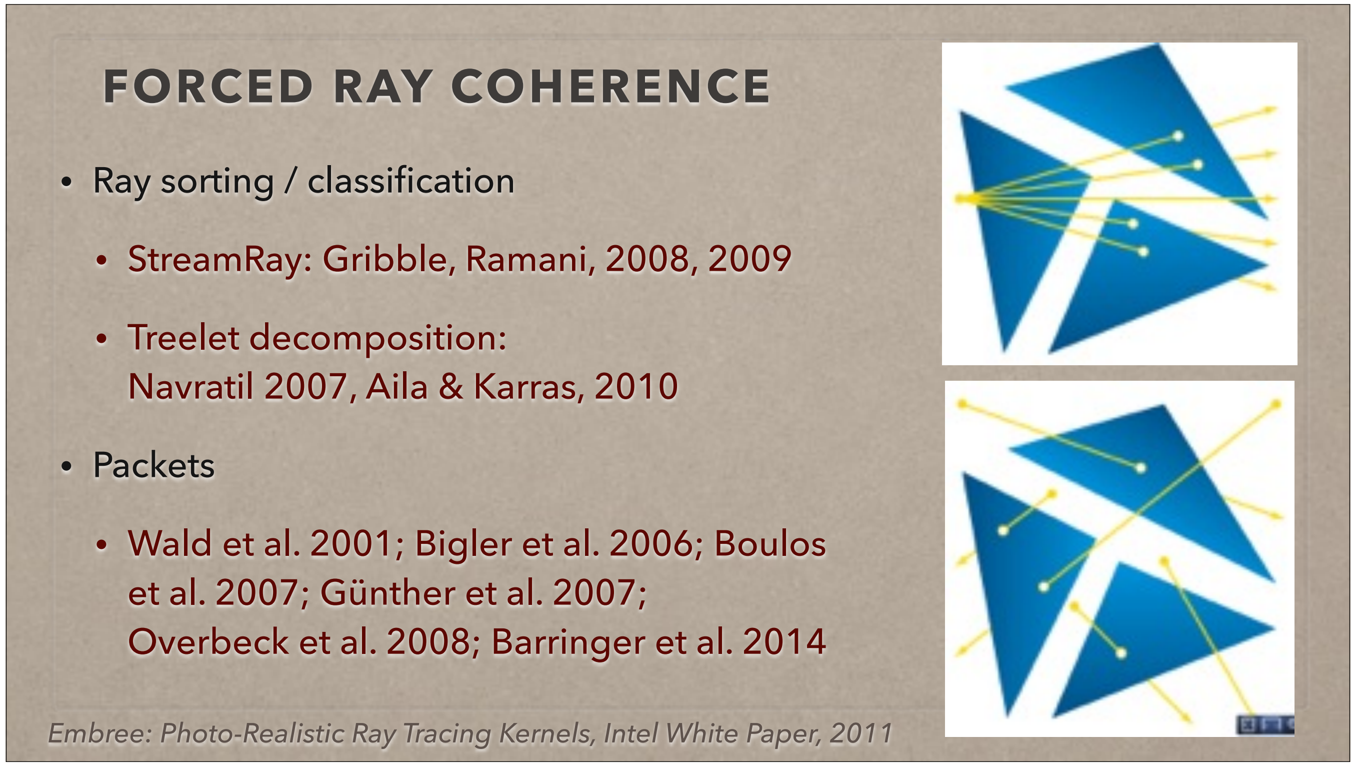
- When off-chip DRAM memory is accessed on a DIMM, one “Bank” is energized. A Bank is the set on-chip memory arrays (mats) that are activated to service any single memory request.
- When a bank is accessed, an entire row of the bank is read (basically a row from every memory array (mat)).
- Note that the Row Buffer which is activated in each active Mat, logically spans all the chips that are used in each Bank.
- This leads to fairly dramatic over-fetch behavior
  - For each 64B access request, something like 8KB of data is fetched into the Row Buffer (the set of Row Buffers in each Mat).
  - This large Row Buffer acts like a ”secret cache!”
  - But, it’s not exposed to the user like a cache is.
- Once you know about it, though, you can think about ways to exploit it and get better behavior!
- This is the crux of this class - knowing something about DRAM behavior (such as the Row Buffer behavior) can lead to thinking about optimization in a whole new way.



BACKGROUND: RAY TRACING ACCELERATION STRUCTURES

- Parallelize on rays
  - Incoherent threads roaming freely through memory

The fundamental operation of ray tracing is determining which (if any) triangle a ray hits. In order to avoid testing every ray against every triangle, ray tracing systems use an "acceleration structure" of some form in order to prune the possible ray-triangle intersections that may occur. Bounding volume hierarchies (BVH)s are a very popular choice. A large portion of the ray tracing computation involves traversing this tree data structure in an unpredictable pattern. These data structures can be very large, depending on the number of triangles in the scene.



FORCED RAY COHERENCE

- Ray sorting / classification
  - StreamRay: Gribble, Ramani, 2008, 2009
  - Treelet decomposition: Navratil 2007, Aila & Karras, 2010
- Packets
  - Wald et al. 2001; Bigler et al. 2006; Boulos et al. 2007; Günther et al. 2007; Overbeck et al. 2008; Barringer et al. 2014

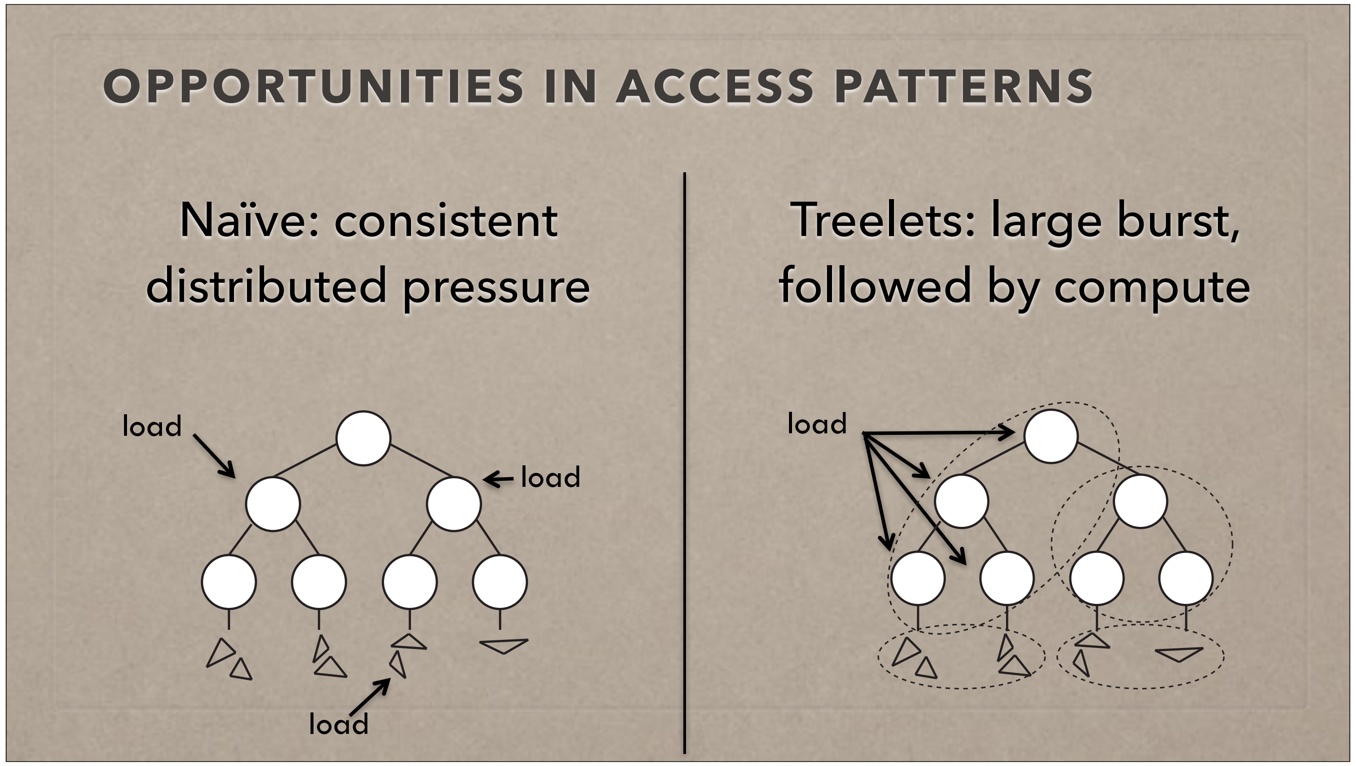
Embree: Photo-Realistic Ray Tracing Kernels, Intel White Paper, 2011

Discovering coherent ray bundles is a heavily researched area. Gribble and Ramani [Gribble et al. 2008, Ramani et al. 2009] use filter operations to group rays in to bins based on their characteristics such as what type of ray they are (shadow, primary, etc…), what they hit (bounding volume, triangle, nothing), and what material they need if they hit geometry. The end goal is to increase SIMD utilization by processing only rays running the same compute kernels.

[Aila et al. 2010] subdivide the BVH tree in to "treelets" designed to fit either in the L1 or L2 data cache. Each treelet has an associated pool of rays that are currently traversing that portion of the BVH. Groups of threads corresponding to a single L1 cache, in their case an NVIDIA SM, picks one treelet and associated ray pool to work on at a time. If a ray crosses a treelet boundary during traversal, the ray is ushered to another ray pool to be processed either by another SM or at a later time. The result is that threads operate on cache-resident data for a prolonged period of time, increasing hit rates and reducing off-chip bandwidth. This technique should be applicable to general parallel architectures, and not just NVIDIA GPUs.

Other techniques collect groups of rays called packets that share a similar origin and direction, and are thus coherent and likely to traverse similar paths through the acceleration structure [Wald et al. 2001; Bigler et al. 2006, Boulos et al. 2007, Günther et al. 2007, Overbeck et al. 2008; Barringer et al. 2014]. These packets typically start as a collection of camera or shadow rays and break apart as individual rays within the packet make different branching decisions in the tree.

References:
[Gribble et al. 2008] GRIBBLE, C., AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In Symposium on Interactive Ray Tracing (IRT08).
[Ramani et al. 2009] RAMANI, K., AND GRIBBLE, C. 2009. StreamRay: A stream filtering architecture for coherent ray tracing. In ASPLOS '09.
[NFLM07] NAVRATIL P., FUSSELL D., LIN C., MARK W.: Dynamic ray scheduling for improved system performance. In Symposium on Interactive Ray Tracing (IRT07) (2007).
[Aila et al. 2010] AILA, T., AND KARRAS, T. 2010. Architecture considerations for tracing incoherent rays. In Proc. High Performance Graphics.
[Wald et al. 2001] WALD, I., SLUSALLEK, P., BENTHIN, C., WAGNER, M. 2006. Interactive Rendering With Coherent Ray Tracing. In Computer Graphics Forum (Proceedings of Eurographics).
[Bigler et al. 2006] BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In Symposium on Interactive Ray Tracing (IRT06).
[Boulos et al. 2007] BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based Whitted and Distribution Ray Tracing. In Proc. Graphics Interface.
[Günther et al. 2007] GÜNTHER, J., POPOV, S., SEIDEL, H.P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. In Symposium on Interactive Ray Tracing (IRT07), 113–118.
[Overbeck et al. 2008] OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time whitted ray tracing. In Symposium on Interactive Ray Tracing (IRT08), 41–48.
[Barringer et al. 2014] BARRINGER, R., AKENINE-MOLLER, T. 2014. Dynamic Ray Stream Traversal. In SIGGRAPH '14.

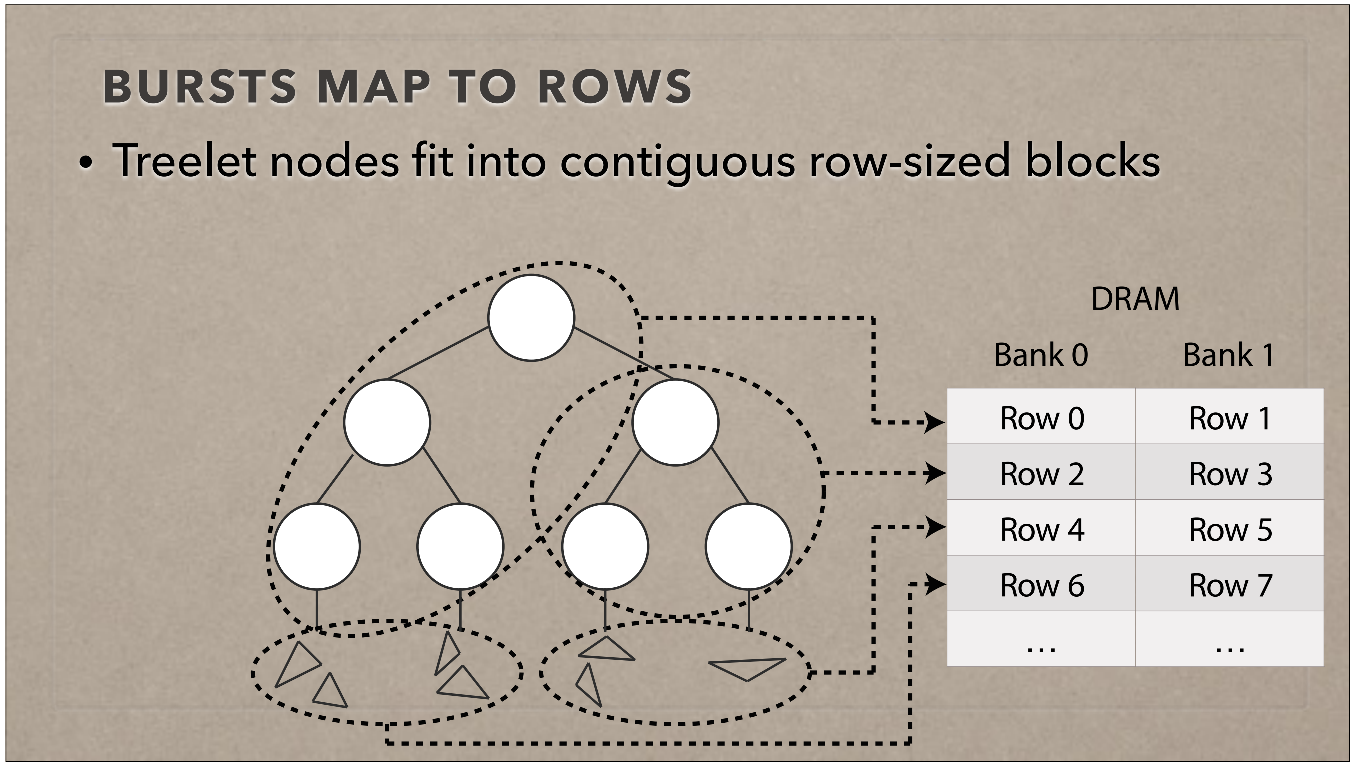


OPPORTUNITIES IN ACCESS PATTERNS

Naïve: consistent distributed pressure

Treelets: large burst, followed by compute

We note that treelets not only increase cache hit rates, as they are designed, but have an interesting side effect. Memory accesses that are unfiltered by the cache (accesses that make it to DRAM) have a bursty structure. When a processor switches to a new treelet, all threads will suddenly demand new data that is not cached, and must be loaded from DRAM. This memory “barrier” forced by treelets presents an interesting opportunity, as seen in the next slide.



BURSTS MAP TO ROWS

- Treelet nodes fit into contiguous row-sized blocks

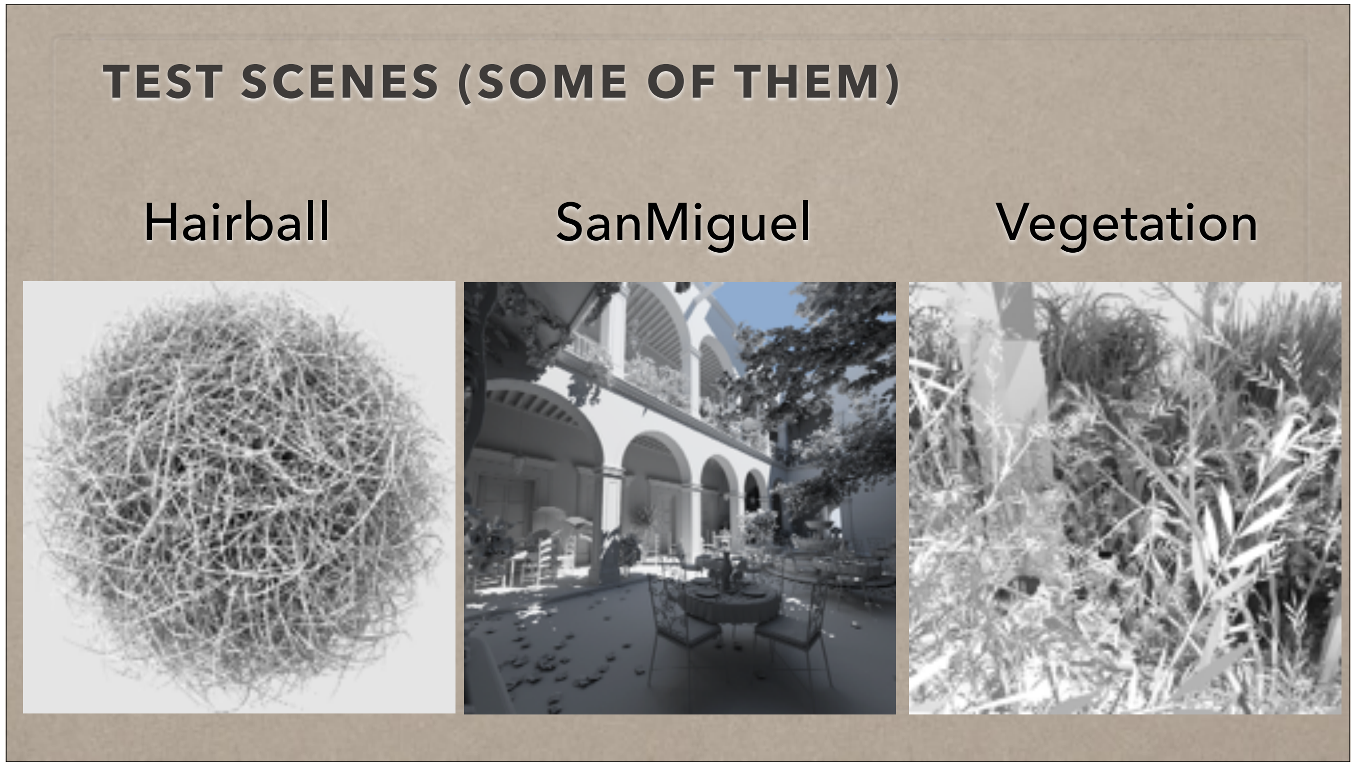
DRAM

Bank 0	Bank 1
Row 0	Row 1
Row 2	Row 3
Row 4	Row 5
Row 6	Row 7
…	…

We rearrange the BVH nodes so that each treelet is one contiguous block of data. Treelets are designed to fit in a multiple of the row buffer size, so that any particular treelet spans as few DRAM rows as possible (2 in the example shown). When a treelet is burst-loaded from memory, it will all come from two open row buffers, greatly amortizing the cost of the loads. Although treelets are sized to fit in the L1 cache as well, this is just a convenience, as our L1 capacities tend to be similar to, or a small multiple of, DRAM row sizes [Jacob et al. 2010].

References:

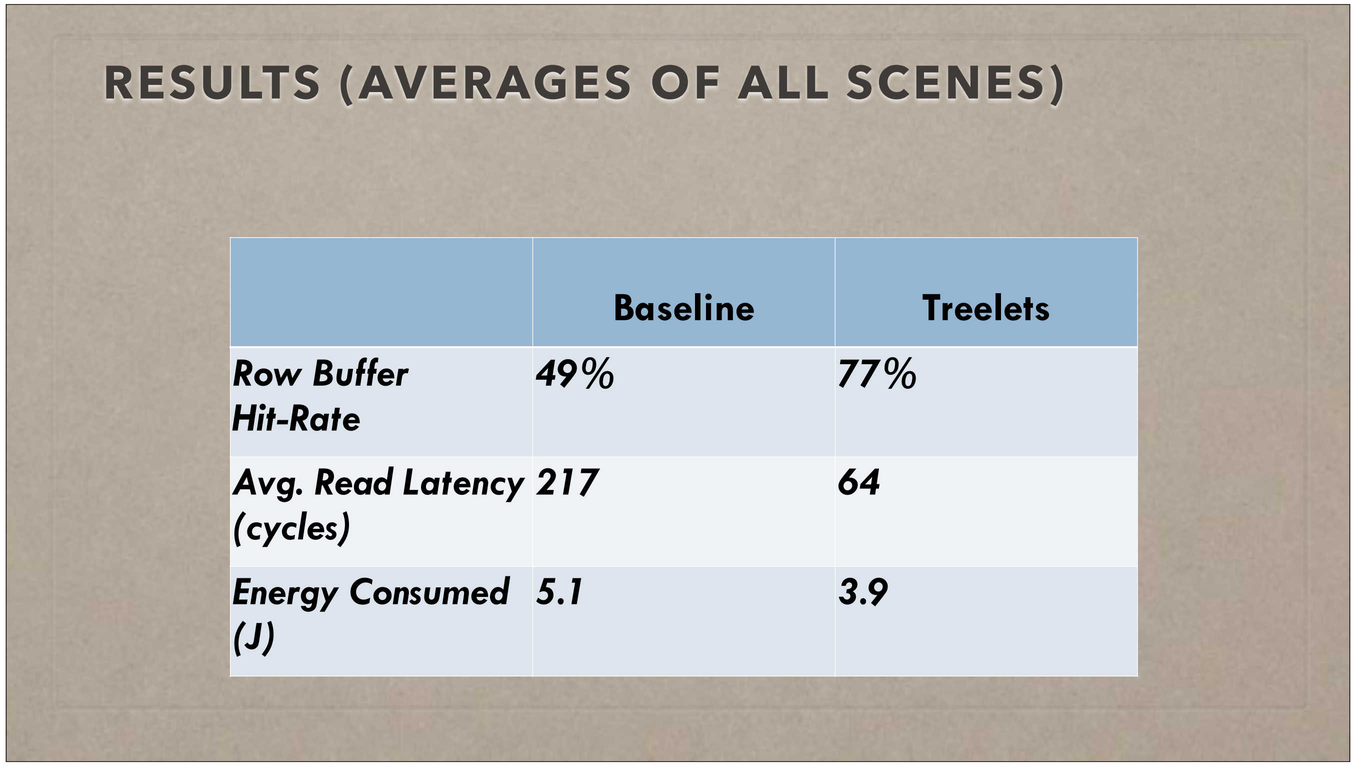
[Jacob et al. 2010] JACOB B., NG S., WANG D.: *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. http://books.google.com/books?id=SrP3aWed-esC.



TEST SCENES (SOME OF THEM)

Hairball SanMiguel Vegetation

We test ray tracing performance for the baseline vs. treelet algorithm in our simulator environment. We use 12 ray tracing benchmark scenes that represent a wide range of geometric size, complexity, and arrangement: Sibenik Cathedral, Fairy Forest, Crytek Sponza, Conference Room, Stanford Dragon, Stanford Buddha, Vegetation, Sodahall, Hairball, and San Miguel. We also use the Dragon and Buddha scenes enclosed in a box to force more ray bounces around the scene. Our tests are rendered at 1024x1024 resolution using Kajiya style path tracing limited to 5 bounces, and a single point light source.



RESULTS (AVERAGES OF ALL SCENES)

	Baseline	Treelets
Row Buffer Hit-Rate	*49%*	*77%*
Avg. Read Latency (cycles)	*217*	*64*
Energy Consumed (J)	*5.1*	*3.9*

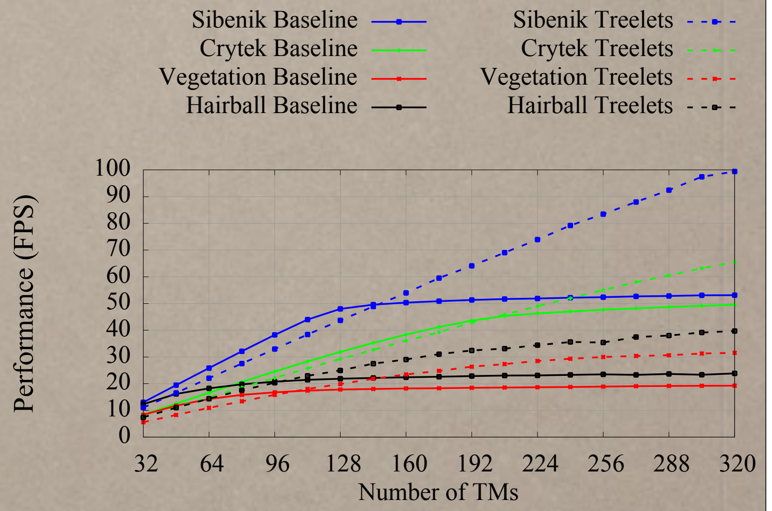
Data collected from [Kopta et al. 2014].

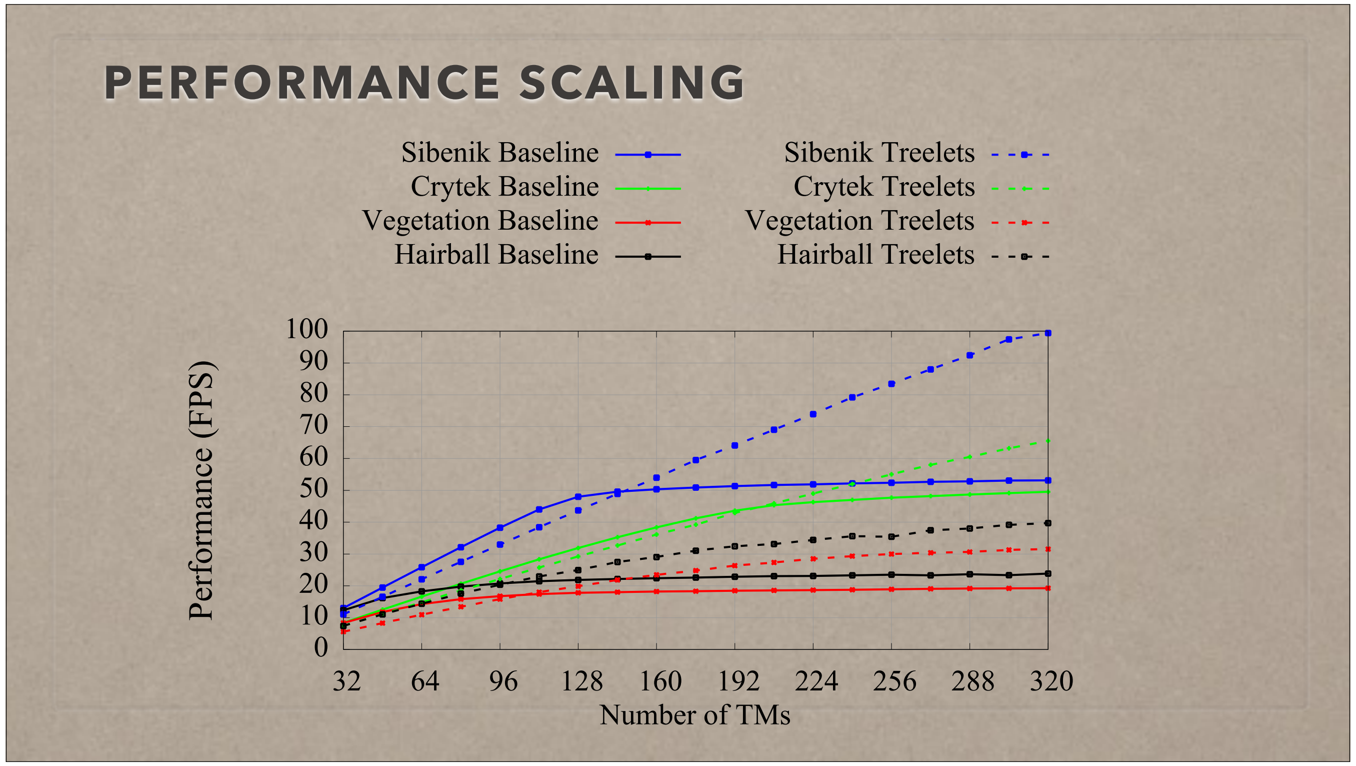
References:

[Kopta et al. 2014] KOPTA, D., SHKURKO, K., SPJUT, J., BRUNVAND, E., AND DAVIS, A. 2014. Memory Considerations for Low Energy Ray Tracing. Computer Graphics Forum. To Appear.

RESULTS

- DRAM energy reduced by up to 43%
  - Latency by up to 80%
- Higher row buffer hit rate → closer to peak bandwidth
  - Performance scales better with more threads





PERFORMANCE SCALING

As we increase the number of compute cores available, performance increases, but only to a certain point. The naïve baseline technique's performance (solid lines) quickly plateaus due to data starvation at the main memory bottleneck. Using treelets (dashed lines) with the resulting increased row buffer hit rate enables the chip to approach closer to the peak bandwidth capabilities of DRAM, and thus more cores can be fed with data. This figure shows a selection of some of our benchmark scenes [Kopta et al. 2014].

References:

[Kopta et al. 2014] KOPTA, D., SHKURKO, K., SPJUT, J., BRUNVAND, E., AND DAVIS, A. 2014. Memory Considerations for Low Energy Ray Tracing. Computer Graphics Forum. To Appear.

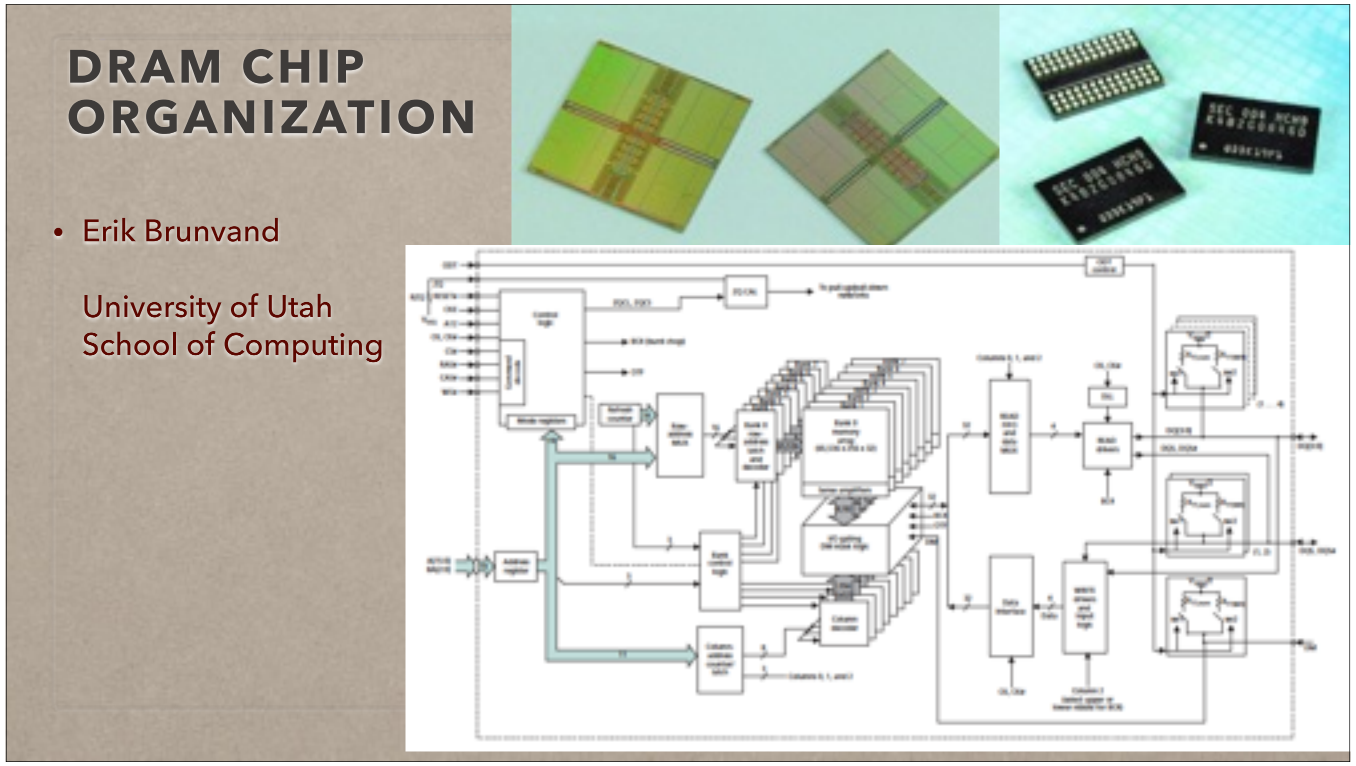
SURPRISE RESULT

- Energy / latency can decrease even when consumed bandwidth increases!
  - Example (avg. latency / DRAM access):

    70M reads @ 55% RB hit rate: 132ns
    78M reads @ 80% RB hit rate: 31ns

  - Example (Total DRAM Energy / frame)

    133M reads @31% RB hit rate: 8.6J/frame
    224M reads @78% RB hit rate: 6.8J/frame

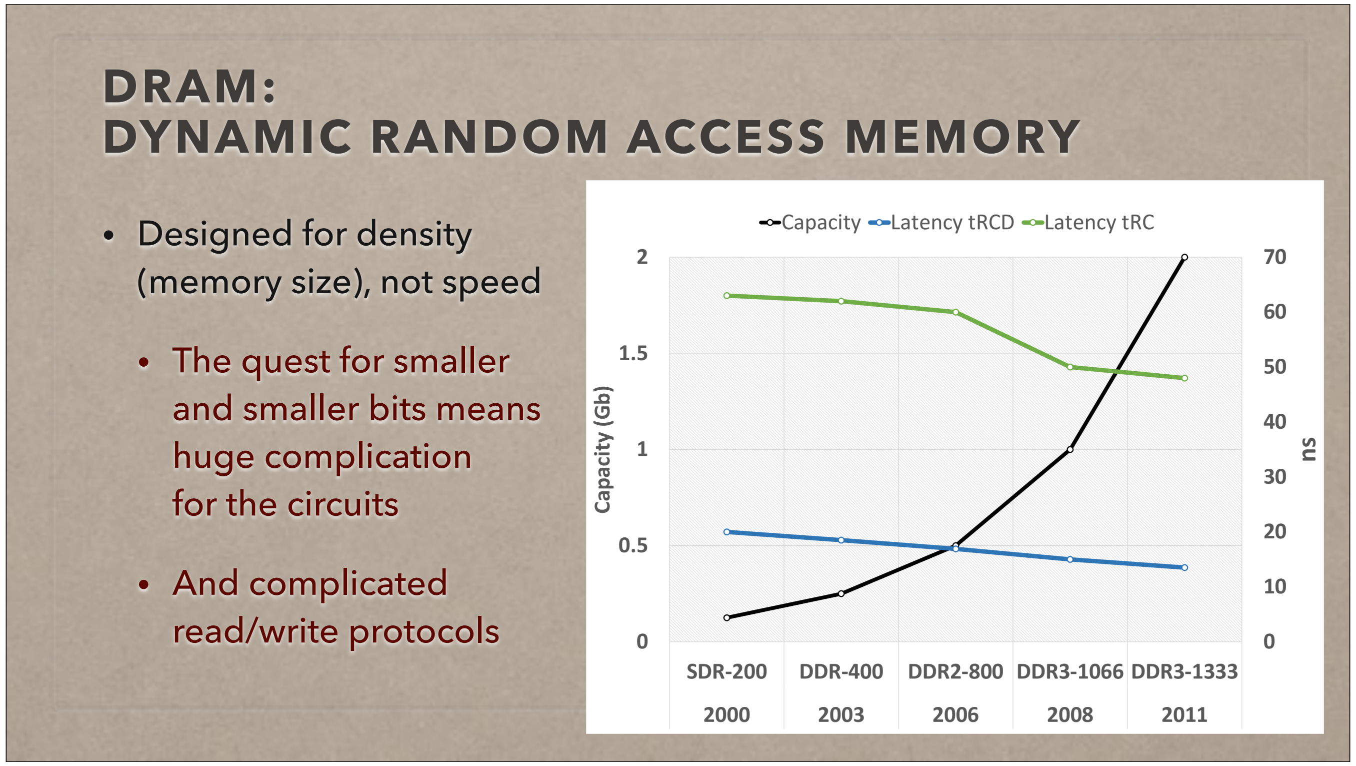


DRAM CHIP ORGANIZATION

- Erik Brunvand

  University of Utah
  School of Computing

DRAM behavior at the circuit and chip level is the fundamental source of much of the complexity, and the speed and power overheads, of this type of memory. I’m only going to scratch the surface of exactly how these chips work, but we think it’s important to know at least a little about *why* these chips are so slow and complex.



DRAM: DYNAMIC RANDOM ACCESS MEMORY

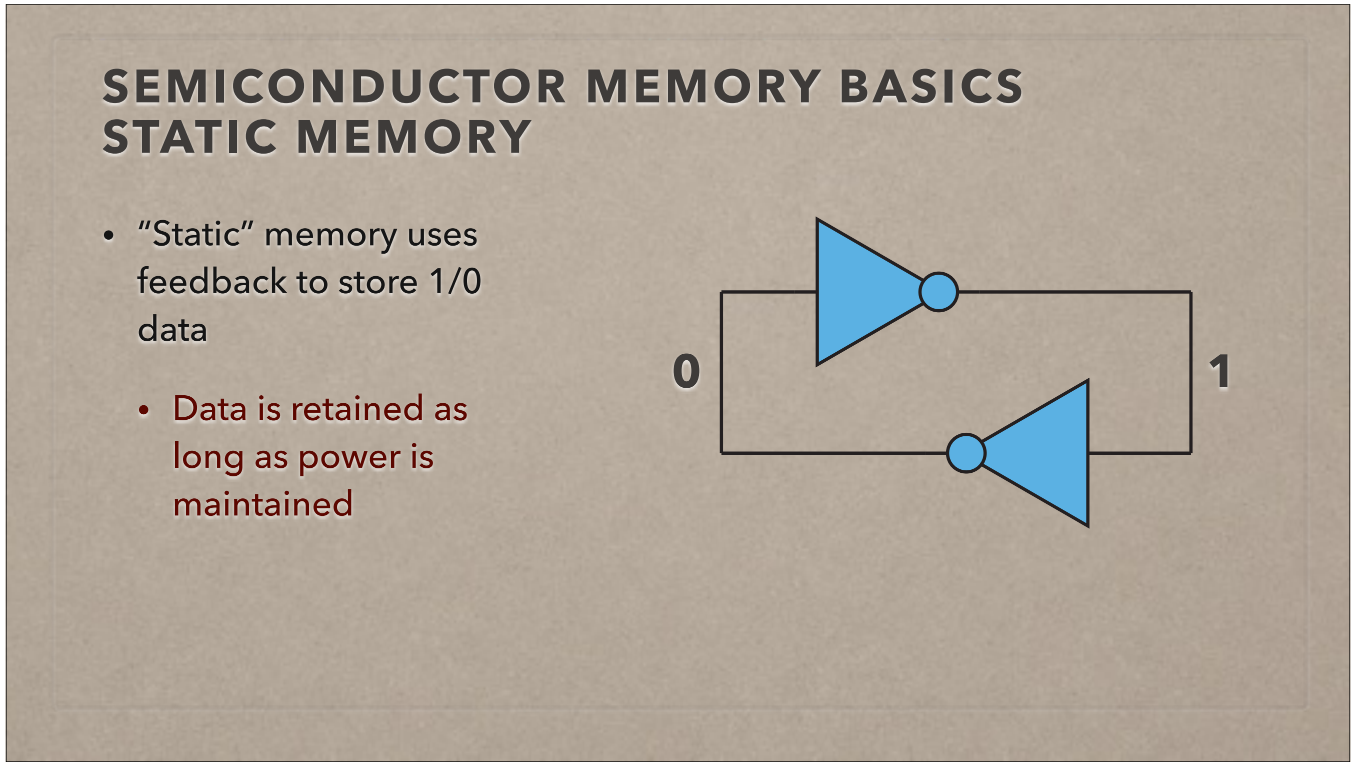
- Designed for density (memory size), not speed
  - The quest for smaller and smaller bits means huge complication for the circuits
  - And complicated read/write protocols

Note - capacity is going way up, but latency is not changing much. This is because of the fundamental circuits used for DRAM, and how long it takes to read/write to those structures.

CPUs are getting faster by around 58%/yr

DRAM is getting faster by around 7%/yr (or less)

This results in a growing gap between the speed of processors and the speed of memory. What's not shown in this diagram is that power of DRAMs is also not going down much. This results in a growing power gap that is becoming at least (if not more) worrisome than the latency gap.



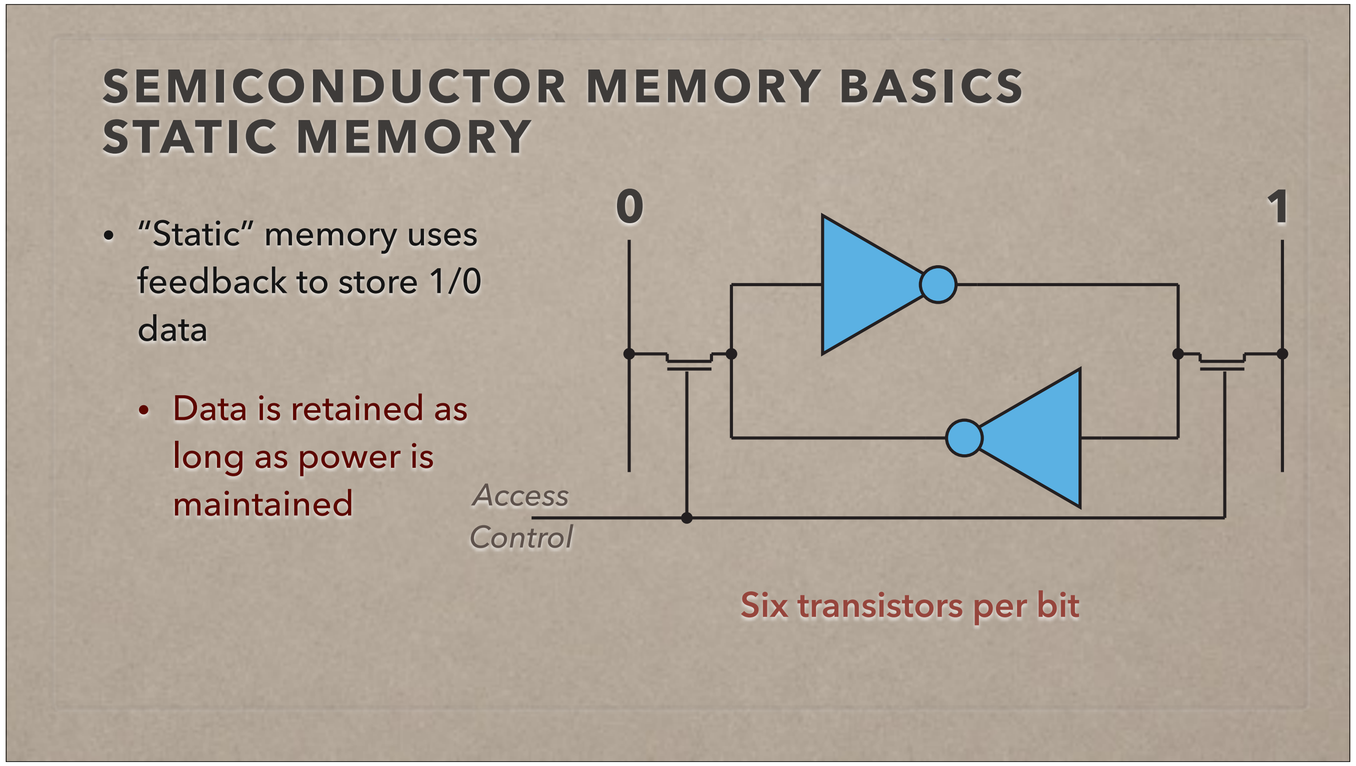
SEMICONDUCTOR MEMORY BASICS STATIC MEMORY

- “Static” memory uses feedback to store 1/0 data
  - Data is retained as long as power is maintained

In order to understand DRAM behavior, and why it works the way it does, it’s helpful to contrast DRAM (Dynamic RAM) with SRAM (Static RAM).

Static Ram is the type of RAM used in most places on-chip (RF, caches, etc.) The fundamental circuit is a pair of inverter circuits feeding back on each other. The feedback, combined with the gain (amplification) inherent in these circuits, reinforces the state of the circuit. If the input to the top inverter is 0, then the output is 1. This makes the output of the bottom inverter 0, which reinforces the state.

- The opposite state of this circuit is where the 1 value is on the left and the 0 value is on the right. That state is also reinforced using the same feedback argument.
- As long as the power is on, the feedback/gain of the circuit keeps the data stable in the “bit”
- Of course, this isn’t all that interesting unless you can change the bit’s value when you want to write new data.



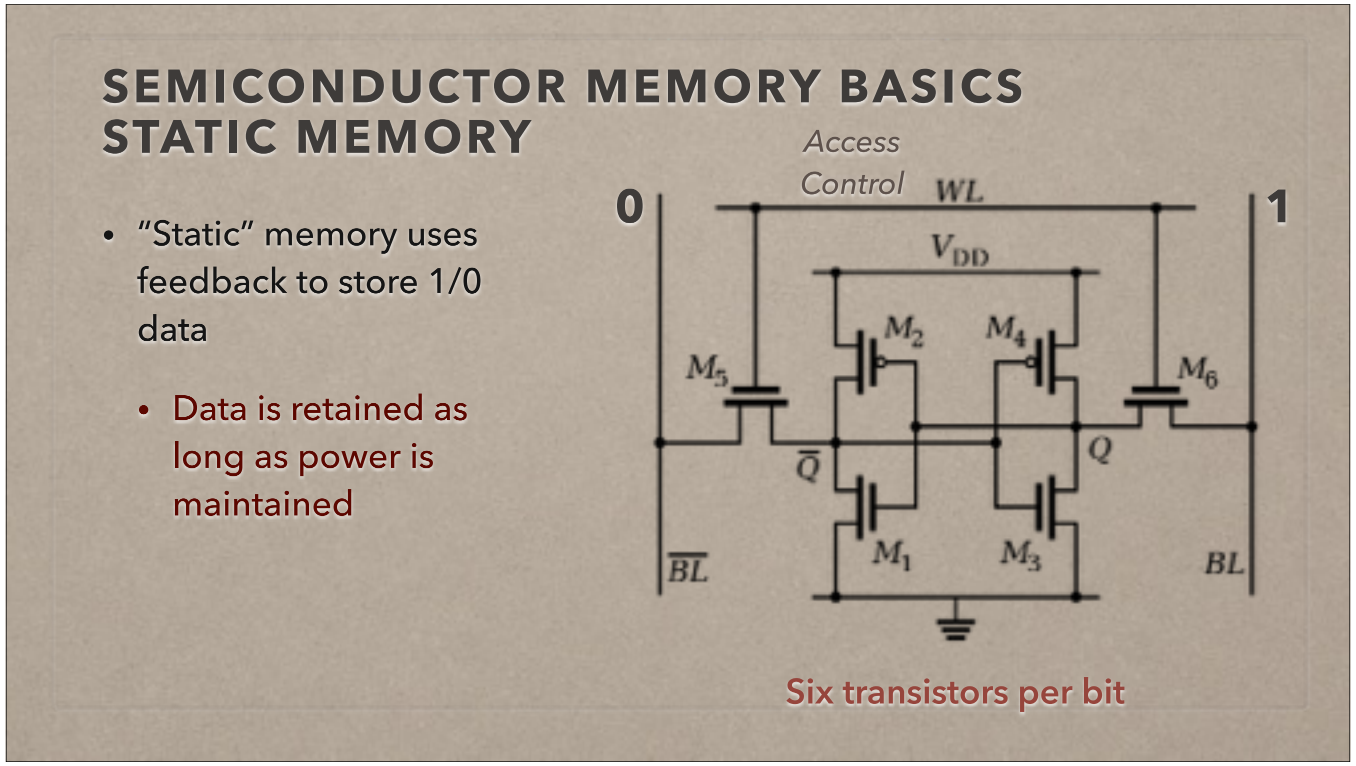
SEMICONDUCTOR MEMORY BASICS
STATIC MEMORY

- “Static” memory uses feedback to store 1/0 data
    - Data is retained as long as power is maintained

Six transistors per bit

- Writing new data into a bit cell involves overdriving the values in the cell. In order to do that, we use access transistors that are used to connect the bit cell to “bit lines” that can be used to drive new data to the cell.
- If the values being driven onto the bit lines are stronger than the little inverters in the bit, they can be overdriven to the new value.
- Reading the bit involves coupling the inverters to the same bit lines (not being driven at the time) and sensing the voltage change on the wires.
- Because the bit cell has active drivers (even if they’re small), the voltage change on the bit lines can be sensed quite quickly.
- This explains why RF and small caches can be read in a single processor cycle.
- Each bit cell can be made with six transistors (2 each for the inverters)
- Of course, the support circuits (address decoders, sense amplifiers, write-drivers, etc.) add some transistors to the mix, but the bit cell arrays are the lion’s share of the transistors in a memory structure.

Six transistors doesn’t seem like many, but it’s too many for really dense memory!



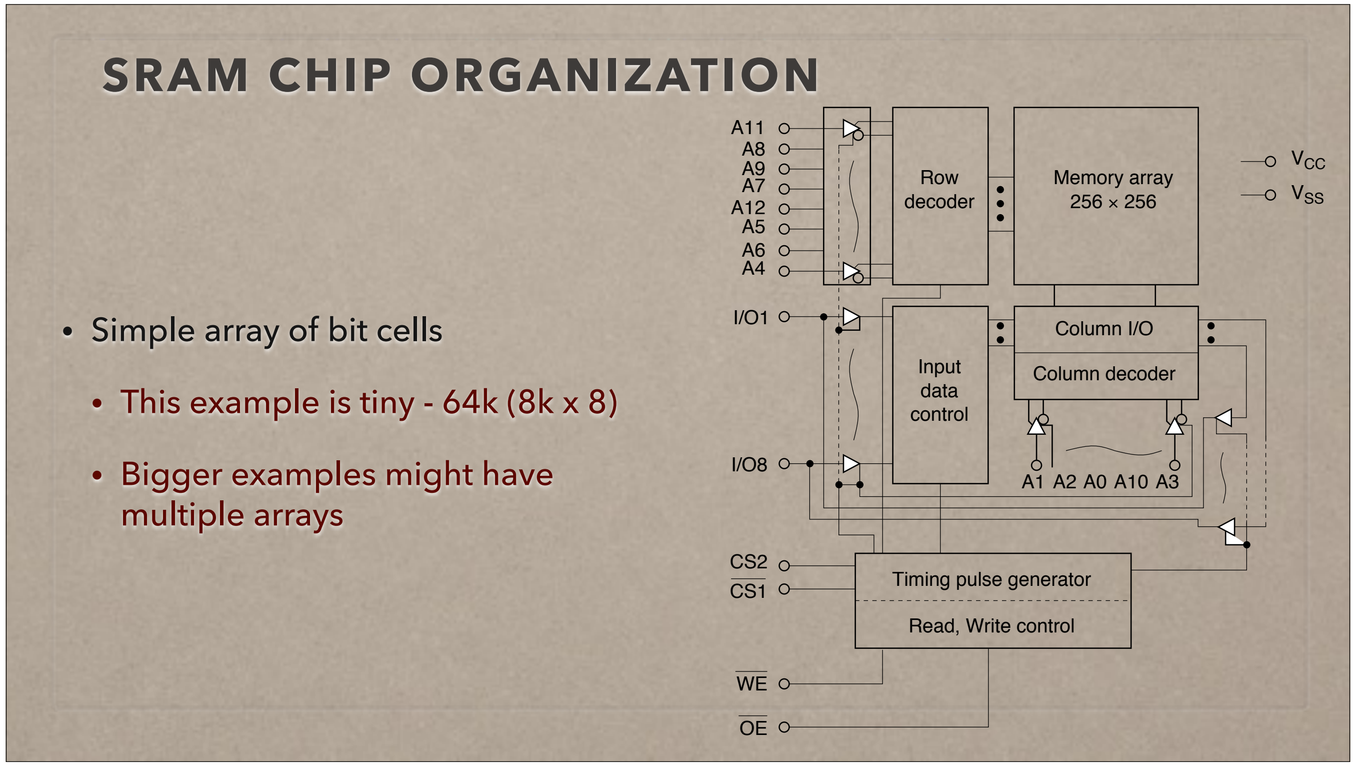
SEMICONDUCTOR MEMORY BASICS
STATIC MEMORY

- "Static" memory uses feedback to store 1/0 data
  - Data is retained as long as power is maintained

Six transistors per bit

- Writing new data into a bit cell involves overdriving the values in the cell. In order to do that, we use access transistors that are used to connect the bit cell to "bit lines" that can be used to drive new data to the cell.
- If the values being driven onto the bit lines are stronger than the little inverters in the bit, they can be overdriven to the new value.
- Reading the bit involves coupling the inverters to the same bit lines (not being driven at the time) and sensing the voltage change on the wires.
- Because the bit cell has active drivers (even if they're small), the voltage change on the bit lines can be sensed quite quickly.
- This explains why RF and small caches can be read in a single processor cycle.
- Each bit cell can be made with six transistors (2 each for the inverters)
- Of course, the support circuits (address decoders, sense amplifiers, write-drivers, etc.) add some transistors to the mix, but the bit cell arrays are the lion's share of the transistors in a memory structure.

Six transistors doesn't seem like many, but it's too many for really dense memory!

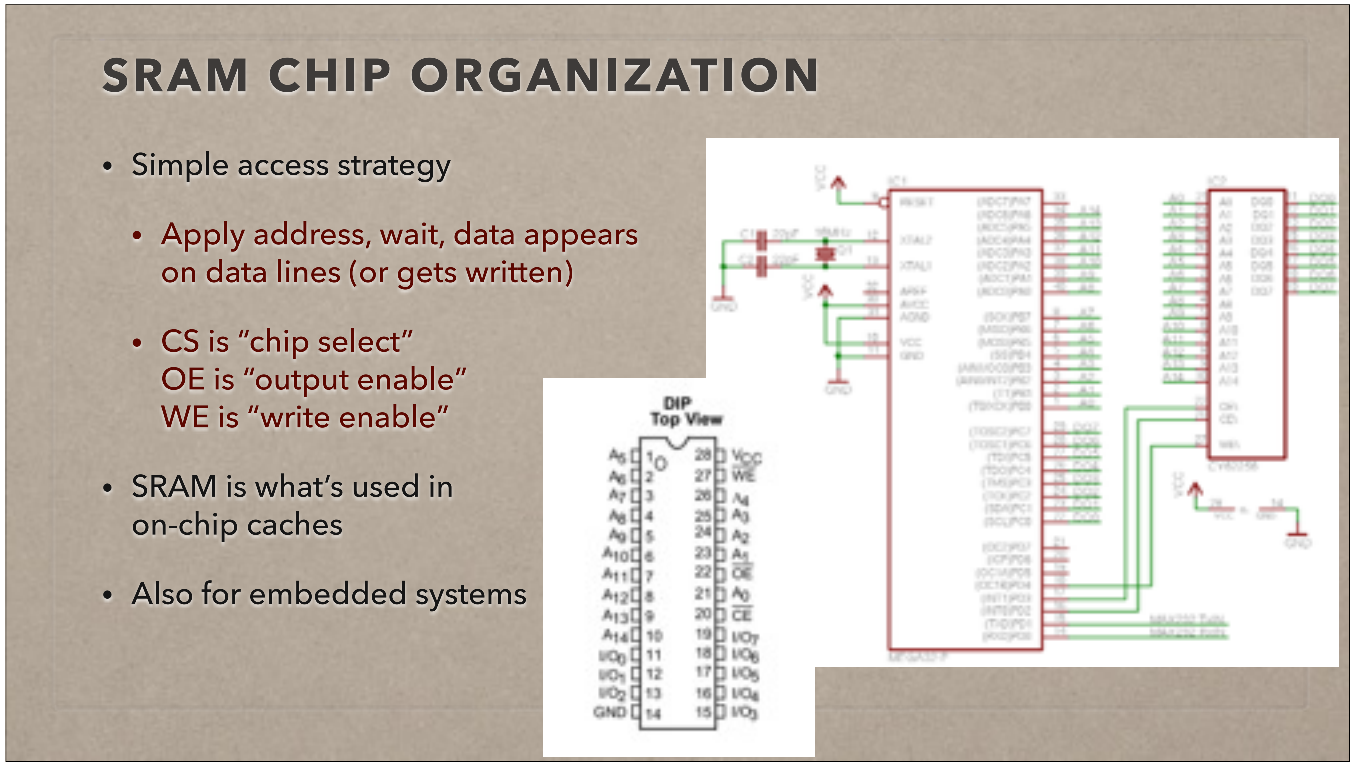


SRAM CHIP ORGANIZATION

- Simple array of bit cells
  - This example is tiny - 64k (8k x 8)
  - Bigger examples might have multiple arrays

Here's the organization of a simple SRAM

- There's an array of those 6-transistor bits
- A decoder for the portion of the address that maps to a row, and the portion of the address that maps to the column
  - The row portion identifies a row of the array to access
  - The column portion picks a bit (or group of bits) out of that row to read/write.
  - That data is delivered to / written from the external I/O pins

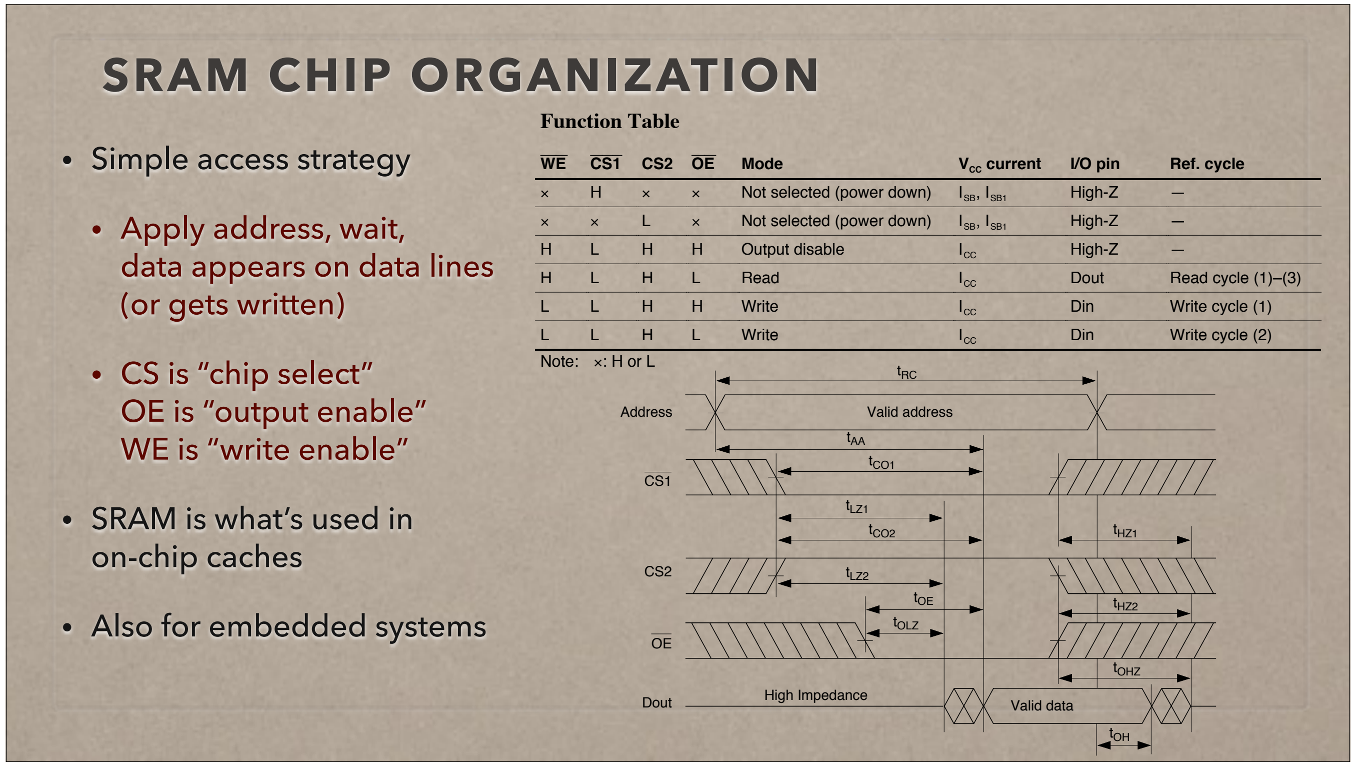


SRAM CHIP ORGANIZATION

- Simple access strategy
  - Apply address, wait, data appears on data lines (or gets written)
  - CS is “chip select”
    OE is “output enable”
    WE is “write enable”
- SRAM is what’s used in on-chip caches
- Also for embedded systems

The access operation at the chip level is very simple:

- Read: send the address to the chip, along with OE, wait for the data to come back (Async)
- Write: set data to be written on the data lines, and address on the address line. Assert WE, and wait.



SRAM CHIP ORGANIZATION

- Simple access strategy
  - Apply address, wait, data appears on data lines (or gets written)
  - CS is “chip select”
    OE is “output enable”
    WE is “write enable”
- SRAM is what’s used in on-chip caches
- Also for embedded systems

Function Table

$\overline{WE}$	$\overline{CS1}$	CS2	$\overline{OE}$	Mode	V_{cc} current	I/O pin	Ref. cycle
×	H	×	×	Not selected (power down)	I_{SB}, I_{SB1}	High-Z	—
×	×	L	×	Not selected (power down)	I_{SB}, I_{SB1}	High-Z	—
H	L	H	H	Output disable	I_{CC}	High-Z	—
H	L	H	L	Read	I_{CC}	Dout	Read cycle (1)–(3)
L	L	H	H	Write	I_{CC}	Din	Write cycle (1)
L	L	H	L	Write	I_{CC}	Din	Write cycle (2)

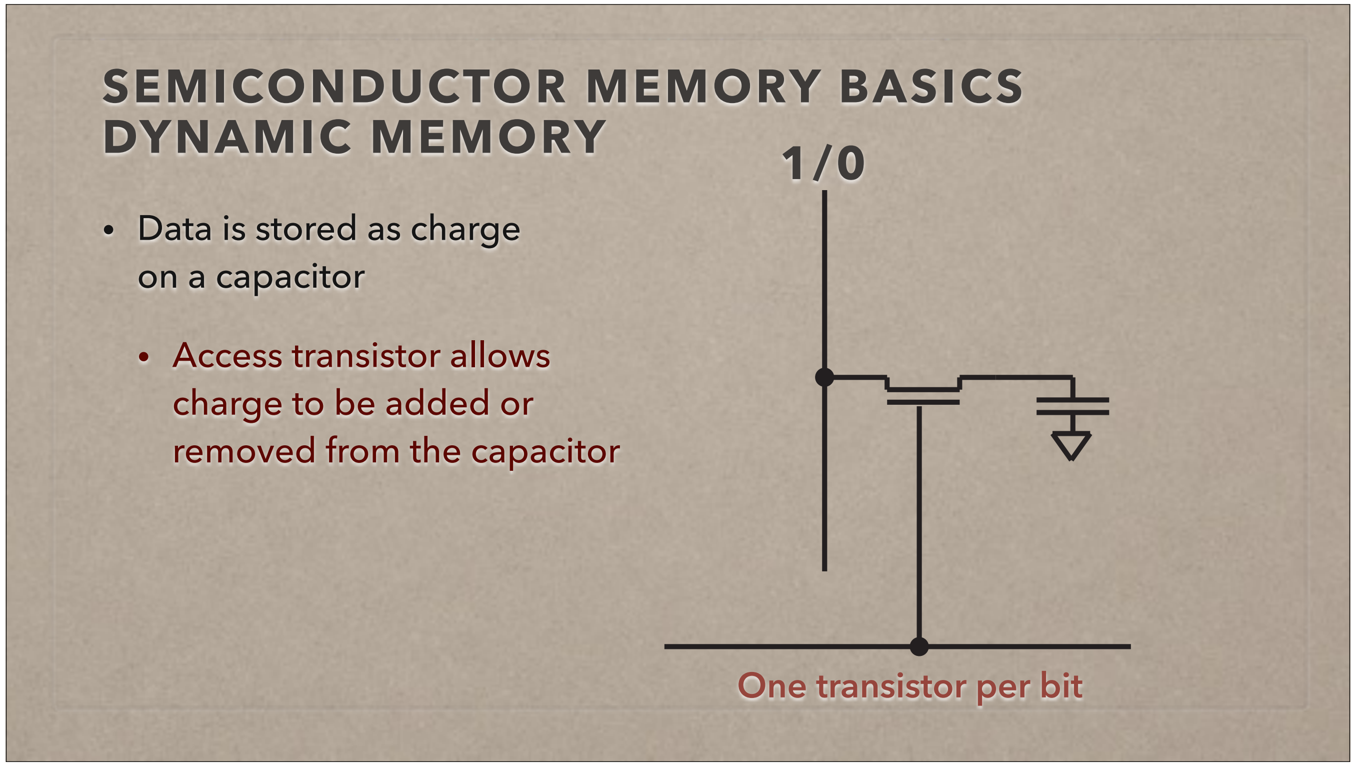
Note: ×: H or L

The access operation at the chip level is very simple:

Read: send the address to the chip, along with OE, wait for the data to come back (Async)

Write: set data to be written on the data lines, and address on the address line. Assert WE, and wait.

The timing chart shows pretty much all the details you need - mostly how long to wait for operations to be completed, and how the timing of the two control lines (OE and WE) related to the address and data.



SEMICONDUCTOR MEMORY BASICS
DYNAMIC MEMORY

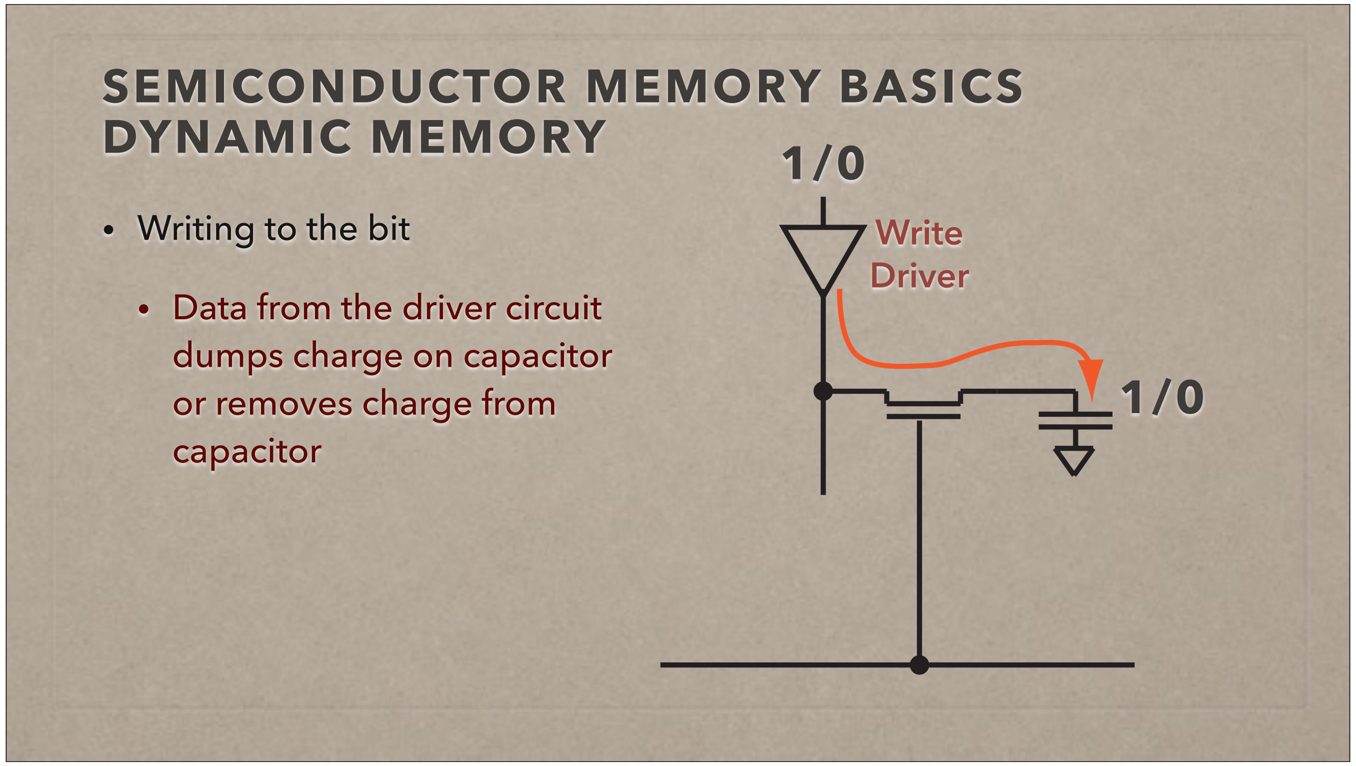
- Data is stored as charge on a capacitor
  - Access transistor allows charge to be added or removed from the capacitor

1/0

One transistor per bit

DRAM uses only one transistor per bit. The value is stored as charge on a (tiny) capacitor. This is about as simple as it gets!

- This makes the area per bit cell potentially very small.
  - But only if you can make the capacitor very efficiently.
- But, it makes a lot of things trickier because of that obsession with size.
- Data in the bit cell is stored as charge on the gate of the capacitor.
  - Charge the cap up to a high voltage for a 1, discharge to a low voltage for a 0, for example.
- But, in order to keep the size of the cell small, the cap should be physically small too. This means that there’s not much charge stored.
  - It makes it a challenge to sense whether the bit is charged or discharged.

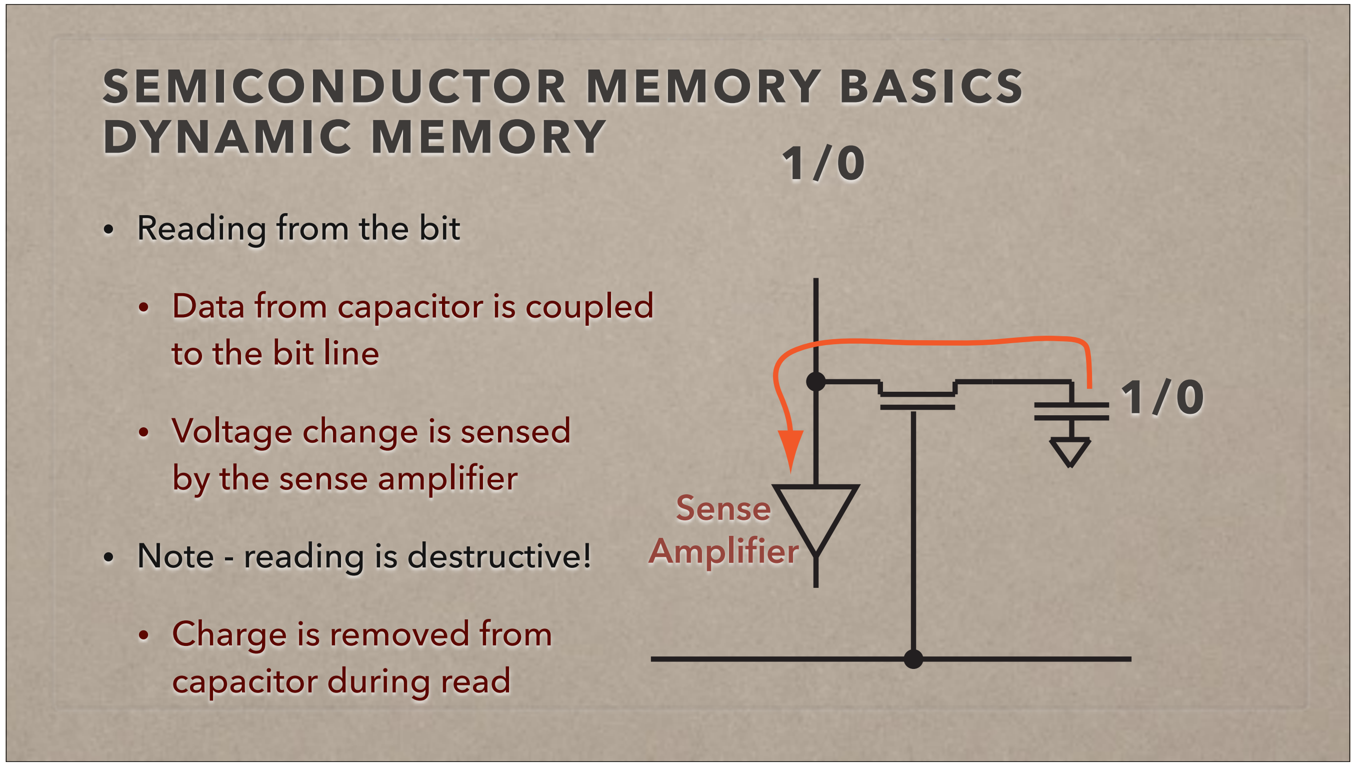


SEMICONDUCTOR MEMORY BASICS
DYNAMIC MEMORY

- Writing to the bit
  - Data from the driver circuit dumps charge on capacitor or removes charge from capacitor

Back to the single-transistor bit cell:

- Writing to the capacitor involves dumping charge onto the capacitor, or removing charge from the capacitor.
- A write driver drives a strong signal onto the bit line.
- The access transistor allows that value to be set onto the capacitor.



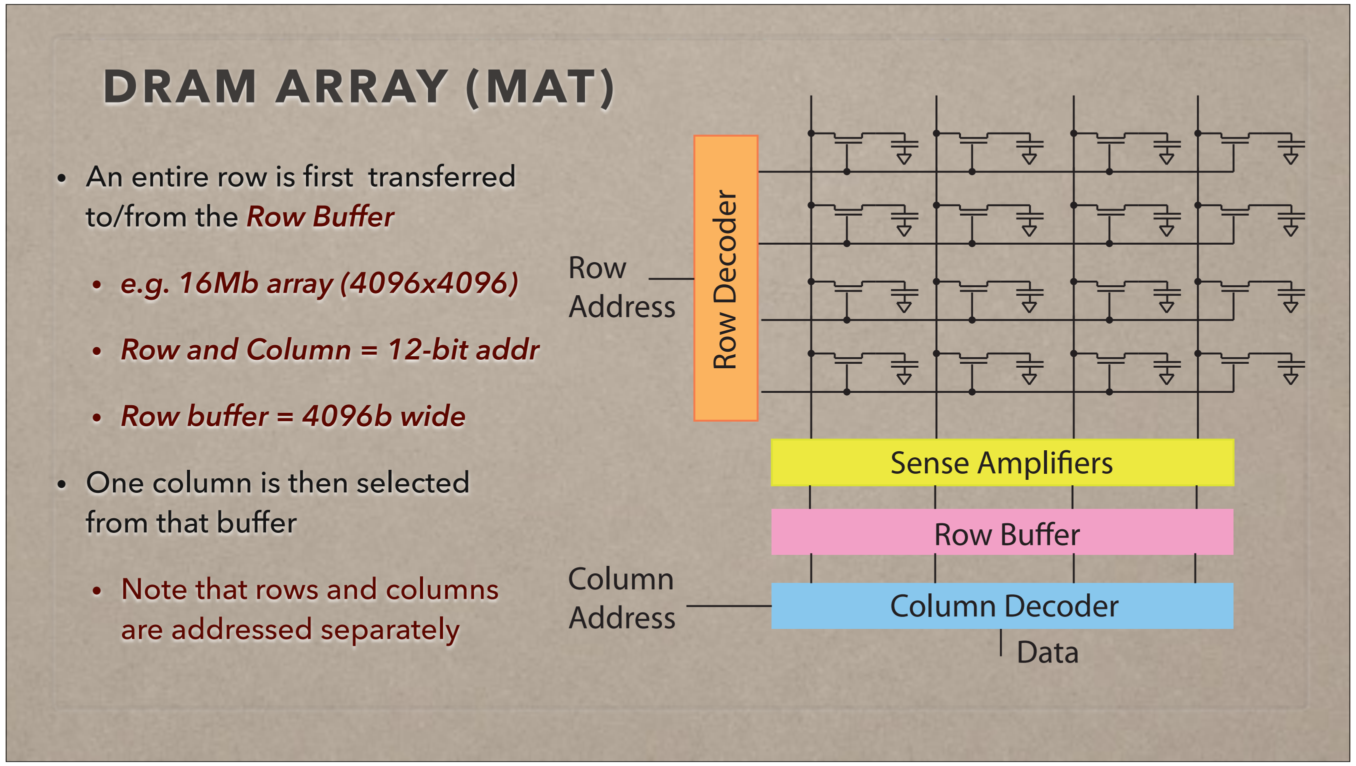
SEMICONDUCTOR MEMORY BASICS
DYNAMIC MEMORY

- Reading from the bit
  - Data from capacitor is coupled to the bit line
  - Voltage change is sensed by the sense amplifier
- Note - reading is destructive!
  - Charge is removed from capacitor during read

- Reading from the cell involves coupling the capacitor to the bit line and sensing the voltage change.
- Note that these caps are really really small, so the voltage change is also really really small.
- This means that sensitive analog sense amplifiers are needed to accurately evaluate that change - In this case “sensitive” also means “slow”
- The bit line is typically discharged to Vdd/2, and a second dummy line is also “precharged” to the same value. Then the bit is coupled to the bit line, and the difference between the two is sensed.

The worst part, though is that reading is destructive because the charge is shared with the charge of the bit line.

- So, data needs to be restored after a read. Every read is followed by a write-back to the bit cell that was read.
  - This has both speed and power implications!



DRAM ARRAY (MAT)

- An entire row is first transferred to/from the *Row Buffer*
    - *e.g. 16Mb array (4096x4096)*
    - *Row and Column = 12-bit addr*
    - *Row buffer = 4096b wide*
- One column is then selected from that buffer
    - Note that rows and columns are addressed separately

Row Address

Row Decoder

Sense Amplifiers

Row Buffer

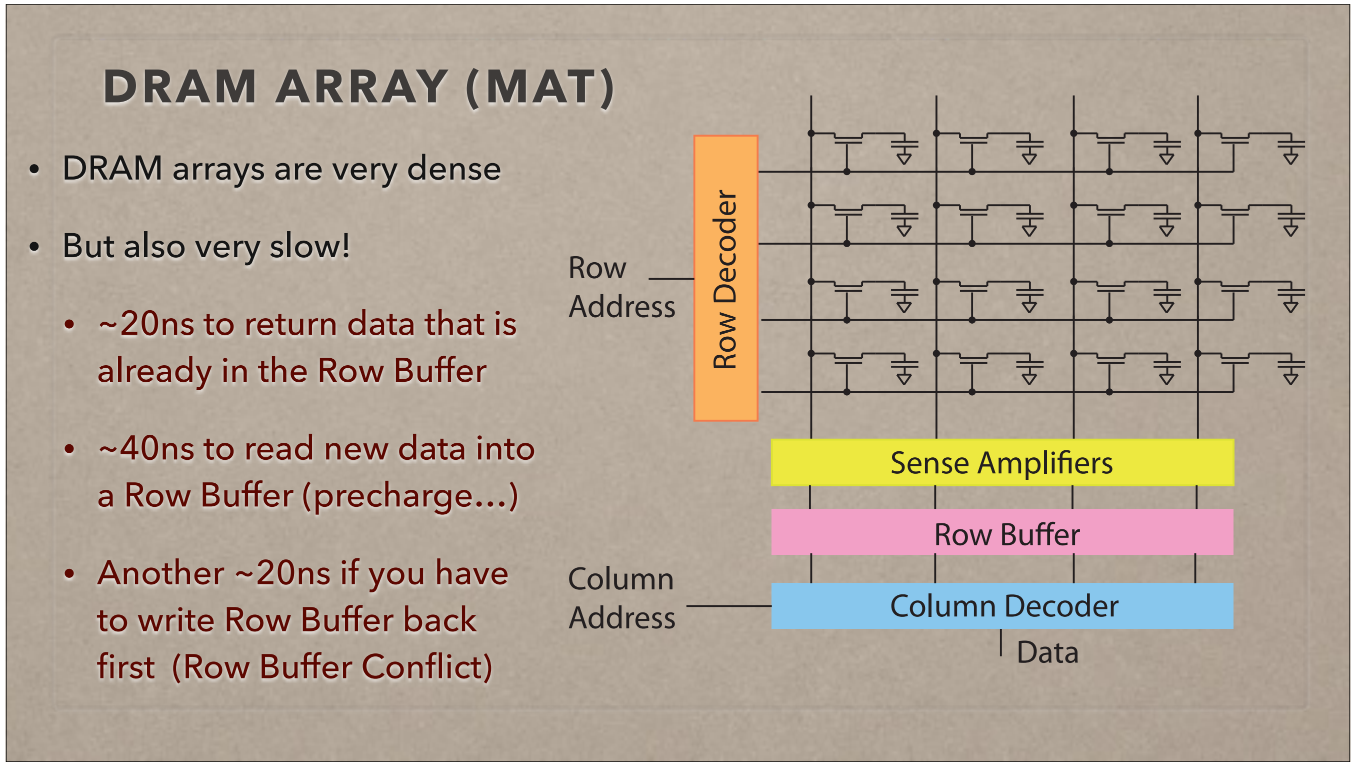
Column Address

Column Decoder

Data

The basic organization of DRAM is an "array" or "mat" of bit cells, surrounded by some support circuits that decode addresses, and control the behavior of the sense amplifiers and write-drivers.

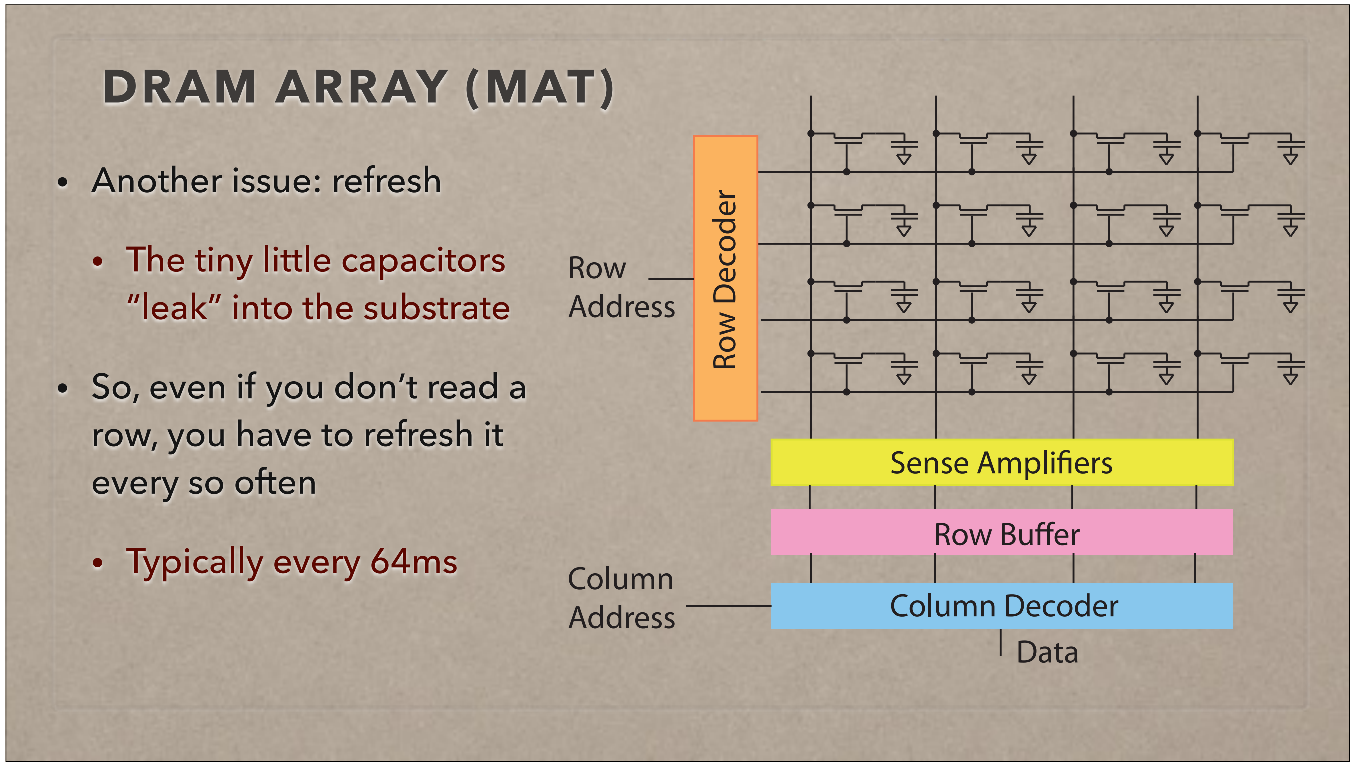
- For a read: First a row is addressed through the row decoder.
    - An entire row is transferred to the Row Buffer (which is made of Static RAM)
- Then a single bit is selected from the Row Buffer using the Column Address and delivered to the user.
- Because reading destroyed the values in that row, the row is then written back to the DRAM array
- Note that rows and columns are addressed separately, and sequentially.
    - Good news: this means that you only need half the address pins on the chip!
    - Bad news: it's sequential so it's twice as slow.
- The Row Buffer is fascinating though…
    - Once an entire row is in the Row Buffer, you can read bits out of that Row Buffer by just changing the Column Address.
    - The Row Buffer acts like a "Row Cache" - if there's data in the Row Buffer, it can be read much more quickly than a "Row Buffer Miss"



DRAM ARRAY (MAT)

- DRAM arrays are very dense
- But also very slow!
  - ~20ns to return data that is already in the Row Buffer
  - ~40ns to read new data into a Row Buffer (precharge…)
  - Another ~20ns if you have to write Row Buffer back first (Row Buffer Conflict)

- In addition to being relatively complex, this organization is slow (latency)
- The process of reading a bit involves
  - Precharging all the bit lines
  - Then sending the Row Address through the row decoder to couple one row onto the bit lines
  - The sense amps then detect which bit lines are 1 and which are 0
  - These values are captured in the Row Buffer (Static Ram)
  - Those values are now (or later) written back to the row that you just read (and destroyed in the process of reading).
- So, the amount of time it takes to read data is very different depending on whether the data is in the Row Buffer already or not (an “open row hit” vs. a “row miss”).
- These times don’t look all that long, but consider that a 1GHz a processor cycle is 1ns. At 4GHz, a 60ns DRAM access is 240 processor cycles…



DRAM ARRAY (MAT)

- Another issue: refresh
  - The tiny little capacitors "leak" into the substrate
- So, even if you don't read a row, you have to refresh it every so often
  - Typically every 64ms

Row Address

Row Decoder

Sense Amplifiers

Row Buffer

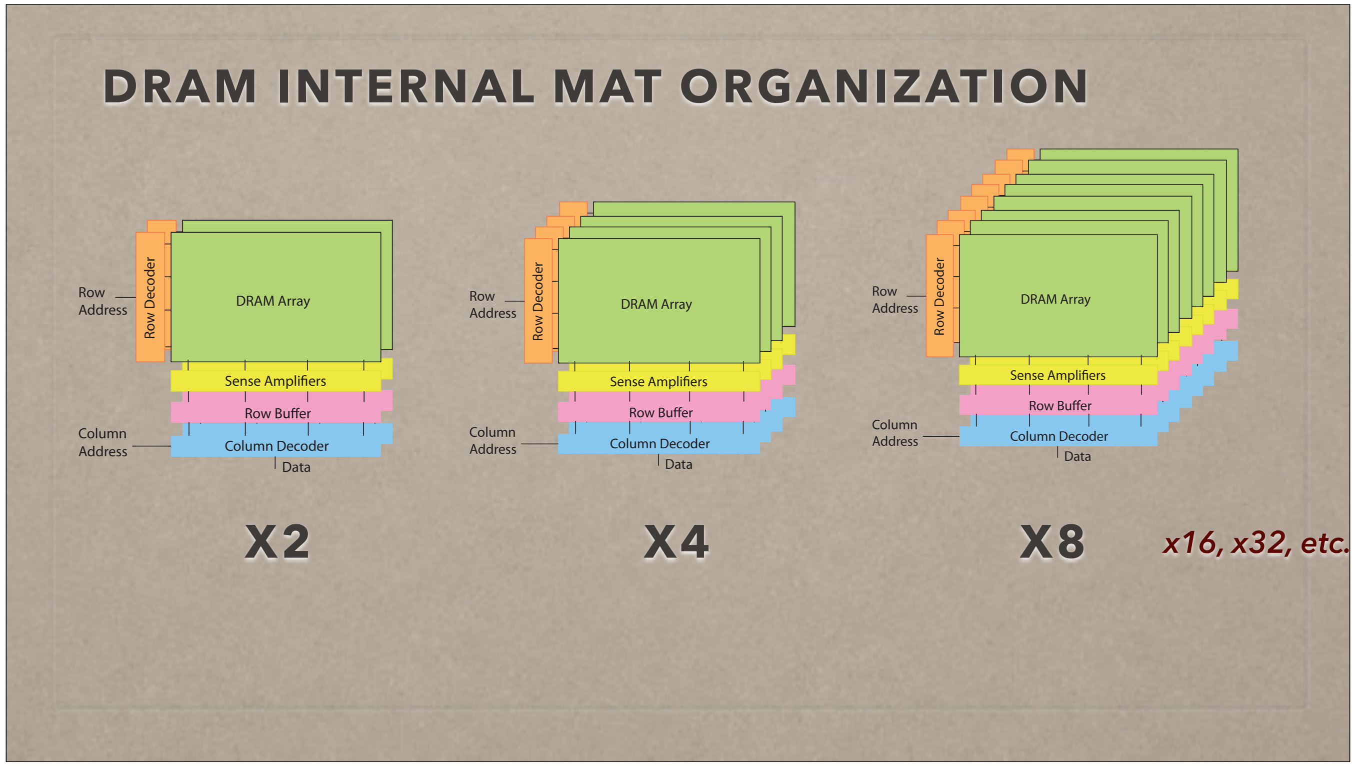
Column Address

Column Decoder

Data

The other implication of the small storage caps used to store bits: they're not perfectly isolated caps.

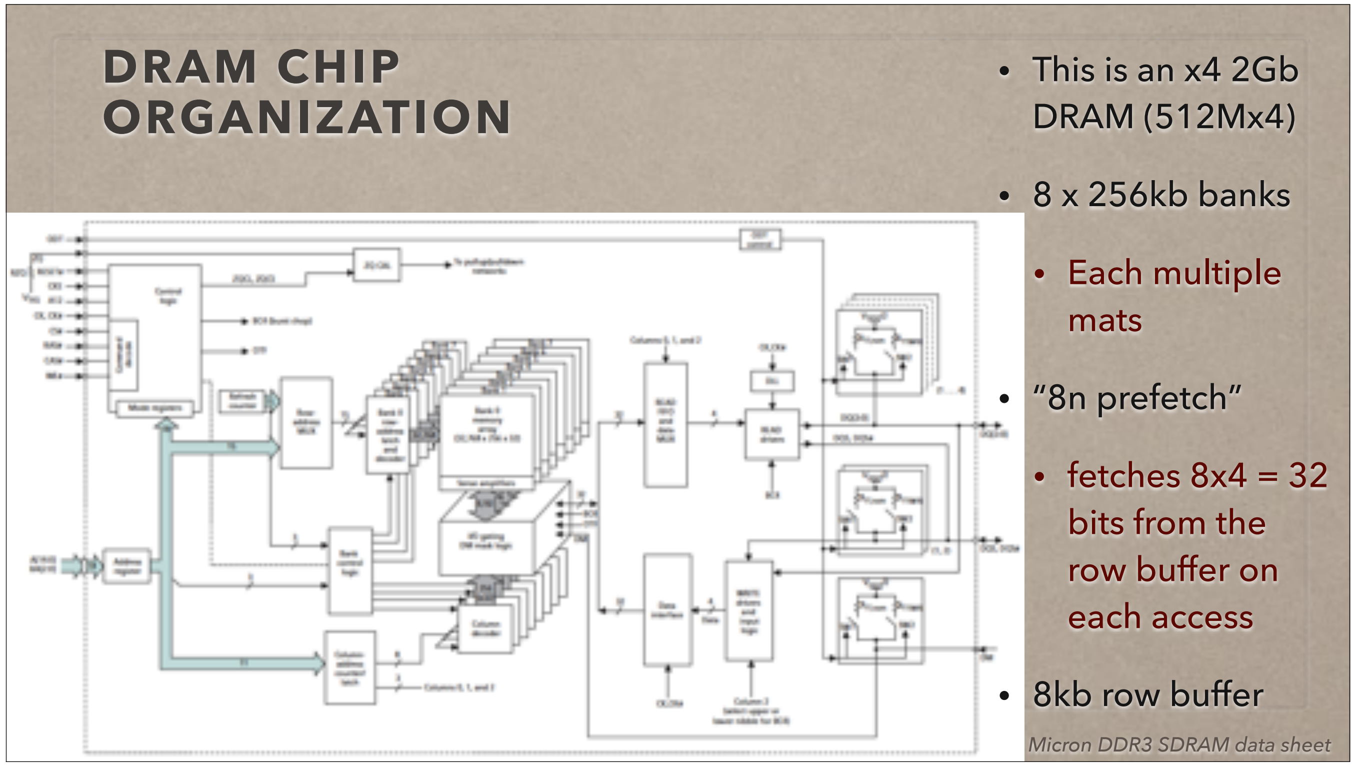
- This means that they "leak" their charge into the substrate
- That means that they lose their data over time (a short time)
- Every cell in the bit array needs to be refreshed in some fixed amount of time - typically 32ms to 64ms.
  - ms are pretty long compared to ns, but this still adds up to 10% or more of the time spent dealing with DRAM.
  - It also adds to the complexity of the controller - someone needs to keep track of which cells need refreshing, and when.



DRAM INTERNAL MAT ORGANIZATION

The column decoder on a DRAM array (mat) typically returns only a single bit from the Row Buffer.

- DRAM chips return a larger (but still relatively small) data word
    - Typically between 4-16 bits per column address
- This is accomplished by powering up multiple mats per access.
- There are lots of mats on a chip!
- A mat is basically the chunk size of memory that the DRAM circuit designers can build that stays reliable and stays within the latency requirements.
    - you get more speed/bandwidth by putting them in parallel

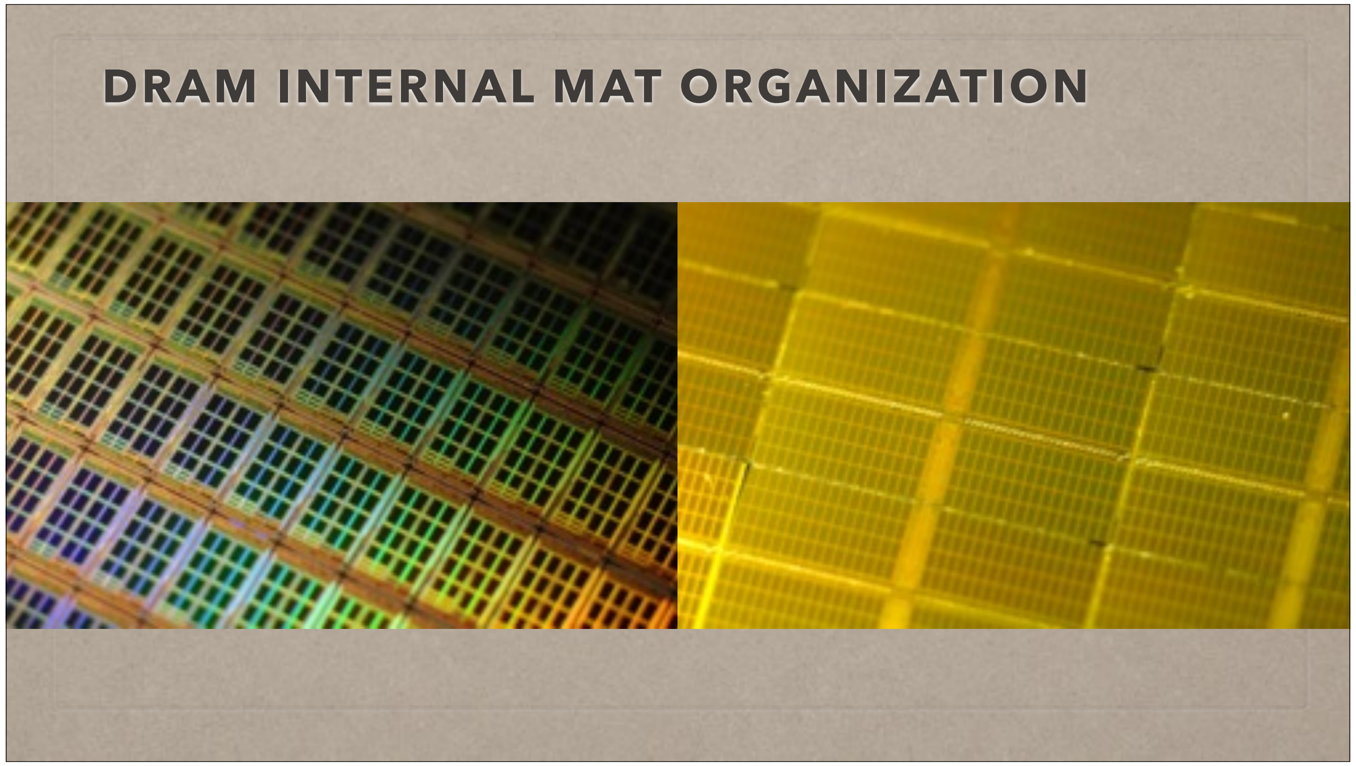


DRAM CHIP ORGANIZATION

- This is an x4 2Gb DRAM (512Mx4)
- 8 x 256kb banks
  - Each multiple mats
- "8n prefetch"
  - fetches 8x4 = 32 bits from the row buffer on each access
- 8kb row buffer

Micron DDR3 SDRAM data sheet

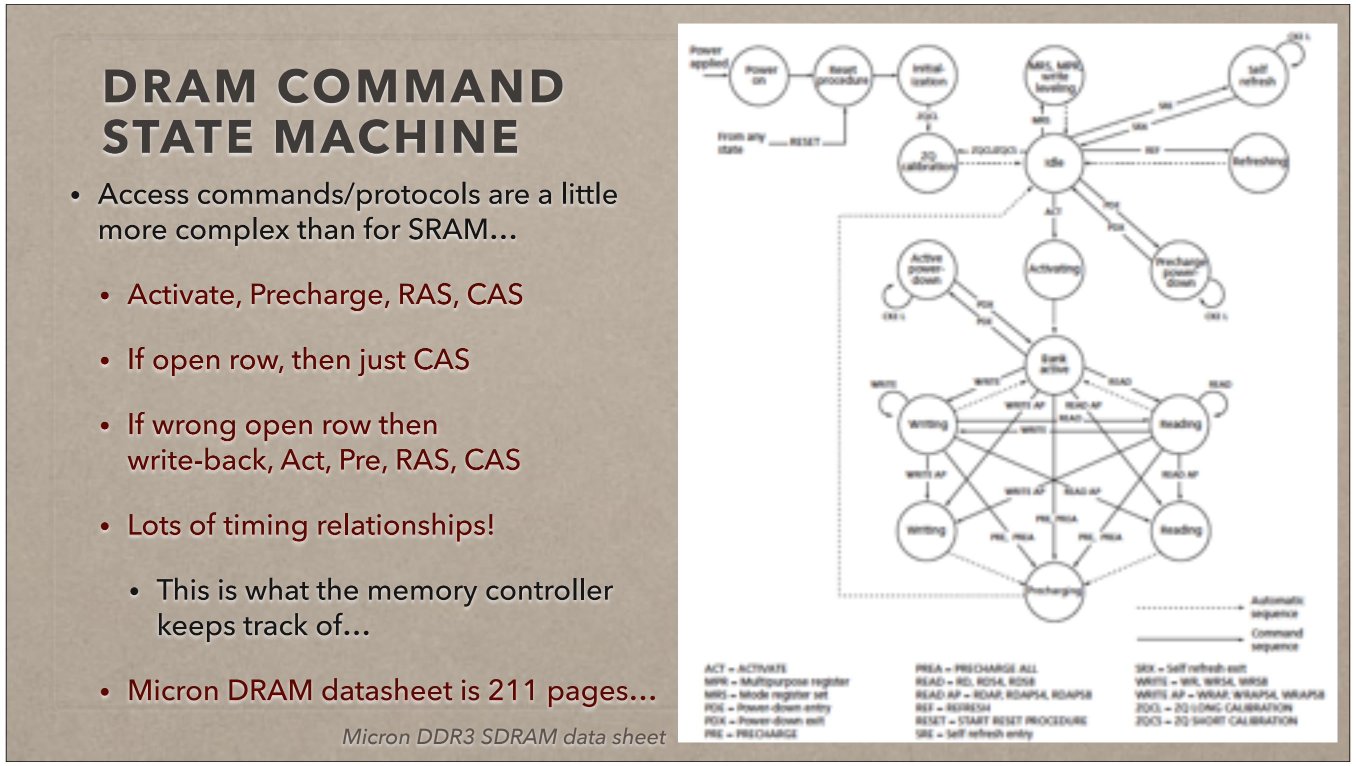
DRAM behavior at the circuit and chip level is the fundamental source of much of the complexity, and the speed and power overheads, of this type of memory. I'm only going to scratch the surface of exactly how these chips work, but we think it's important to know at least a little about *why* these chips are so slow and complex.



DRAM INTERNAL MAT ORGANIZATION

These are not particularly modern DRAM chips, but they do show the mat-based structure on the chips

- Each little square is a mat
- The data is routed around the chip on wires that are in-between the mats.
- The shiny bumps (see right picture) are the bonding pads that connect the chip to the package. They’re actually in the center of the chip for this die. This chips has four large sections, each with 4x16 mats. There’s routing/addressing, and other circuity in the center.



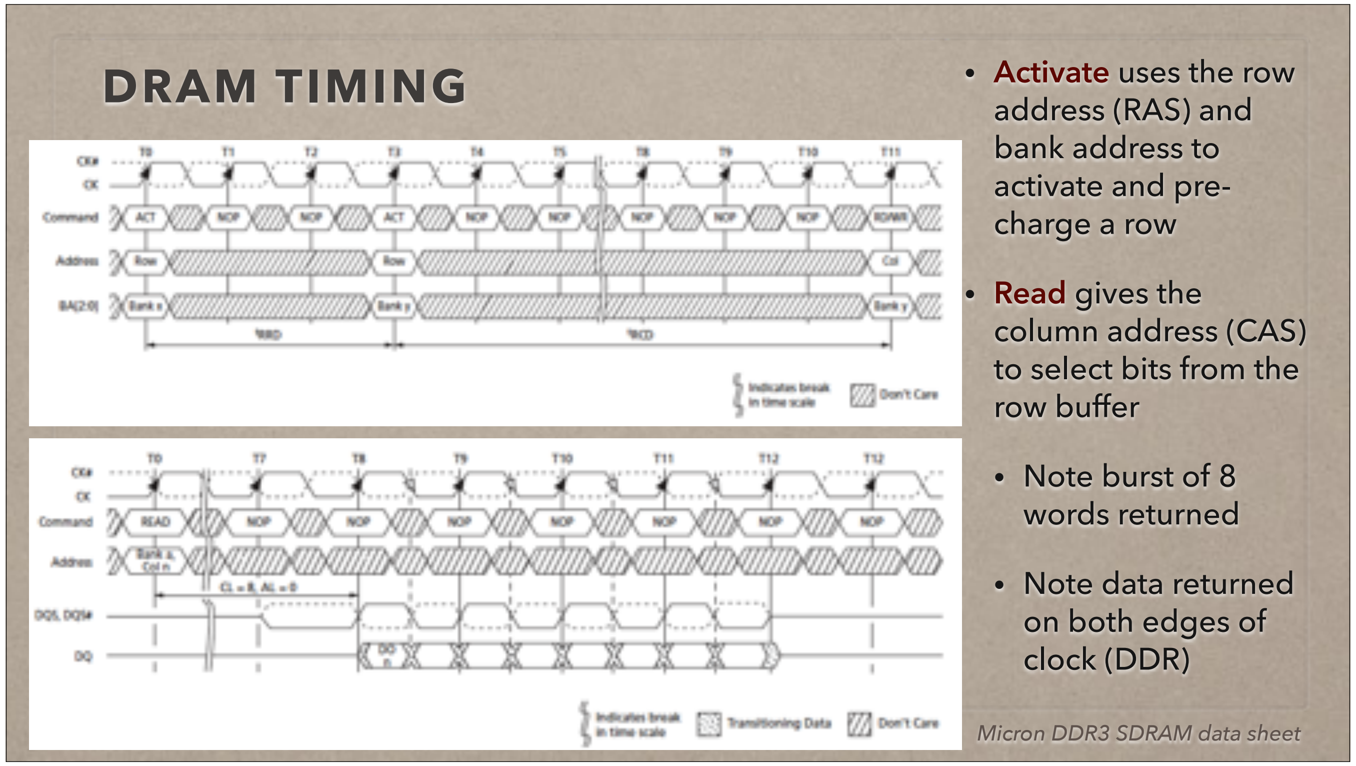
DRAM COMMAND STATE MACHINE

- Access commands/protocols are a little more complex than for SRAM…
  - Activate, Precharge, RAS, CAS
  - If open row, then just CAS
  - If wrong open row then write-back, Act, Pre, RAS, CAS
  - Lots of timing relationships!
    - This is what the memory controller keeps track of…
  - Micron DRAM datasheet is 211 pages…

Micron DDR3 SDRAM data sheet

In contrast to SRAM, getting data into and out-of a DRAM is much more complex!

- There’s a multi-state state machine that defines the current state of the chip
  - The states involve pre-charging arrays (mats), banks, etc.
  - Addresses are always sent in two pieces: row address and column address sent separately on the same address lines.
  - Lots of control signals!
  - Lots of timing signals!
  - Lots of behaviors



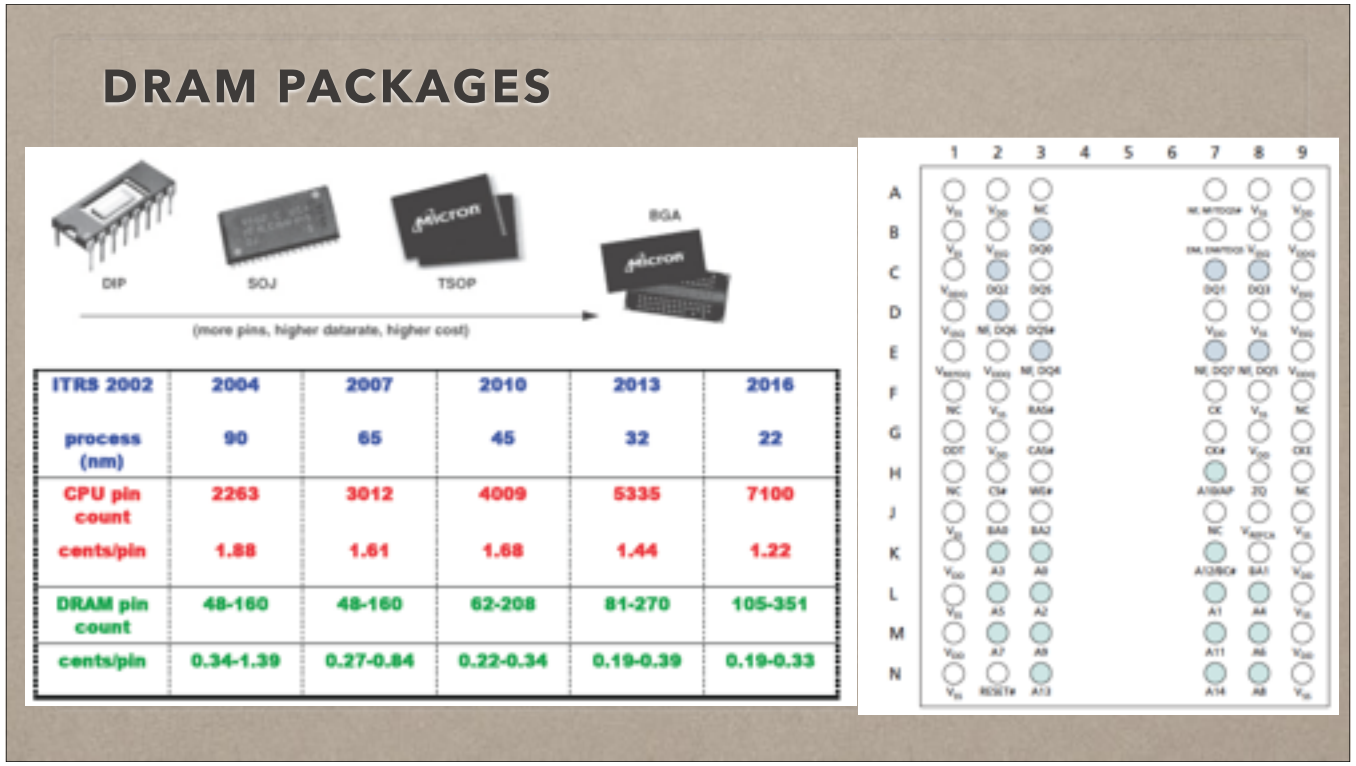
DRAM TIMING

- Activate uses the row address (RAS) and bank address to activate and pre-charge a row
- Read gives the column address (CAS) to select bits from the row buffer
  - Note burst of 8 words returned
  - Note data returned on both edges of clock (DDR)

Micron DDR3 SDRAM data sheet

The (extremely basic) access process is:

- Send the row address, and RAS to establish a bank address and pre-charge the mats in a bank.
  - This reads one row of data (destructively) into the Row Buffer
- Send the column address and CAS to select some bits from the row buffer to be delivered to the output pins.
- Typically the read command (sent with the CAS) returns a sequential burst of data - 8 words in 8 successive clock cycles for example.
  - The data return is separated from the CAS by some delay
  - You can queue up (pipeline) other requests during that pipeline delay.



DRAM PACKAGES

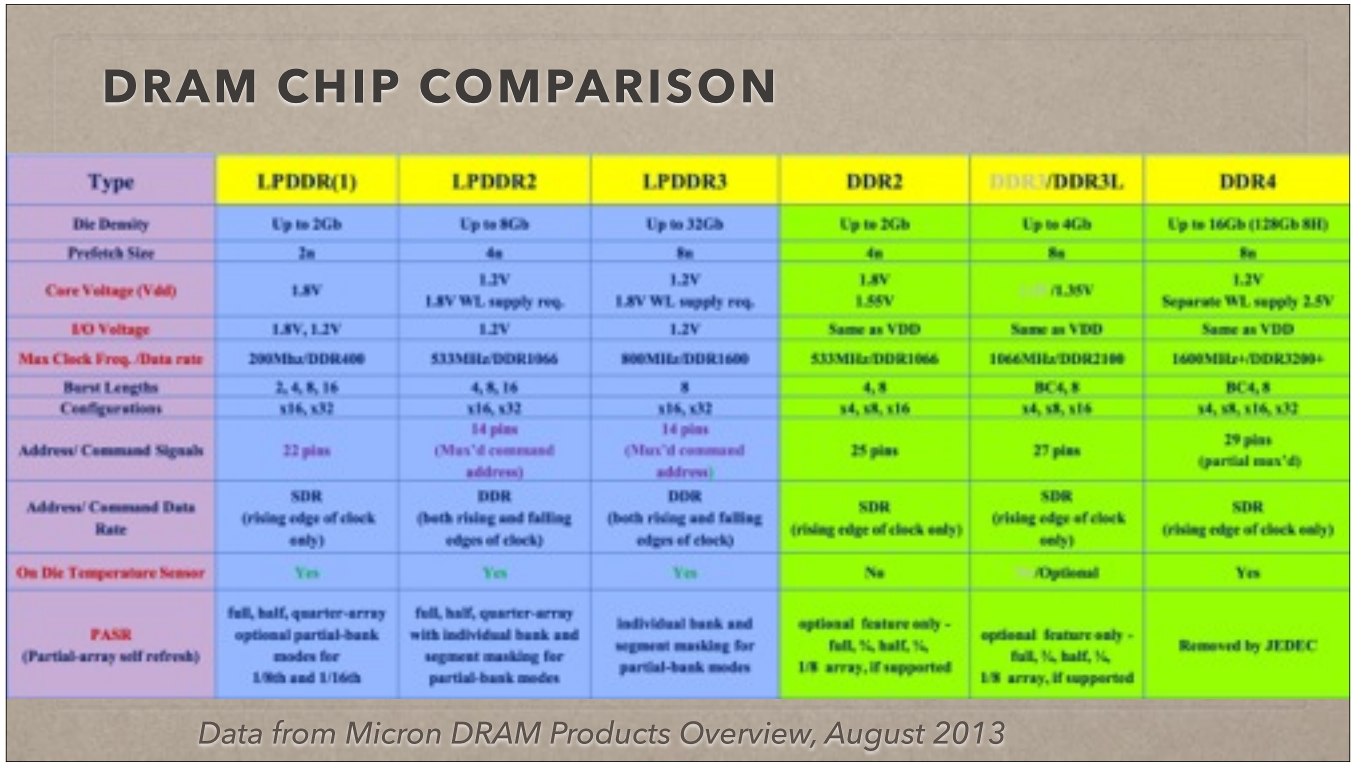
ITRS 2002	2004	2007	2010	2013	2016
process (nm)	90	65	45	32	22
CPU pin count	2263	3012	4009	5335	7100
cents/pin	1.88	1.61	1.68	1.44	1.22
DRAM pin count	48-160	48-160	62-208	81-270	105-351
cents/pin	0.34-1.39	0.27-0.84	0.22-0.34	0.19-0.39	0.19-0.33

	1	2	3	4	5	6	7	8	9
A	V_{SS}	V_{DD}	NC				NF, NF/TDQS#	V_{SS}	V_{DD}
B	V_{SS}	V_{SSQ}	DQ0				DM, DM/TDQS	V_{SSQ}	V_{DDQ}
C	V_{DDQ}	DQ2	DQS				DQ1	DQ3	V_{SSQ}
D	V_{SSQ}	NF, DQ6	DQS#				V_{DD}	V_{SS}	V_{SSQ}
E	V_{REFDQ}	V_{DDQ}	NF, DQ4				NF, DQ7	NF, DQ5	V_{DDQ}
F	NC	V_{SS}	RAS#				CK	V_{SS}	NC
G	ODT	V_{DD}	CAS#				CK#	V_{DD}	CKE
H	NC	CS#	WE#				A10/AP	ZQ	NC
J	V_{SS}	BA0	BA2				NC	V_{REFCA}	V_{SS}
K	V_{DD}	A3	A0				A12/BC#	BA1	V_{DD}
L	V_{SS}	A5	A2				A1	A4	V_{SS}
M	V_{DD}	A7	A9				A11	A6	V_{DD}
N	V_{SS}	RESET#	A13				A14	A8	V_{SS}

Part of the reason for the underlying access style (split row and column addressing) is because of how the internal mats work.

The other reason is to minimize the number of pins on the package. Pins are expensive! Splitting the address in this way cuts the number of address pins in half.

Notice how many control pins there are though…

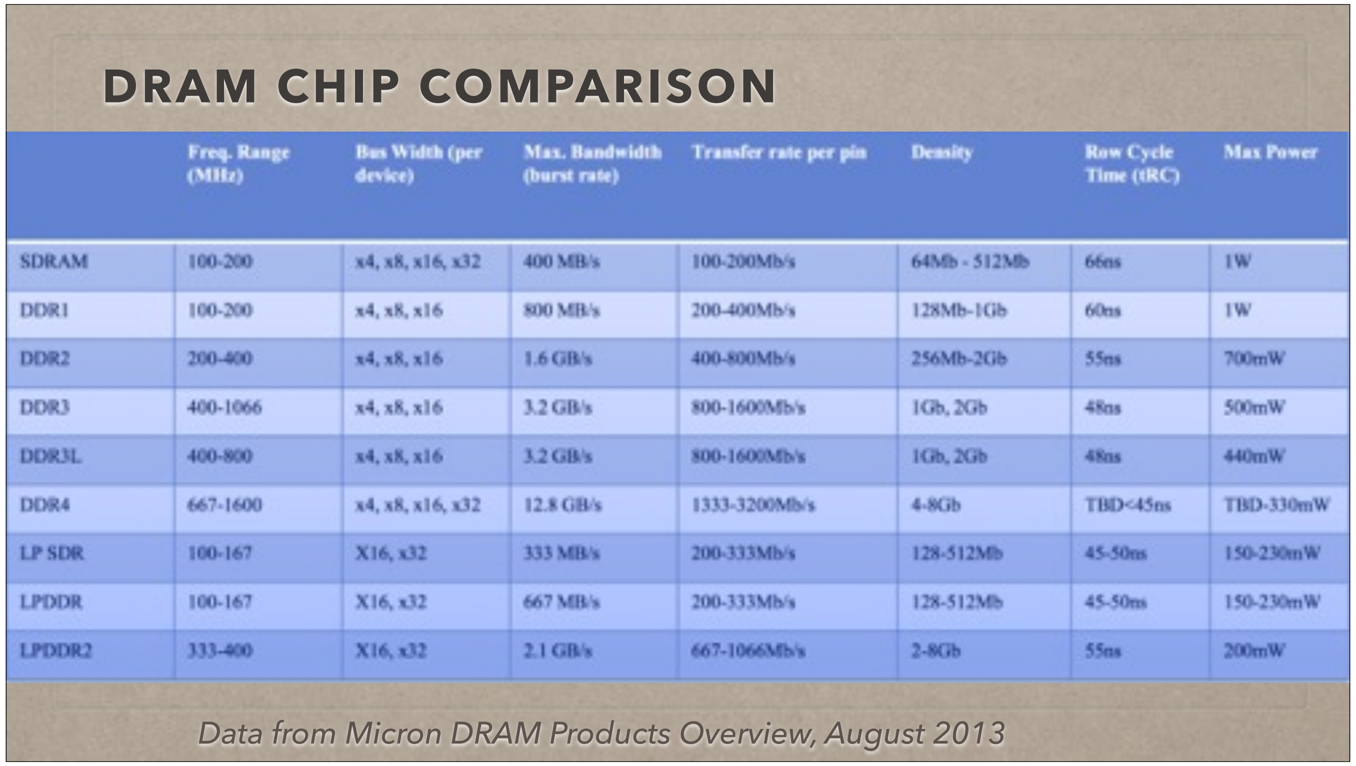


DRAM CHIP COMPARISON

Type	LPDDR(1)	LPDDR2	LPDDR3	DDR2	DDR3/DDR3L	DDR4
Die Density	Up to 2Gb	Up to 8Gb	Up to 32Gb	Up to 2Gb	Up to 4Gb	Up to 16Gb (128Gb 8H)
Prefetch Size	2n	4n	8n	4n	8n	8n
Core Voltage (Vdd)	1.8V	1.2V 1.8V WL supply req.	1.2V 1.8V WL supply req.	1.8V 1.55V	1.5V/1.35V	1.2V Separate WL supply 2.5V
I/O Voltage	1.8V, 1.2V	1.2V	1.2V	Same as VDD	Same as VDD	Same as VDD
Max Clock Freq. /Data rate	200Mhz/DDR400	533MHz/DDR1066	800MHz/DDR1600	533MHz/DDR1066	1066MHz/DDR2100	1600MHz+/DDR3200+
Burst Lengths	2, 4, 8, 16	4, 8, 16	8	4, 8	BC4, 8	BC4, 8
Configurations	x16, x32	x16, x32	x16, x32	x4, x8, x16	x4, x8, x16	x4, x8, x16, x32
Address/ Command Signals	22 pins	14 pins (Mux'd command address)	14 pins (Mux'd command address)	25 pins	27 pins	29 pins (partial mux'd)
Address/ Command Data Rate	SDR (rising edge of clock only)	DDR (both rising and falling edges of clock)	DDR (both rising and falling edges of clock)	SDR (rising edge of clock only)	SDR (rising edge of clock only)	SDR (rising edge of clock only)
On Die Temperature Sensor	Yes	Yes	Yes	No	No/Optional	Yes
PASR (Partial-array self refresh)	full, half, quarter-array optional partial-bank modes for 1/8th and 1/16th	full, half, quarter-array with individual bank and segment masking for partial-bank modes	individual bank and segment masking for partial-bank modes	optional feature only - full, ¾, half, ¼, 1/8 array, if supported	optional feature only - full, ¾, half, ¼, 1/8 array, if supported	Removed by JEDEC

Data from Micron DRAM Products Overview, August 2013

Here are some specs from Micron that can help explain the differences between desktop DRAM (DDR2, DDR3L, DDR4) and “low power” DRAM (LPDDR, LPDDR2, and LPDDR3).

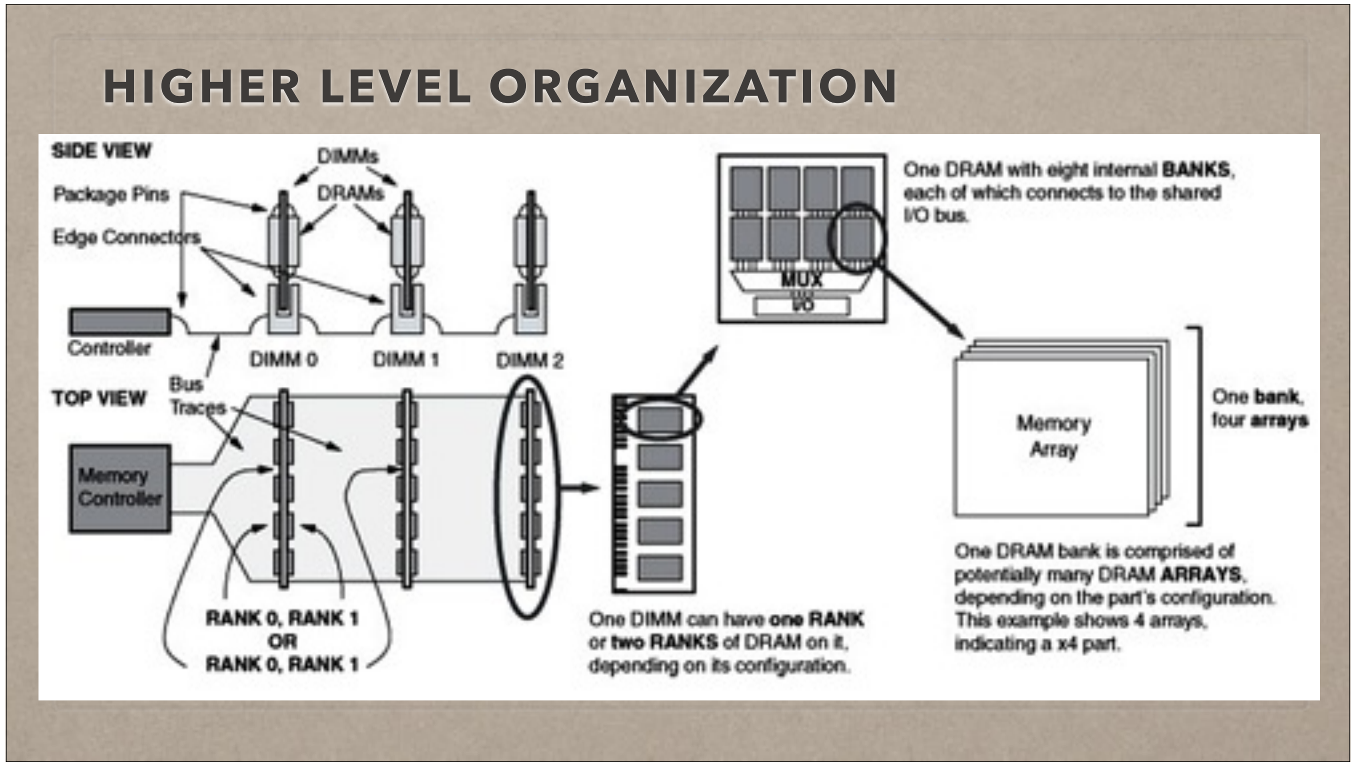


DRAM CHIP COMPARISON

	Freq. Range (MHz)	Bus Width (per device)	Max. Bandwidth (burst rate)	Transfer rate per pin	Density	Row Cycle Time (tRC)	Max Power
SDRAM	100-200	x4, x8, x16, x32	400 MB/s	100-200Mb/s	64Mb - 512Mb	66ns	1W
DDR1	100-200	x4, x8, x16	800 MB/s	200-400Mb/s	128Mb-1Gb	60ns	1W
DDR2	200-400	x4, x8, x16	1.6 GB/s	400-800Mb/s	256Mb-2Gb	55ns	700mW
DDR3	400-1066	x4, x8, x16	3.2 GB/s	800-1600Mb/s	1Gb, 2Gb	48ns	500mW
DDR3L	400-800	x4, x8, x16	3.2 GB/s	800-1600Mb/s	1Gb, 2Gb	48ns	440mW
DDR4	667-1600	x4, x8, x16, x32	12.8 GB/s	1333-3200Mb/s	4-8Gb	TBD<45ns	TBD-330mW
LP SDR	100-167	X16, x32	333 MB/s	200-333Mb/s	128-512Mb	45-50ns	150-230mW
LPDDR	100-167	X16, x32	667 MB/s	200-333Mb/s	128-512Mb	45-50ns	150-230mW
LPDDR2	333-400	X16, x32	2.1 GB/s	667-1066Mb/s	2-8Gb	55ns	200mW

Data from Micron DRAM Products Overview, August 2013

Here are some specs from Micron that can help explain the differences between desktop DRAM (DDR2, DDR3, DDR3L, DDR4) and "low power" DRAM (LPDDR, LPDDR2, and LPDDR3).

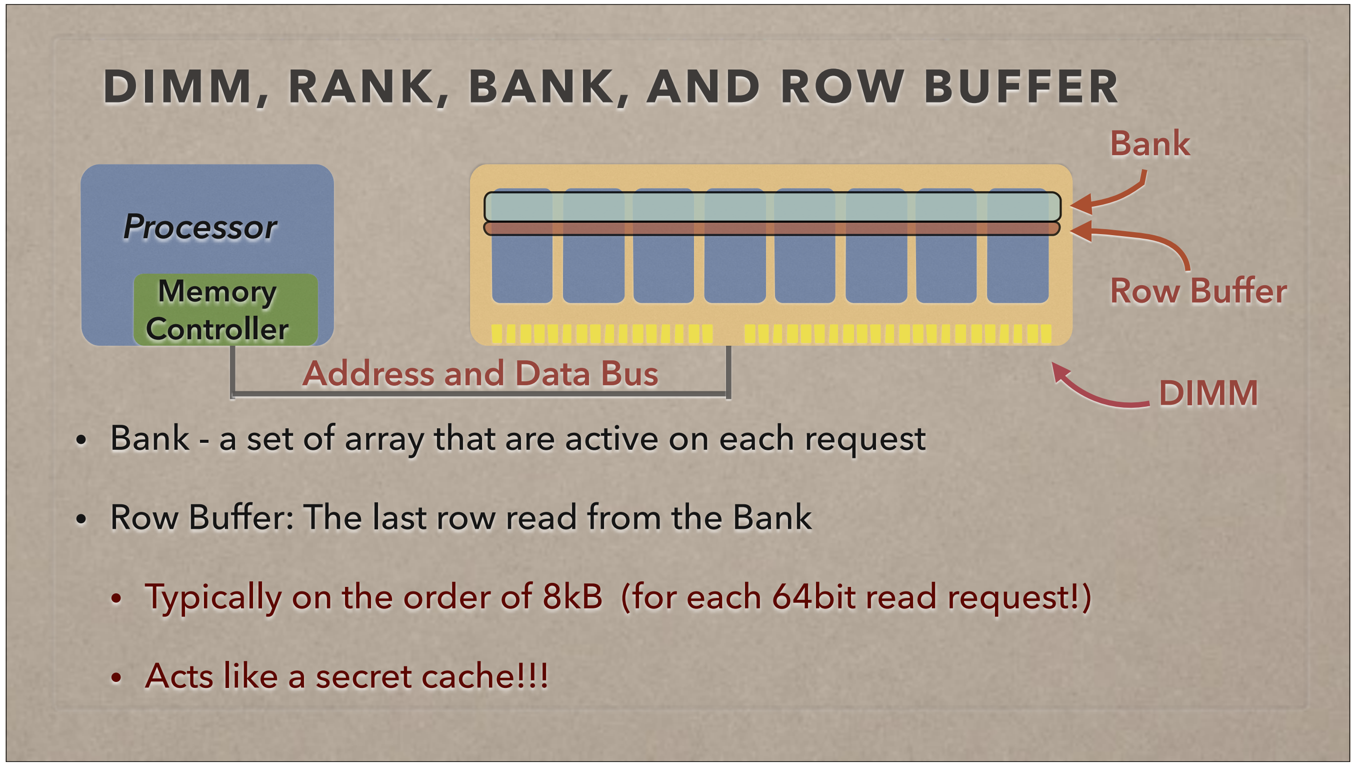


HIGHER LEVEL ORGANIZATION

SIDE VIEW
Package Pins
Edge Connectors
DIMMs
DRAMs
Controller
DIMM 0
DIMM 1
DIMM 2
TOP VIEW
Bus Traces
Memory Controller
RANK 0, RANK 1
OR
RANK 0, RANK 1
MUX
I/O
One DRAM with eight internal **BANKS**, each of which connects to the shared I/O bus.
One DIMM can have **one RANK** or **two RANKS** of DRAM on it, depending on its configuration.
Memory Array
One **bank**, four **arrays**
One DRAM bank is comprised of potentially many DRAM **ARRAYS**, depending on the part's configuration. This example shows 4 arrays, indicating a x4 part.

This is a peek ahead at Nil's section…

- The DRAM chips that I've described are assembled onto DIMMs into a hierarchy
- That hierarchy consists of DIMMs (memory sticks), Ranks (sets of chips on the DIMM), Banks (portions of the chip that can be activated for an i/o request), and mats (each Bank is made up of a set of Mats).
- This will be explained in more detail in the next section of the course.

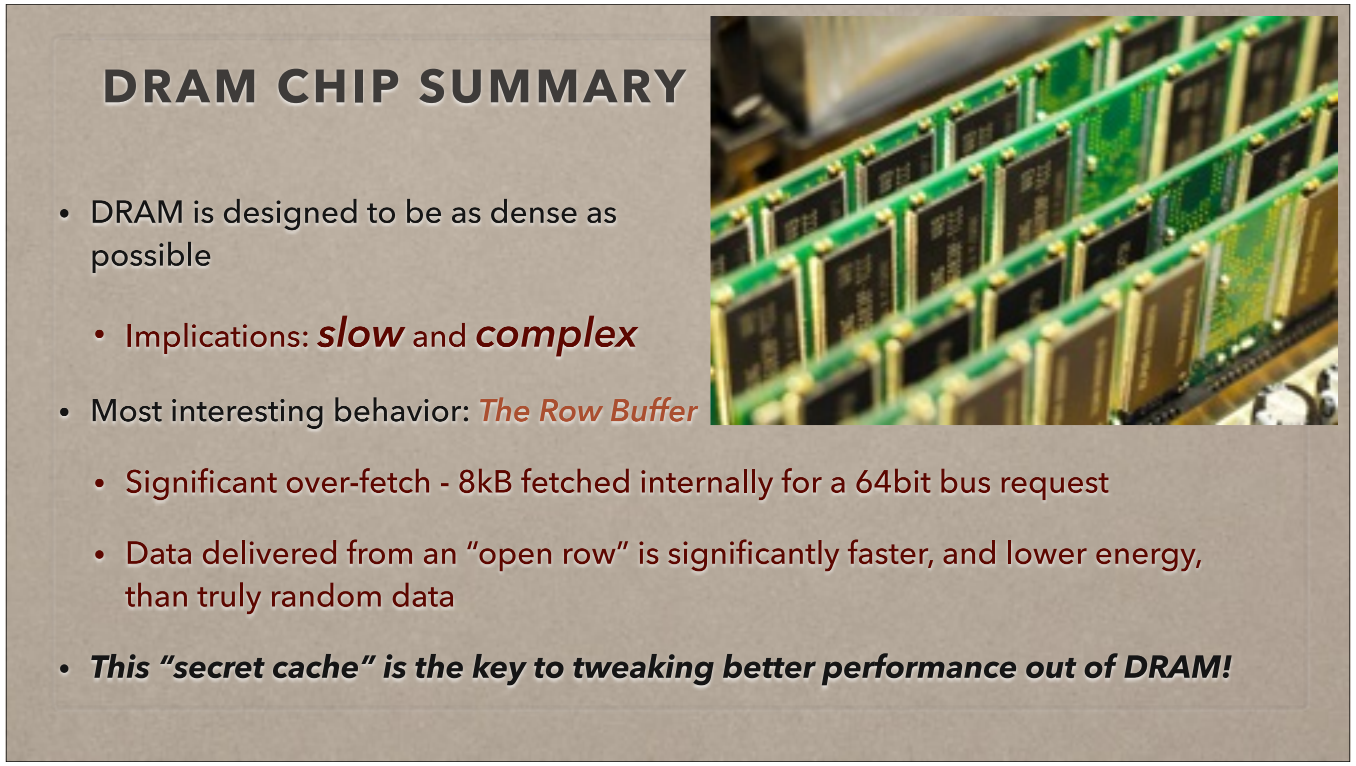


DIMM, RANK, BANK, AND ROW BUFFER

- Bank - a set of array that are active on each request
- Row Buffer: The last row read from the Bank
  - Typically on the order of 8kB (for each 64bit read request!)
  - Acts like a secret cache!!!

To close out this section, note that the Row Buffer which is activated in each active Mat, logically spans all the chips that are used in each Bank.

- This leads to fairly dramatic over-fetch behavior
- For each 64B access request, something like 8kB of data is fetched into the Row Buffer (the set of Row Buffers in each Mat).
- This large Row Buffer acts like a "secret cache!"
- But, it's not exposed to the user like a cache is.
- Once you know about it, though, you can think about ways to exploit it and get better behavior!
- This is the crux of this class - knowing something about DRAM behavior (such as the Row Buffer behavior) can lead to thinking about optimization in a whole new way.

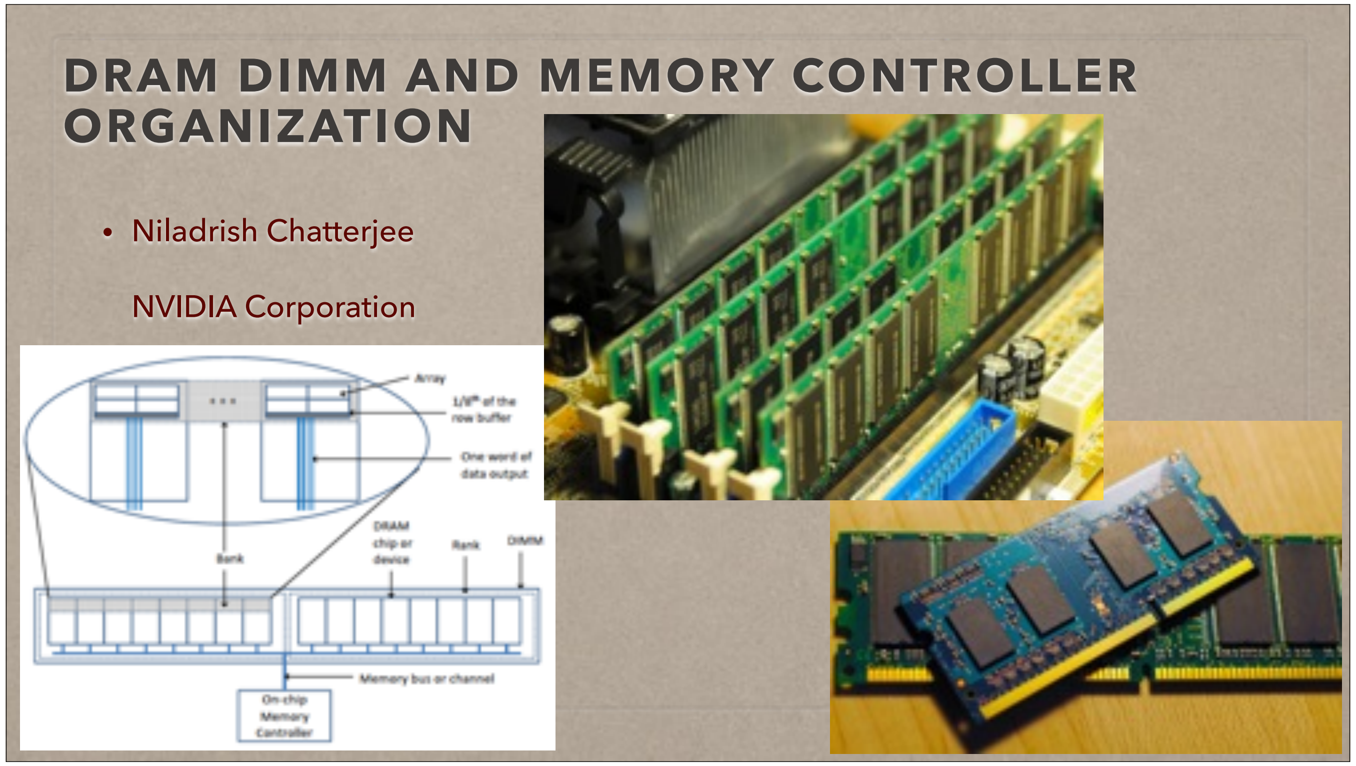


DRAM CHIP SUMMARY

- DRAM is designed to be as dense as possible
  - Implications: ***slow*** and ***complex***
- Most interesting behavior: *The Row Buffer*
  - Significant over-fetch - 8kB fetched internally for a 64bit bus request
  - Data delivered from an "open row" is significantly faster, and lower energy, than truly random data
- ***This "secret cache" is the key to tweaking better performance out of DRAM!***

To finish this section of the course:

- DRAM chips are complex and slow!
  - This is a direct consequence of trying to be dense (larger memories)
  - DRAM is all about density!
- There's at least one really interesting behavior that you can exploit
  - The Row Buffer…
  - It turns out that the latency and power of an open-row hit is SO much less than for a row buffer miss, that it can lead to dramatic improvements in system behavior
  - This is true even if the raw bandwidth to main memory goes up!



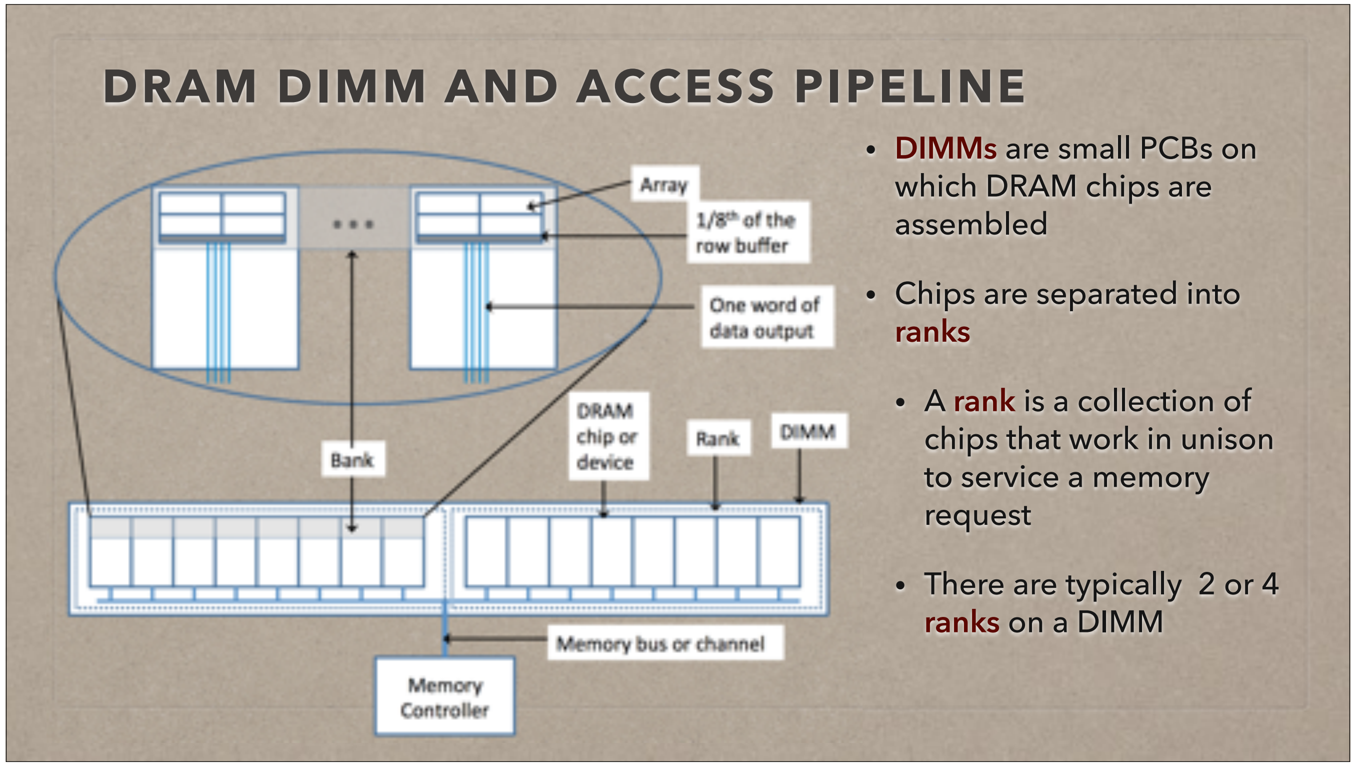
DRAM DIMM AND MEMORY CONTROLLER ORGANIZATION

- Niladrish Chatterjee

  NVIDIA Corporation

In the following few slides I will focus on

- how the memory system is built with the chips that Erik talked about
- the functions of the memory controller
- finally, some of the main challenges faced by GPU memory architects, and what solutions are on the horizon

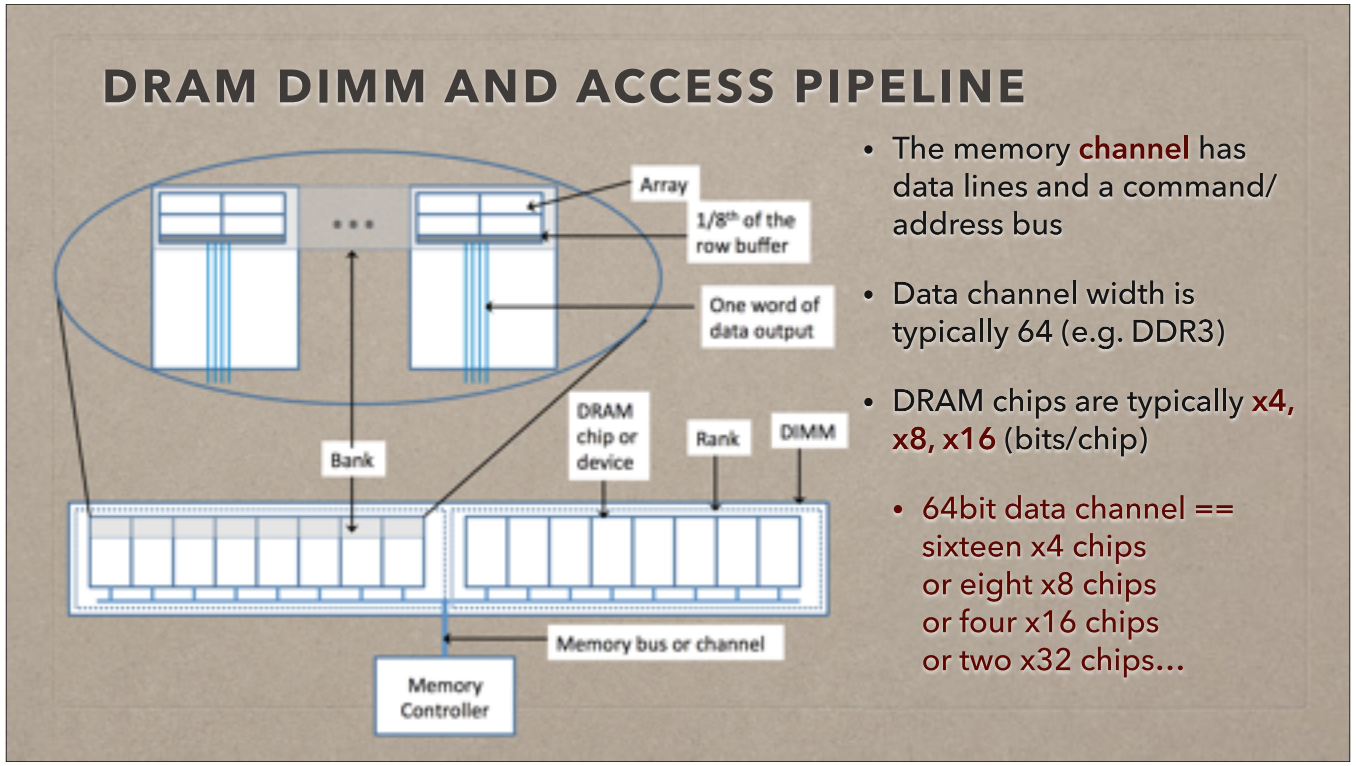


DRAM DIMM AND ACCESS PIPELINE

- DIMMs are small PCBs on which DRAM chips are assembled
- Chips are separated into ranks
  - A rank is a collection of chips that work in unison to service a memory request
  - There are typically 2 or 4 ranks on a DIMM

This figure shows a single memory channel

- Each channel has its own controller and different channels are totally independent.
- In CPU-land, DRAM chips are arranged in Dual-Inline-Memory-Modules, or DIMMs (memory sticks you buy).
- Each DIMM might have multiple ranks on it and each rank is responsible for servicing a memory request in its entirety. There are 2 to 4 ranks on a DIMM, all sharing the channel wires – and only 1 rank can transmit or receive data on it.
- Now each rank on each DIMM represents a "landing spot" or "drop" for the signal on the memory channel. It is getting increasingly hard to accommodate multiple such drops on a high-frequency channel due to signal integrity issues.
- DDR4 will allow only 1 drop at the high speeds
- So you can imagine that GPUs have run into this problem some time back – because a GDDR channel runs at a 3GHz compared to the more modest 1066MHz of the highest-end DDR3.
  - GPUs thus have 1 rank on a channel and solder the chips directly onto the GPU PCB.

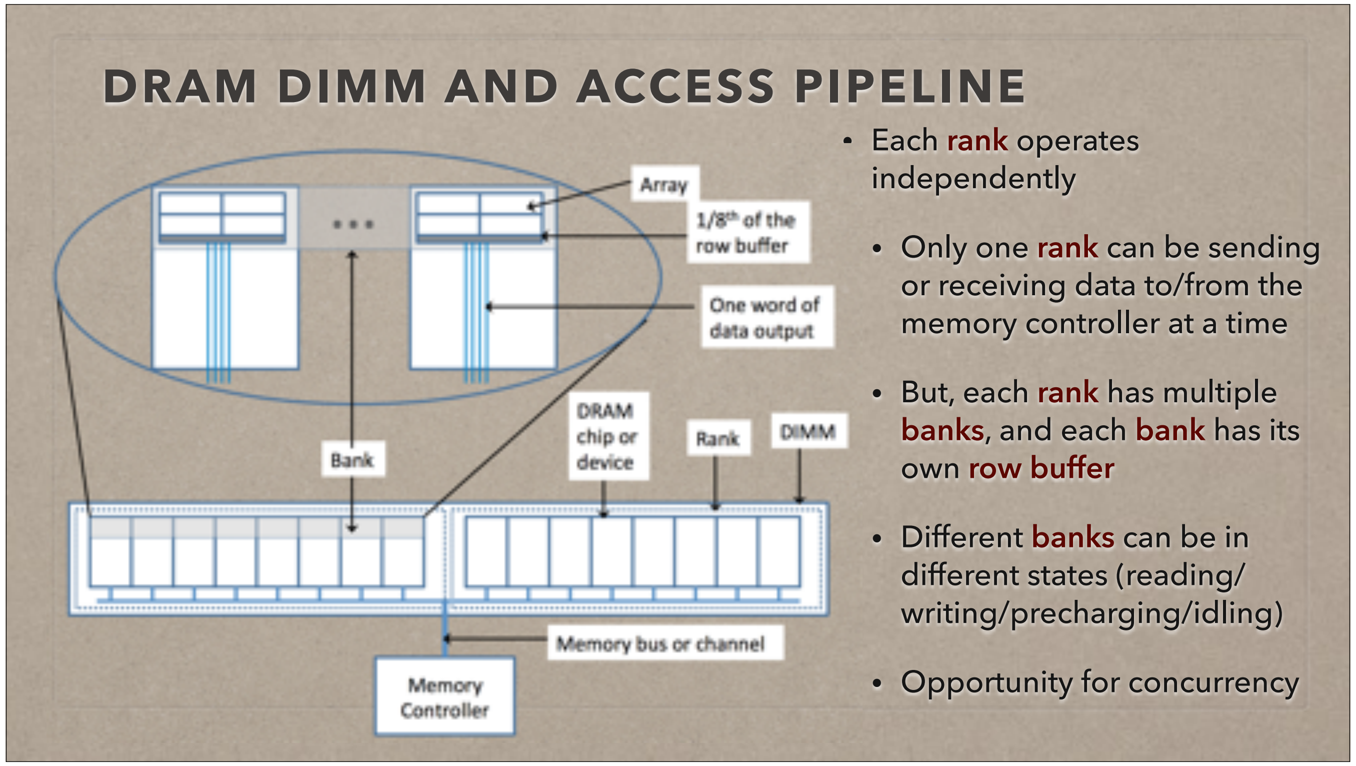


DRAM DIMM AND ACCESS PIPELINE

- The memory **channel** has data lines and a command/ address bus
- Data channel width is typically 64 (e.g. DDR3)
- DRAM chips are typically **x4, x8, x16** (bits/chip)
  - 64bit data channel == sixteen x4 chips or eight x8 chips or four x16 chips or two x32 chips…

Lets look at how a rank is formed with the DRAM chips

- each channel has data lines and a command address bus
- the JEDEC standard for DRAM sets a DRAM channel width at 64 for DDR3
  - GDDR can use 32 bit channels too
  - so a rank needs to have 65 data pins across all its chips
- Now DRAM chips can be x4,x8 or x16 (or even x32 in higher end GDDR5 and LPDDR3 chips)
- With x4 chips, you need sixteen chips for a rank, with x8 you need 8 and so on



DRAM DIMM AND ACCESS PIPELINE

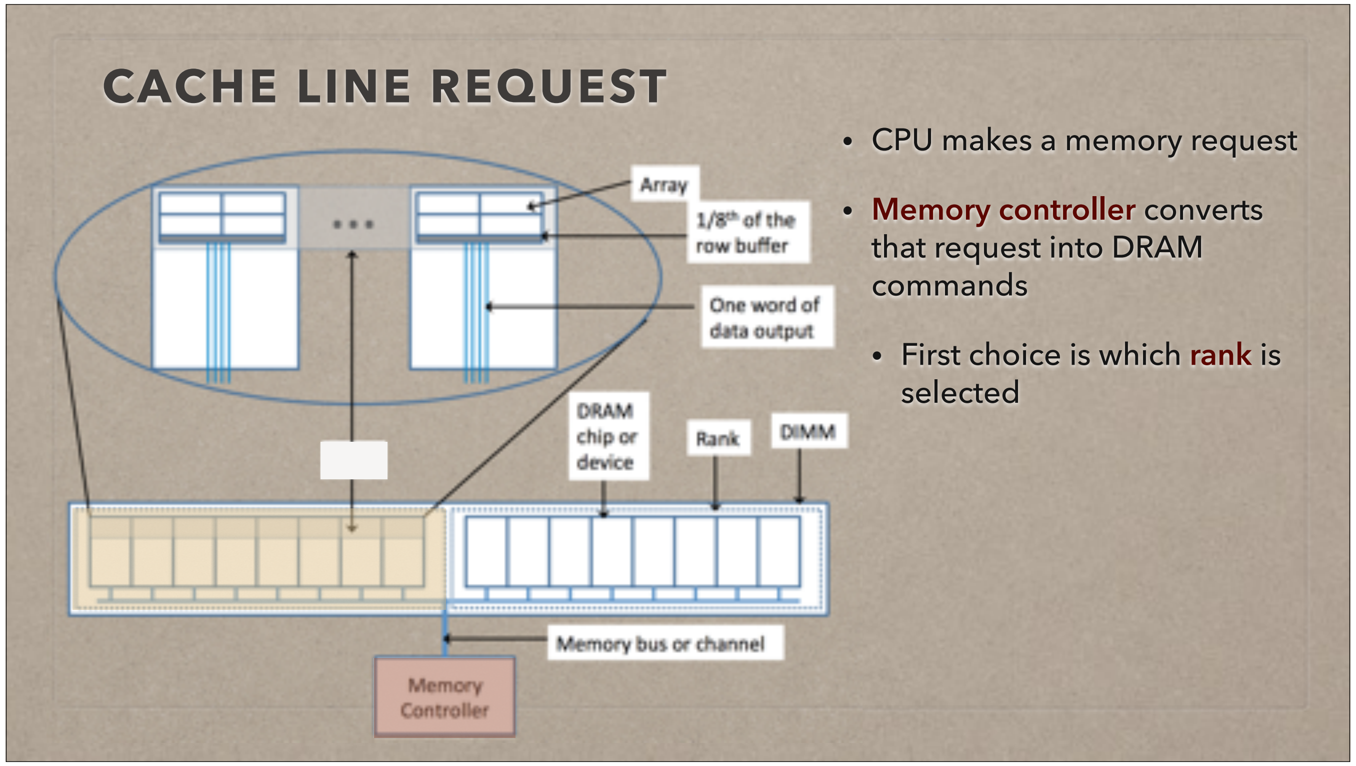
- Each **rank** operates independently
- Only one **rank** can be sending or receiving data to/from the memory controller at a time
- But, each **rank** has multiple **banks**, and each **bank** has its own **row buffer**
- Different **banks** can be in different states (reading/writing/precharging/idling)
- Opportunity for concurrency

As I said earlier, at a time only one rank can be using the data bus (note that every rank will see the address / cmd sent from the memory controller, but only the chosen one will react)

- Within a rank though, the different banks can be doing different things and be in different states (subject to some constraints)
- A rank-level bank is nothing but the same bank in all the chips of the rank (i.e. bank 1 of all chips)
- So in a rank, there can be as many row-buffers open as the number of banks with each chip holding 1/8th of the row-buffer (if these are x8 chips)

The parallel operation across banks is what allows you to hide the impact of row-misses on the bandwidth

- if there is a row-miss in a bank, you can possibly do row-hits from other banks during the row-miss to keep the bus busy.
- higher the number of banks, higher are the chances of achieving this
    - so GDDR5 has 16 banks while DDR3 has 8 (now DDR4 will have 16 as well)
- Important take-away point
    - the number of data pins available (i.e. number of channels) and the frequency of these pins determine the “PEAK” bandwidth
    - how much of that you can actually sustain, will depend on your bank count, access pattern and how smart your memory controller is. Will see this soon in more detail.



CACHE LINE REQUEST

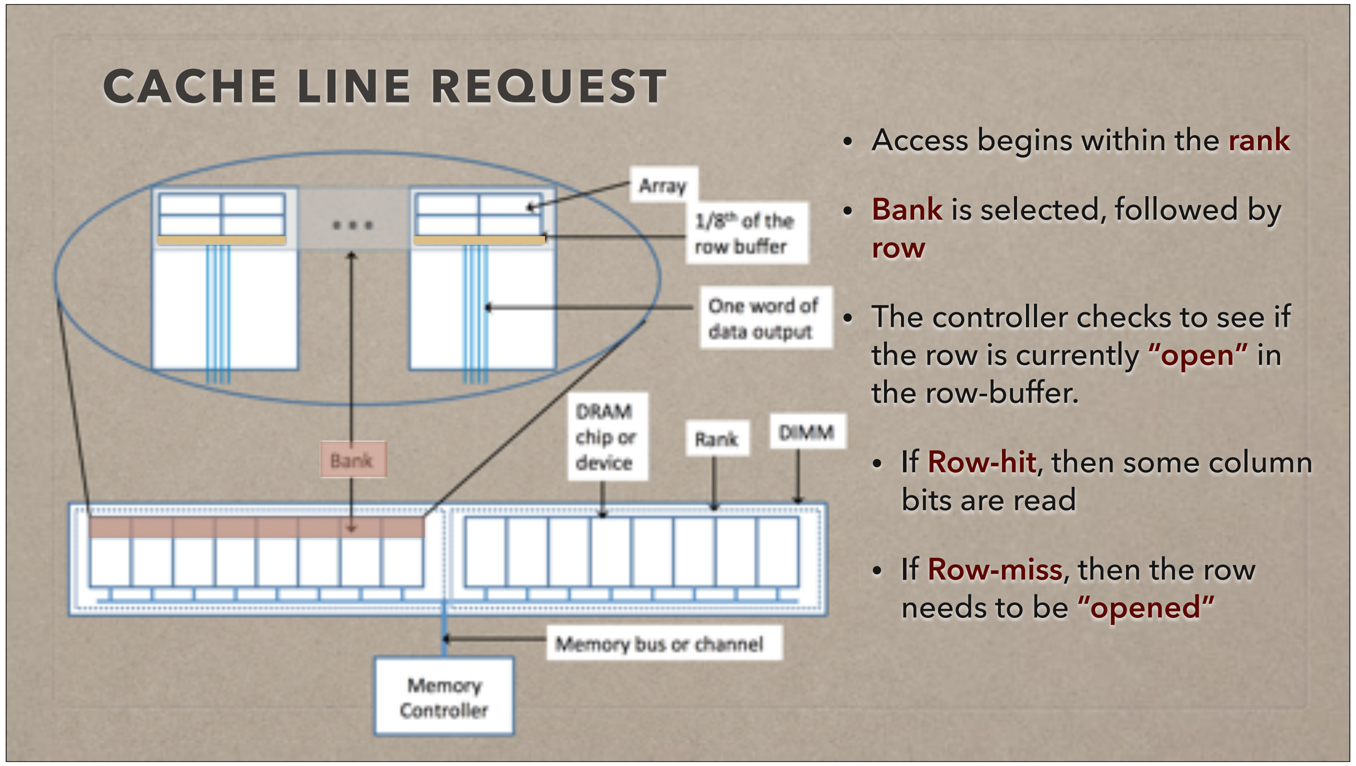
- CPU makes a memory request
- Memory controller converts that request into DRAM commands
    - First choice is which rank is selected

Let's now look at the memory access mechanism, i.e., how a cache-line is fetched from the DRAM. Typically a cache-line is the smallest granularity of access.

The memory controller decodes the address to first determine which rank to access.

- note that there is another level of address decode somewhere before the memory controller to determine which memory controller this request should be sent to

The memory controller the sends the appropriate commands to the rank to retrieve the cache-line.

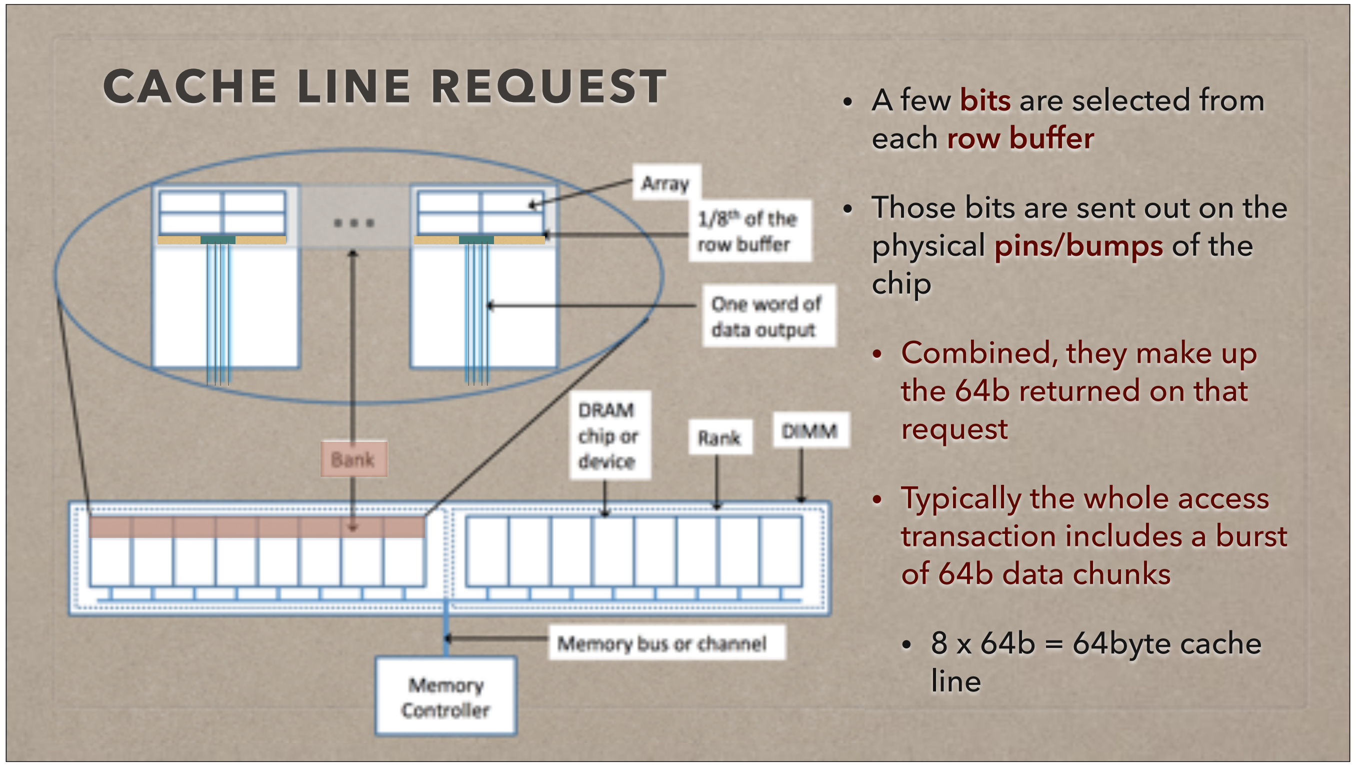


CACHE LINE REQUEST

- Access begins within the **rank**
- **Bank** is selected, followed by **row**
- The controller checks to see if the row is currently **"open"** in the row-buffer.
  - If **Row-hit**, then some column bits are read
  - If **Row-miss**, then the row needs to be **"opened"**

The memory controller knows what row-address is currently open.

- If the cache-line is present in the row, then a read command is issued with the column address
- If there is a row-miss, then the bank is first precharged, and then an activate is issued for the required row, and then finally the read is done from the row-buffer
- A row-miss increases the latency and reduces the bandwidth utilization of the bus
  - miss-rate will depend on the spatial locality in the memory request stream
  - the memory controller can reorder requests to achieve higher row-hit rates.



CACHE LINE REQUEST

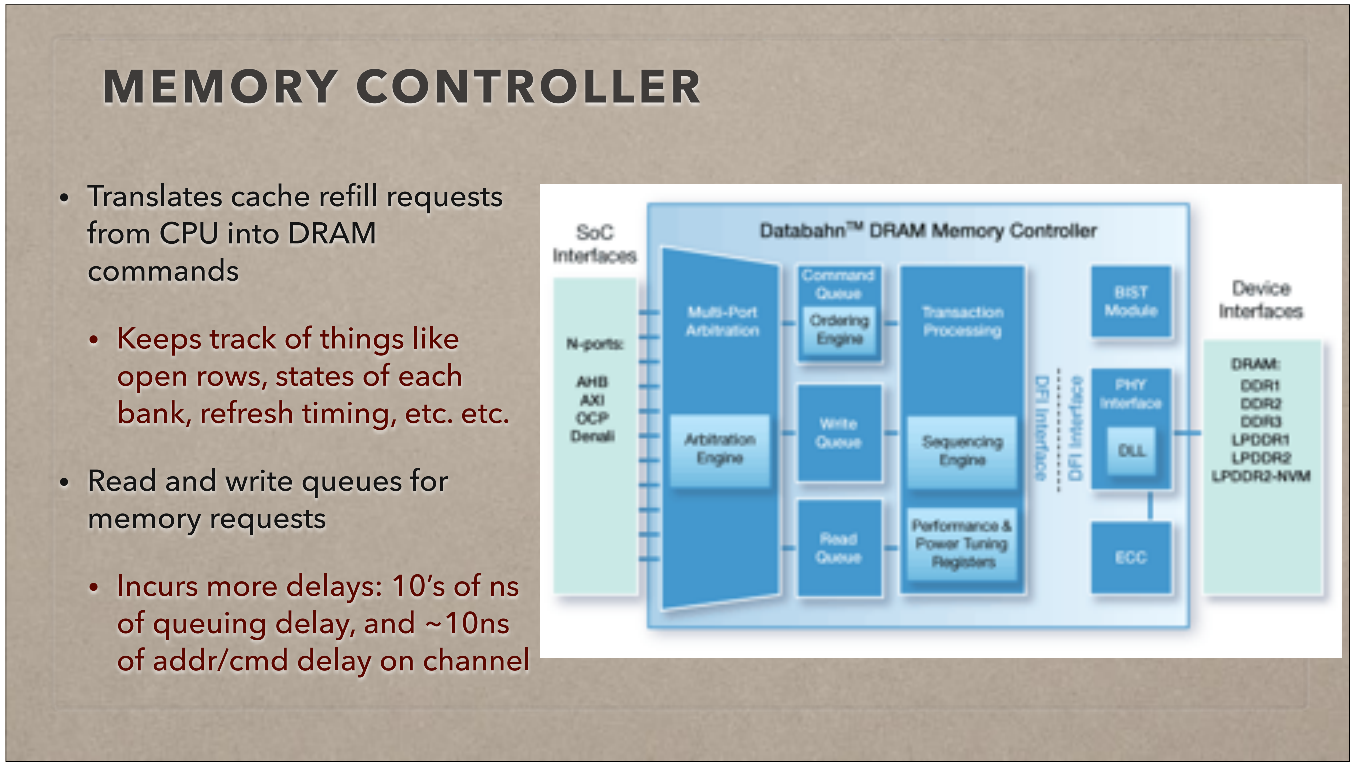
- A few **bits** are selected from each **row buffer**
- Those bits are sent out on the physical **pins/bumps** of the chip
  - Combined, they make up the 64b returned on that request
  - Typically the whole access transaction includes a burst of 64b data chunks
    - 8 x 64b = 64byte cache line

Note that when a read command is issued, a few bits are read from each chip to compose the whole cache line.

This allows using the whole pin bandwidth of the channel.

Each read requires several cycles

- 64B is transferred over 8 cycles.



MEMORY CONTROLLER

- Translates cache refill requests from CPU into DRAM commands
  - Keeps track of things like open rows, states of each bank, refresh timing, etc. etc.
- Read and write queues for memory requests
  - Incurs more delays: 10's of ns of queuing delay, and ~10ns of addr/cmd delay on channel

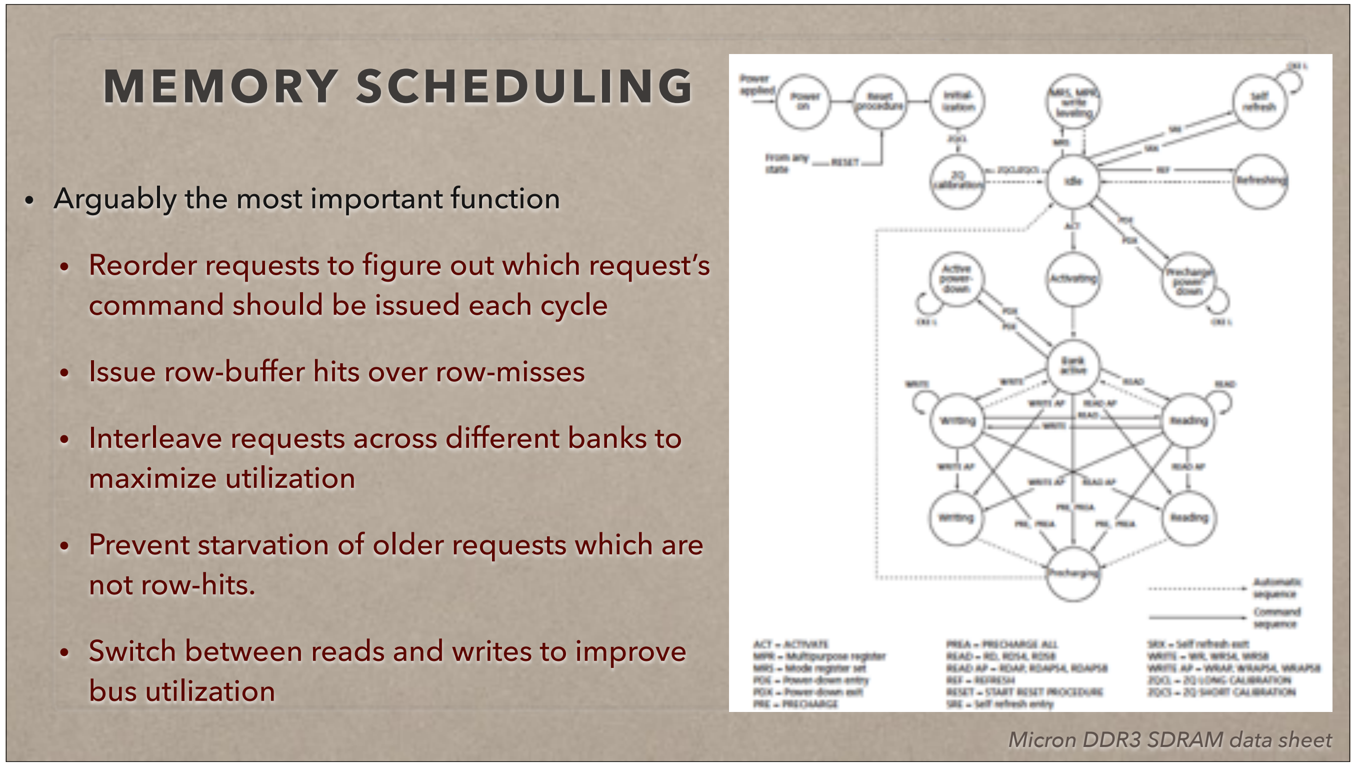
Let us now look at the memory controller

This is the "smarts" of the memory system

- also the most malleable part of the memory system.
- drastic impact on performance and energy
  - processor and gpu vendors spend large effort to outdo competitors in this space.

The main functions of the controller are

- convert memory access requests to DRAM device specific commands.
- maintain DRAM state and hide the complex details of command interaction and timings. Also mundane things like DRAM refresh to preserve data.
- schedule incoming DRAM requests to maximize performance, reduce energy.
- also maintain QoS in SoCs. The CPU, the GPU, and the peripherals have different needs.



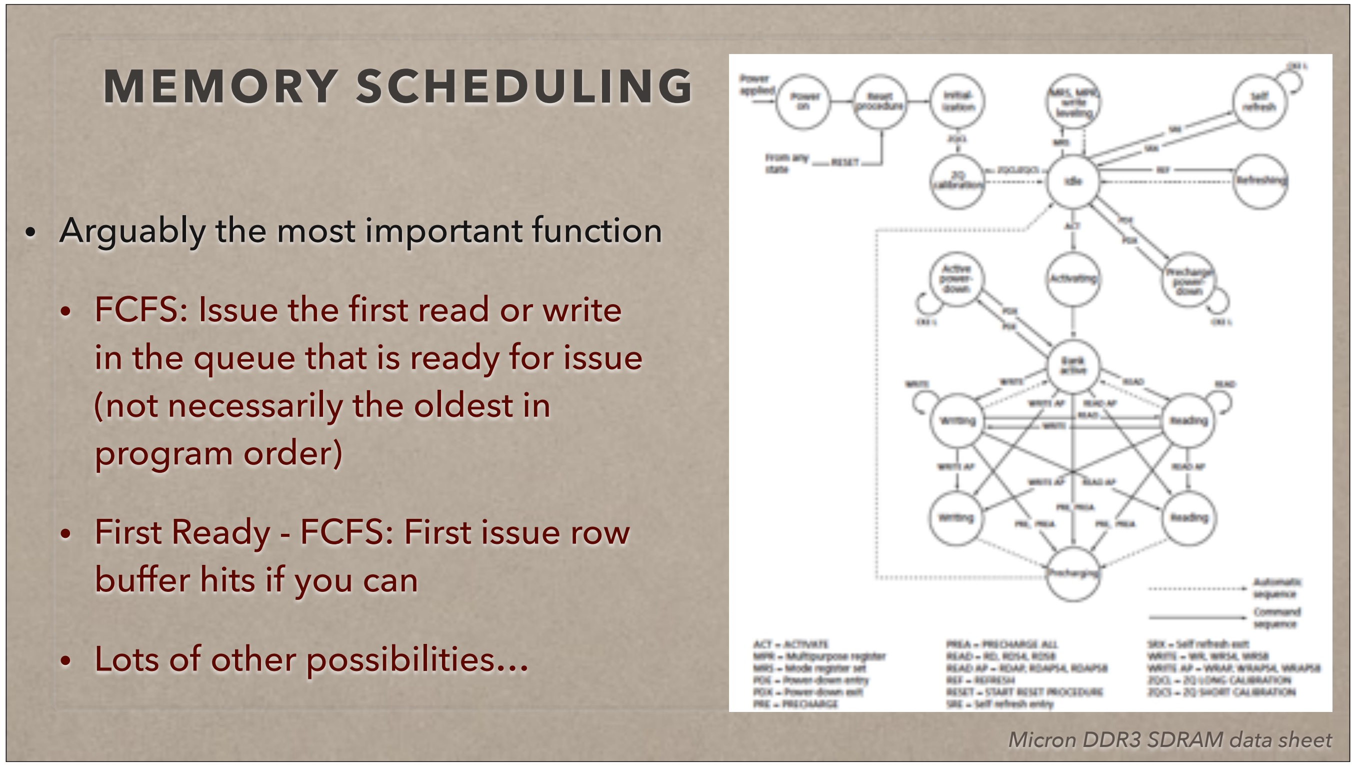
MEMORY SCHEDULING

- Arguably the most important function
  - Reorder requests to figure out which request's command should be issued each cycle
  - Issue row-buffer hits over row-misses
  - Interleave requests across different banks to maximize utilization
  - Prevent starvation of older requests which are not row-hits.
  - Switch between reads and writes to improve bus utilization

Micron DDR3 SDRAM data sheet

Memory scheduling is the most important function of the controller.

- Each cycle the controller will pick a request whose command to issue.
- Some of the common optimizations are
  - look into the request queues to group row-hits together
  - interleave requests across different banks
  - avoid starving older row-miss requests
  - also since writes are not on the critical path (since they are write-backs of dirty lines from the LLC), queue a lot of writes and keep servicing reads until the write-buffer nears full. This needs fewer bus "turnarounds" from read to write and vice versa.



MEMORY SCHEDULING

- Arguably the most important function
    - FCFS: Issue the first read or write in the queue that is ready for issue (not necessarily the oldest in program order)
    - First Ready - FCFS: First issue row buffer hits if you can
    - Lots of other possibilities…

Micron DDR3 SDRAM data sheet

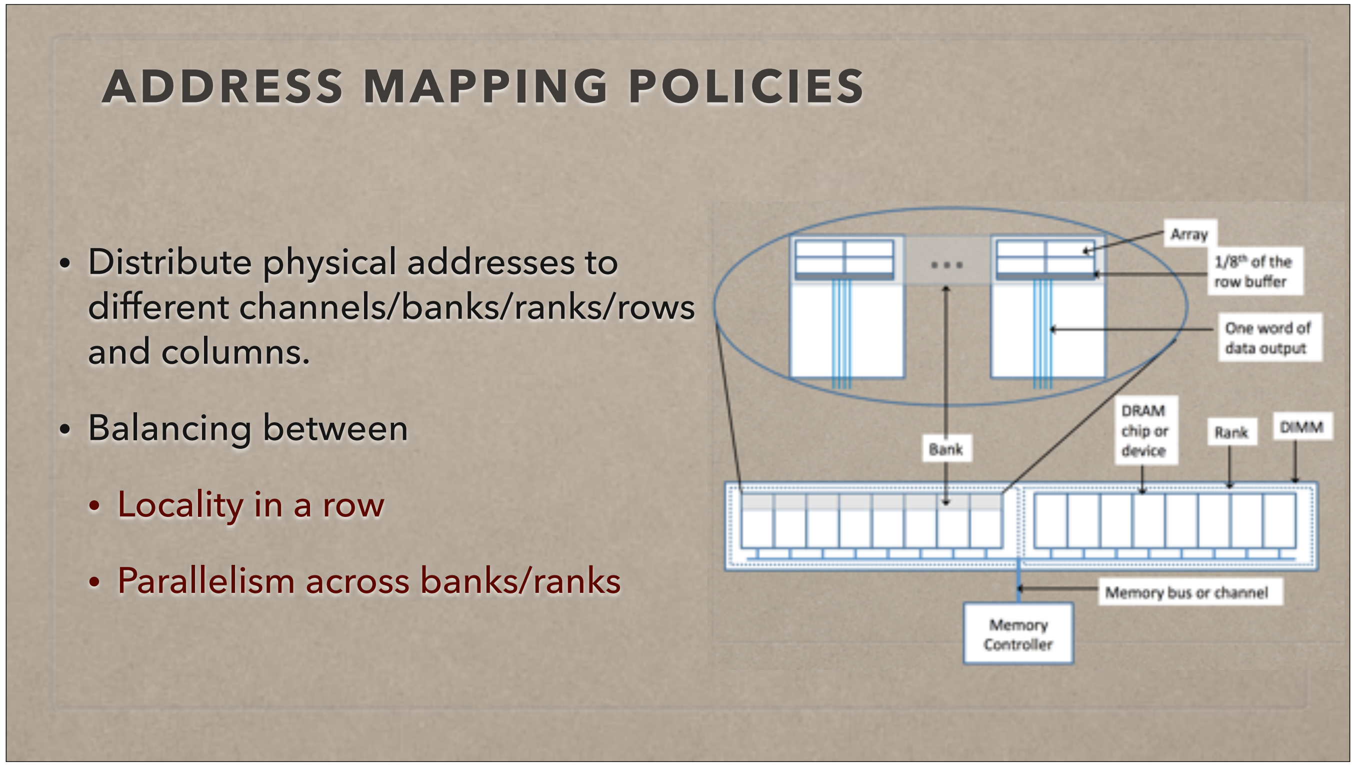
FR-FCFS is the most basic optimized scheduling algorithm and is an evolution of FCFS which is simple and very low performance for most modern systems.

However, many scheduling algorithms exist which also take into account the characteristics of the clients , not just the DRAM devices themselves

- for example, in a CPU, if there are 2 different threads running on different cores, then they might require different treatments from the memory controller depending on their memory access behavior
- a thread that has few cache misses will be latency sensitive, while one that has many cache misses will be bandwidth sensitive. The memory controller will need to reduce the queuing delay of the former and provide good bandwidth to the latter.
- same argument applies in modern SoCs, the CPU is typically latency sensitive while the GPU is hungry for more memory throughput. So the shared memory scheduler in a NVIDIA Tegra or the AMD APU or a Qualcomm Snapdragon will have to take this into consideration when arbitrating between CPU and GPU requests.

The other aspect of scheduler is power consumption

- row hits are good for low power
- but there are low-power modes available for DRAM. Controllers will use heuristics and put DRAM devices into different low power modes when the utilization is low, or when the client doesn’t care about latency too much.

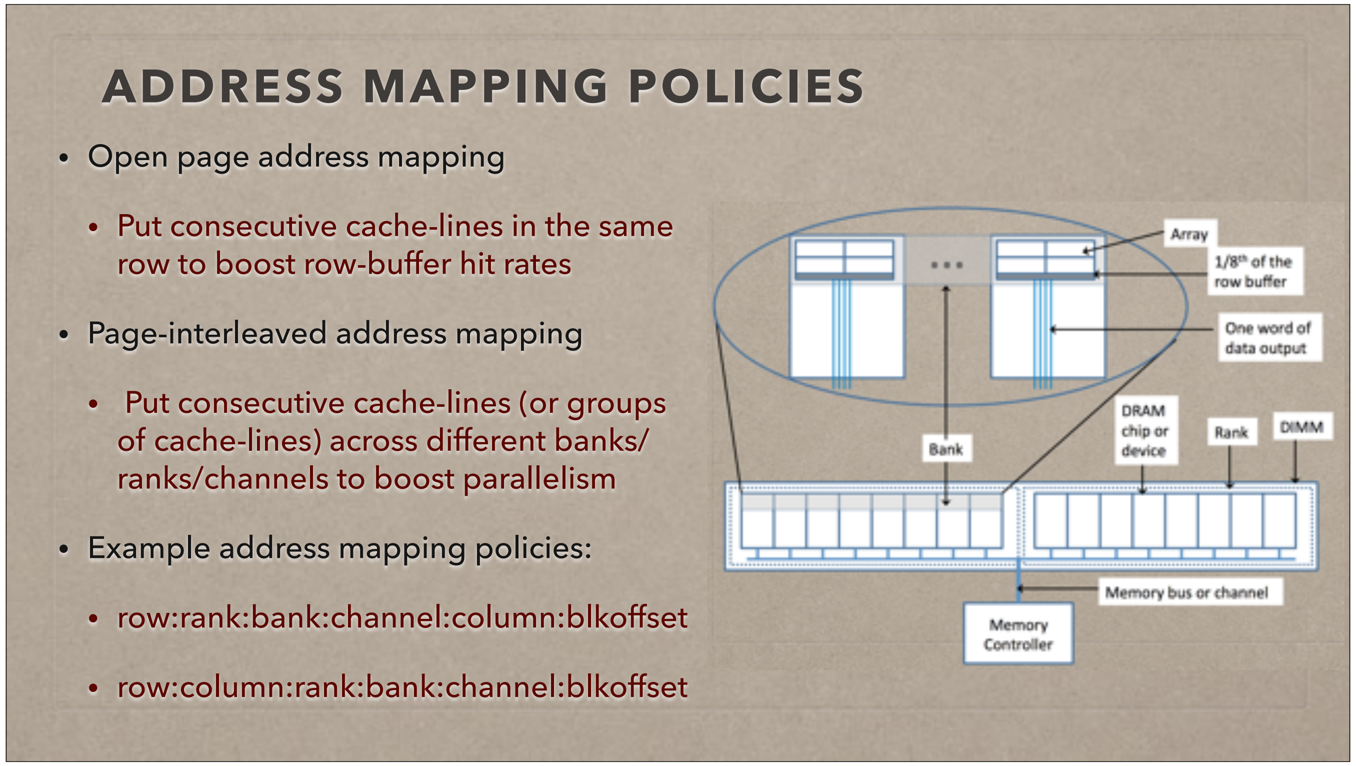


ADDRESS MAPPING POLICIES

- Distribute physical addresses to different channels/banks/ranks/rows and columns.
- Balancing between
  - Locality in a row
  - Parallelism across banks/ranks

The other important task of the memory controller is mapping a cache line address into the bank, rank, row and column address.

If consecutive cache-lines are mapped to the same row, then we can expect good row-hit rates. But also we would want to use the full pin bandwidth of the memory system, as well as improve parallelism across banks.



ADDRESS MAPPING POLICIES

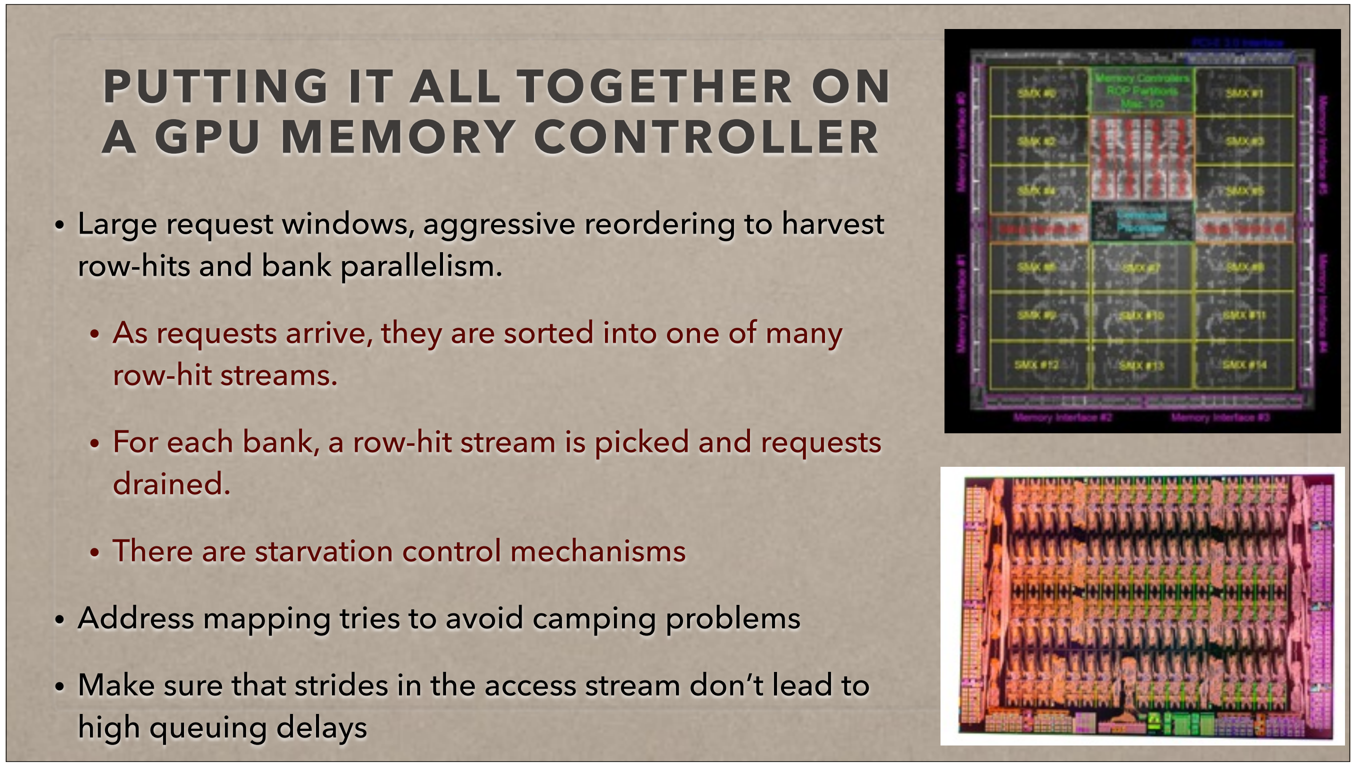
- Open page address mapping
  - Put consecutive cache-lines in the same row to boost row-buffer hit rates
- Page-interleaved address mapping
  - Put consecutive cache-lines (or groups of cache-lines) across different banks/ranks/channels to boost parallelism
- Example address mapping policies:
  - row:rank:bank:channel:column:blkoffset
  - row:column:rank:bank:channel:blkoffset

This need to balance parallelism vs row-hit rate leads to different address mappings.

- page interleaved improves parallelism
- while open page mapping increases row-hit rates
- depending on target applications, some variant of the mix of two is achieved.

During page interleaving, subtle considerations can be very helpful

- Zhao et al. MICRO-33 A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data locality.
- propose an address mapping scheme which ensures that cache-lines that cause cache-conflicts (same set in the cache), don't cause row-conflicts. This means that even when the cache sees many conflict misses, the resulting DRAM requests are row-hits.



PUTTING IT ALL TOGETHER ON A GPU MEMORY CONTROLLER

- Large request windows, aggressive reordering to harvest row-hits and bank parallelism.
  - As requests arrive, they are sorted into one of many row-hit streams.
  - For each bank, a row-hit stream is picked and requests drained.
  - There are starvation control mechanisms
- Address mapping tries to avoid camping problems
- Make sure that strides in the access stream don't lead to high queuing delays

GPUs invest a lot more in their memory controller than CPUs (a trend that is changing, CPUs have hit a steep memory wall)

The advantage that a GPU has over a CPU is that it often doesn't need to worry about latency as much – so many requests can be queued.

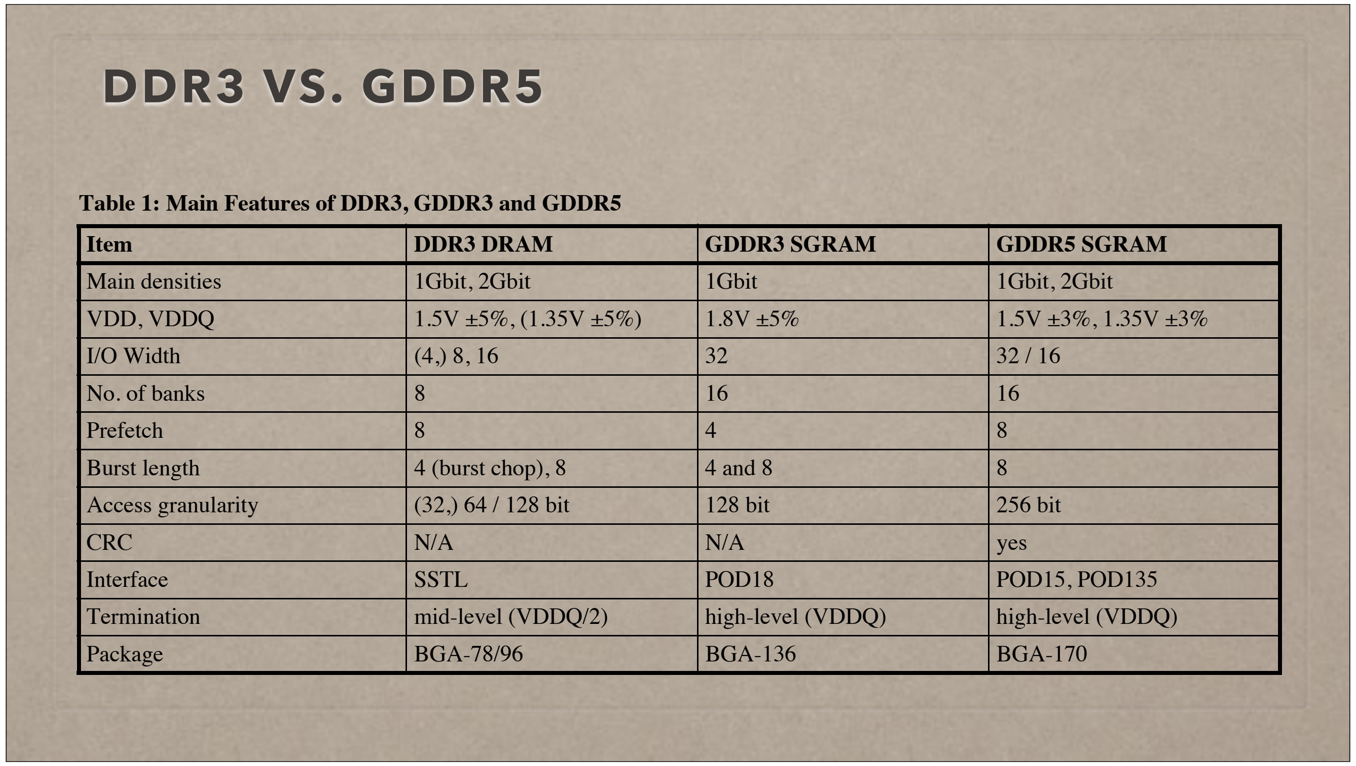
- This offers a larger window to pick the ideal next request
- In a CPU, there will be very little scope to reorder.

The GPU thus sorts the requests into row hit streams for each bank and schedules requests from a stream before moving on to the next.

There are some mechanisms to limit stalling older requests as well.

Also the address map is such that it tries to achieve the highest bandwidth utilization. If the access is strided, it may so happen that a single bank, or channel is getting overloaded – so there are safeguards against that in the address mapping scheme.

Apart from these general strategies, many more subtle, application dependent optimizations that are closely guarded secrets in each company.

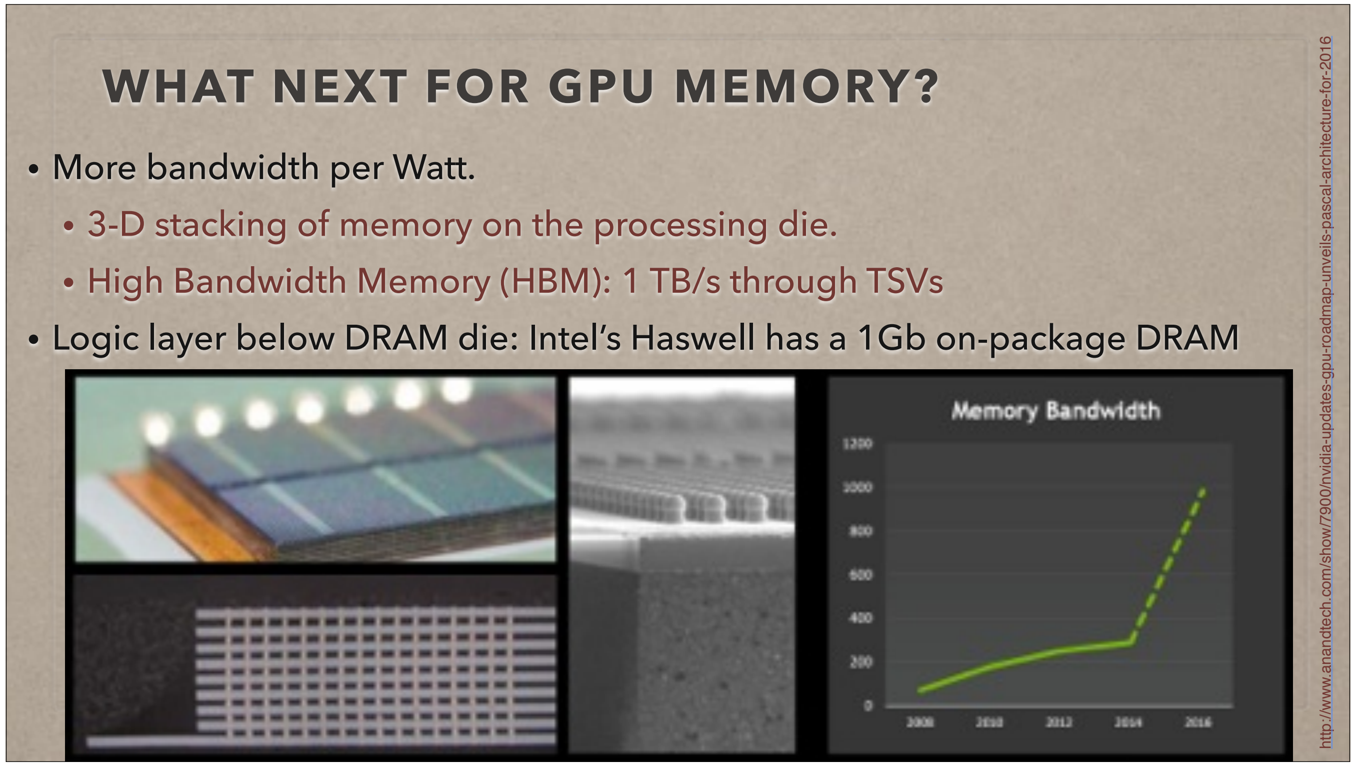


DDR3 VS. GDDR5

Table 1: Main Features of DDR3, GDDR3 and GDDR5

Item	DDR3 DRAM	GDDR3 SGRAM	GDDR5 SGRAM
Main densities	1Gbit, 2Gbit	1Gbit	1Gbit, 2Gbit
VDD, VDDQ	1.5V ±5%, (1.35V ±5%)	1.8V ±5%	1.5V ±3%, 1.35V ±3%
I/O Width	(4,) 8, 16	32	32 / 16
No. of banks	8	16	16
Prefetch	8	4	8
Burst length	4 (burst chop), 8	4 and 8	8
Access granularity	(32,) 64 / 128 bit	128 bit	256 bit
CRC	N/A	N/A	yes
Interface	SSTL	POD18	POD15, POD135
Termination	mid-level (VDDQ/2)	high-level (VDDQ)	high-level (VDDQ)
Package	BGA-78/96	BGA-136	BGA-170

Here’s a table with some of the differences in desktop DRAM (DDR3) and graphics DRAM (GDDR5).
Note that the graphics DRAM has higher chip I/O width, and more banks than typical desktop DRAM. This results in a larger access granularity. And on more pins (and thus expense) per package.



WHAT NEXT FOR GPU MEMORY?

- More bandwidth per Watt.
  - 3-D stacking of memory on the processing die.
  - High Bandwidth Memory (HBM): 1 TB/s through TSVs
- Logic layer below DRAM die: Intel’s Haswell has a 1Gb on-package DRAM

Memory Bandwidth
1200
1000
800
600
400
200
0
2008
2010
2012
2014
2016

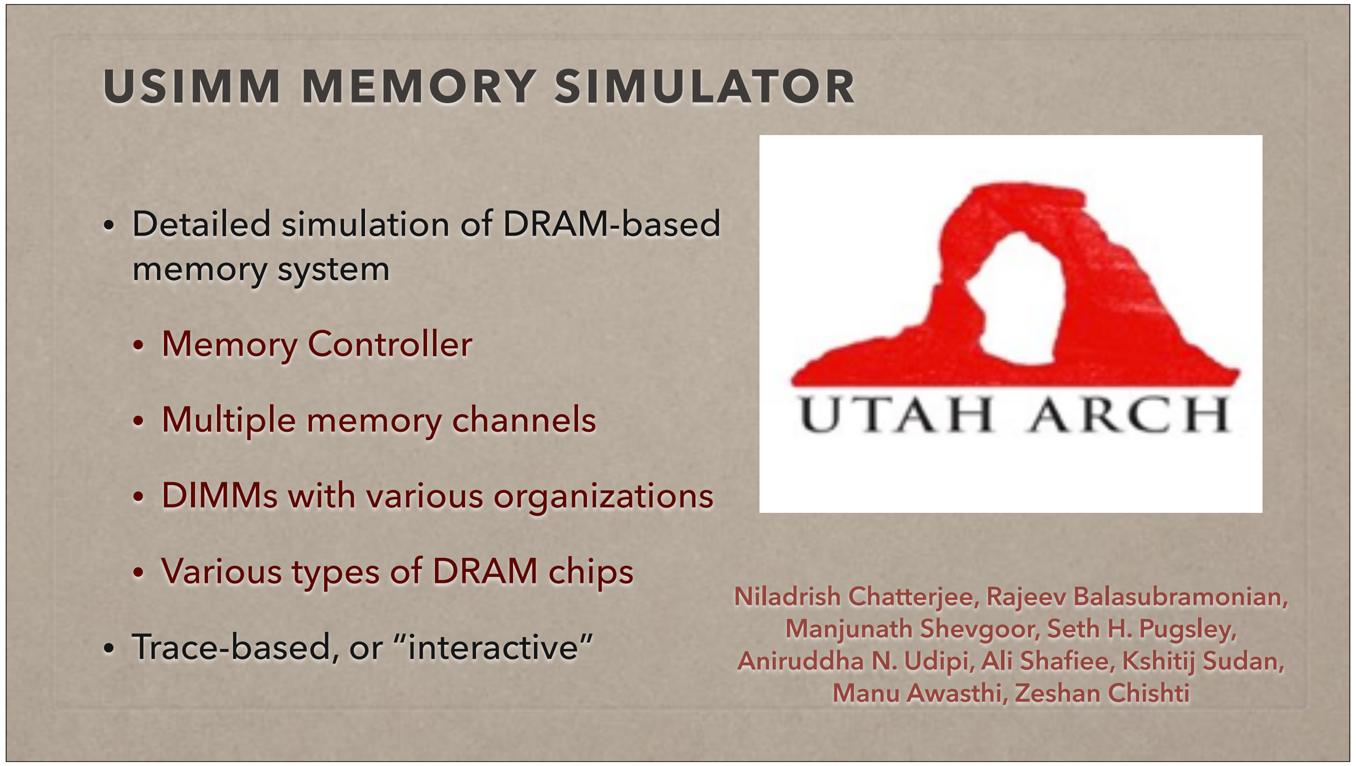
http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016

Higher BW/Watt requirements has led to the High Bandwidth Memory (HBM) standard.

- DRAM dies stacked on the processor die and connected through TSVs to give 1 TBps bandwidth. The figure is from the NVIDIA CEOs talk at GTC this year in California.
- ideal for graphics
- Intel’s IRIS PRO graphics has a 128MB eDRAM which is used as a cache
- HBM not necessarily a cache.

If this is part of programmer visible system memory that also has other slower, but more power efficient DRAM variants, then program hints about data placement can help.

- open research question, will be clearer in the next few years.
- keep an eye on this development, application developers might have more leverage into memory system behavior in the recent future.



USIMM MEMORY SIMULATOR

- Detailed simulation of DRAM-based memory system
  - Memory Controller
  - Multiple memory channels
  - DIMMs with various organizations
  - Various types of DRAM chips
- Trace-based, or “interactive”

UTAH ARCH

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, Zeshan Chishti

The USIMM Simulator was released as the infrastructure for the 3rd Journal of Instruction Level Parallelism organized Workshop on Computer Architecture Competitions (JWAC-3) held with ISCA-2012 at Portland, OR.

- competition between research groups from universities from US, China, Japan as well as industrial researchers from Korea on the “best” memory scheduler
- judged by a program committee consisting of professors and industrial researchers.

This is a DRAM timing and power modeling tool which models in detail

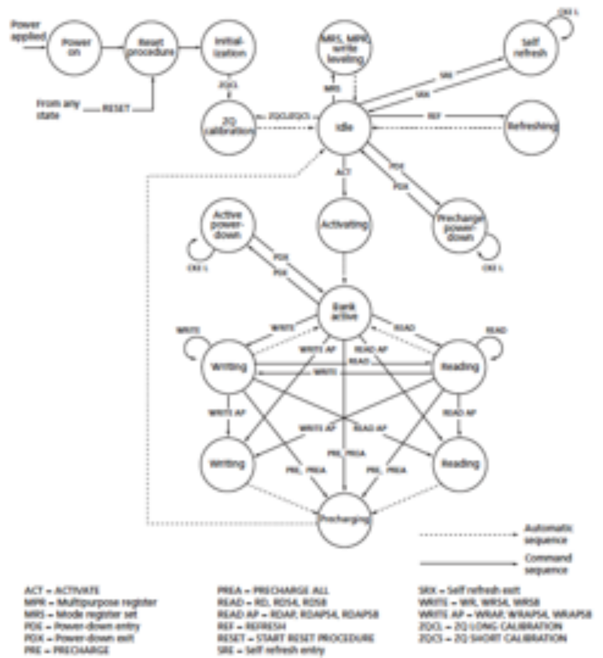
- the memory scheduler
- address mapping
- different memory configurations and devices

The simulator, in its stand alone mode can consume a trace of main memory requests, or it can be coupled with other cycle-driven simulators

- has been used with SIMICS from Windriver and GPGPUSim (besides TRAX which Danny has talked about).

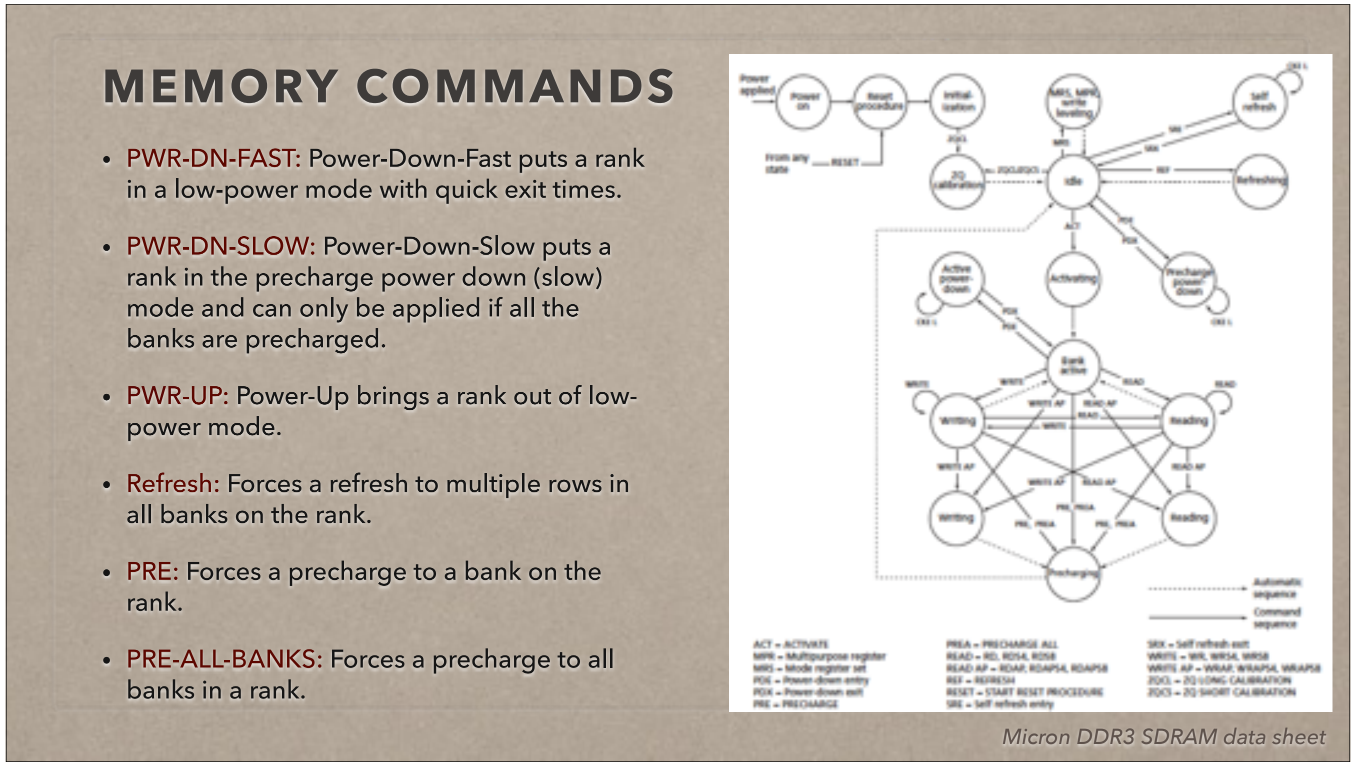
MEMORY COMMANDS

- PRE: Precharge the bitlines of a bank so a new row can be read out.
- ACT: Activate a new row into the bank's row buffer.
- COL-RD: Bring a cache line from the row buffer back to the processor.
- COL-WR: Bring a cache line from the processor to the row buffer.



Micron DDR3 SDRAM data sheet

We model all the DRAM core data commands.



MEMORY COMMANDS

- PWR-DN-FAST: Power-Down-Fast puts a rank in a low-power mode with quick exit times.
- PWR-DN-SLOW: Power-Down-Slow puts a rank in the precharge power down (slow) mode and can only be applied if all the banks are precharged.
- PWR-UP: Power-Up brings a rank out of low-power mode.
- Refresh: Forces a refresh to multiple rows in all banks on the rank.
- PRE: Forces a precharge to a bank on the rank.
- PRE-ALL-BANKS: Forces a precharge to all banks in a rank.

ACT = ACTIVATE
MPR = Multipurpose register
MRS = Mode register set
PDE = Power-down entry
PDX = Power-down exit
PRE = PRECHARGE

PREA = PRECHARGE ALL
READ = RD, RDS4, RDS8
READ AP = RDAP, RDAPS4, RDAPS8
REF = REFRESH
RESET = START RESET PROCEDURE
SRE = Self refresh entry

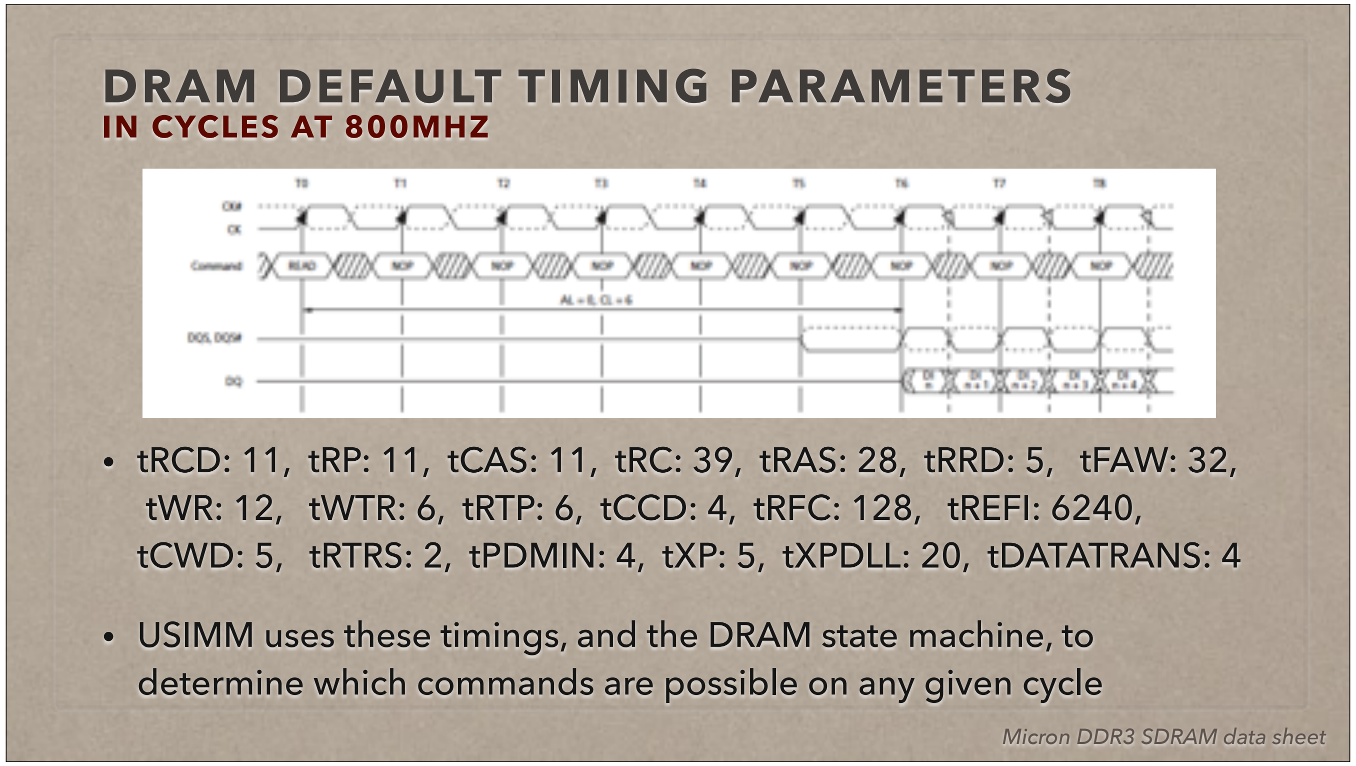
SRX = Self refresh exit
WRITE = WR, WRS4, WRS8
WRITE AP = WRAP, WRAPS4, WRAPS8
ZQCL = ZQ LONG CALIBRATION
ZQCS = ZQ SHORT CALIBRATION

Micron DDR3 SDRAM data sheet

And also different power down commands and refresh.

Memory controller is doing a lot of things beyond your control, behind the scenes.

These hidden things must be modeled accurately!



DRAM DEFAULT TIMING PARAMETERS

IN CYCLES AT 800MHZ

- tRCD: 11, tRP: 11, tCAS: 11, tRC: 39, tRAS: 28, tRRD: 5, tFAW: 32, tWR: 12, tWTR: 6, tRTP: 6, tCCD: 4, tRFC: 128, tREFI: 6240, tCWD: 5, tRTRS: 2, tPDMIN: 4, tXP: 5, tXPDLL: 20, tDATATRANS: 4
- USIMM uses these timings, and the DRAM state machine, to determine which commands are possible on any given cycle

Micron DDR3 SDRAM data sheet

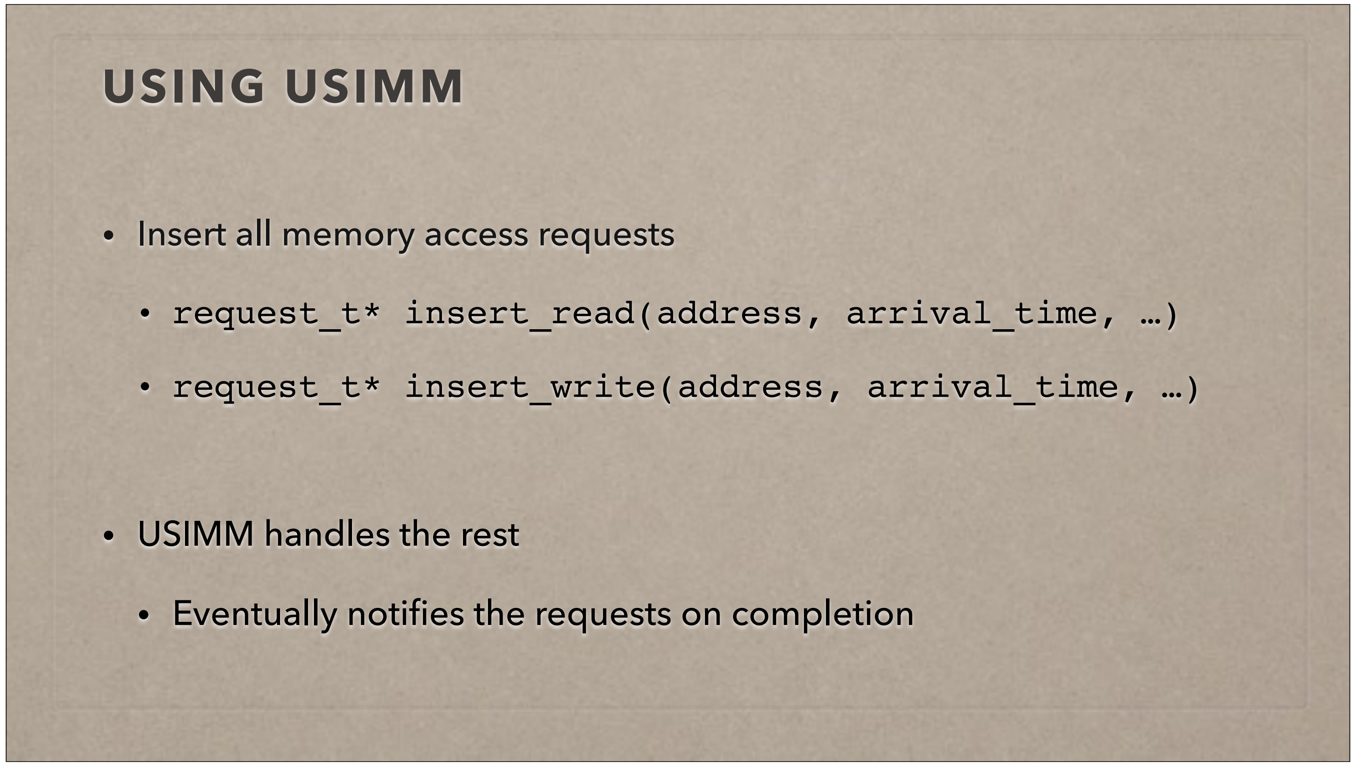
These numbers are cryptic

- essentially there are strict rules on what timing gap is required between commands to the same bank, or different banks of the same rank, and even different ranks.
- all of these are handled in USIMM
- a DRAM microarchitect will be interested in tweaking with these to model a new type of DRAM device or a new device architecture.
- but this is hidden from the scheduler "developer".

USIMM, in each cycle will present the scheduler with a list of options in the form of pending DRAM requests – only a subset of which will be "issuable" in a cycle

- the issuability depends on the DRAM state and timing constraints.

The scheduler should pick the command it wants.



USING USIMM

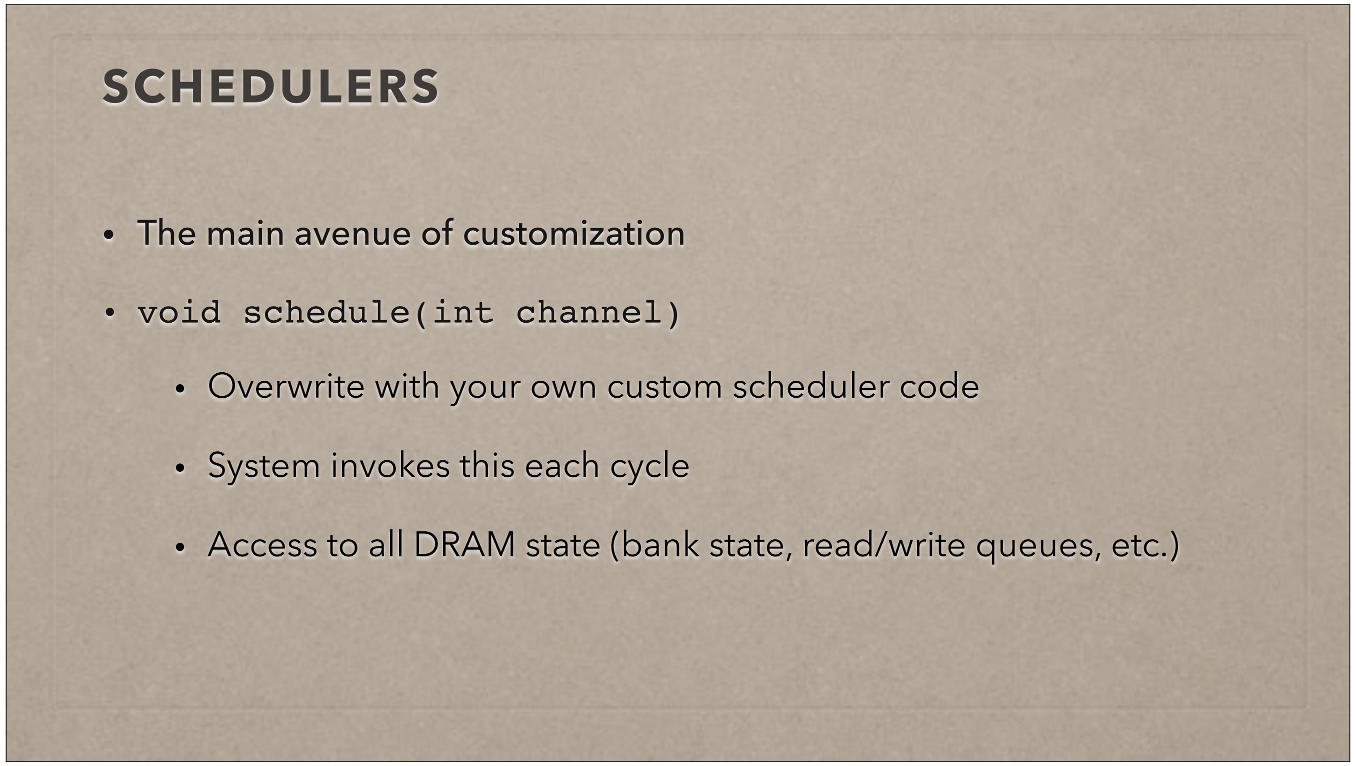
- Insert all memory access requests
  - `request_t* insert_read(address, arrival_time, …)`
  - `request_t* insert_write(address, arrival_time, …)`
- USIMM handles the rest
  - Eventually notifies the requests on completion

Fortunately, this mess is hidden from you.

This is basically the interface to USIMM.

Tell it about all your memory requests. Its internal gears crank away.

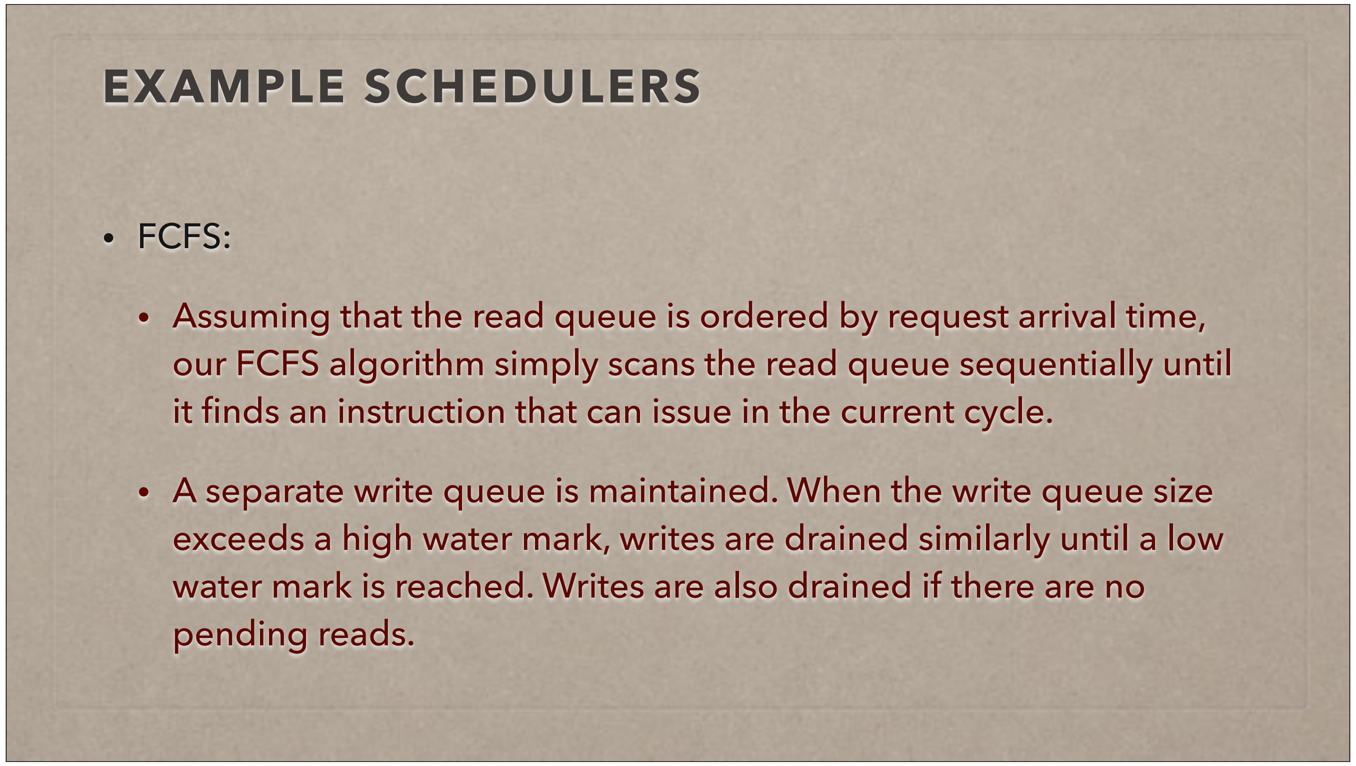
Can be used either on traces, or in real-time.



SCHEDULERS

- The main avenue of customization
- `void schedule(int channel)`
    - Overwrite with your own custom scheduler code
    - System invokes this each cycle
    - Access to all DRAM state (bank state, read/write queues, etc.)

Main avenue of customization - timing parameters are physically constrained.

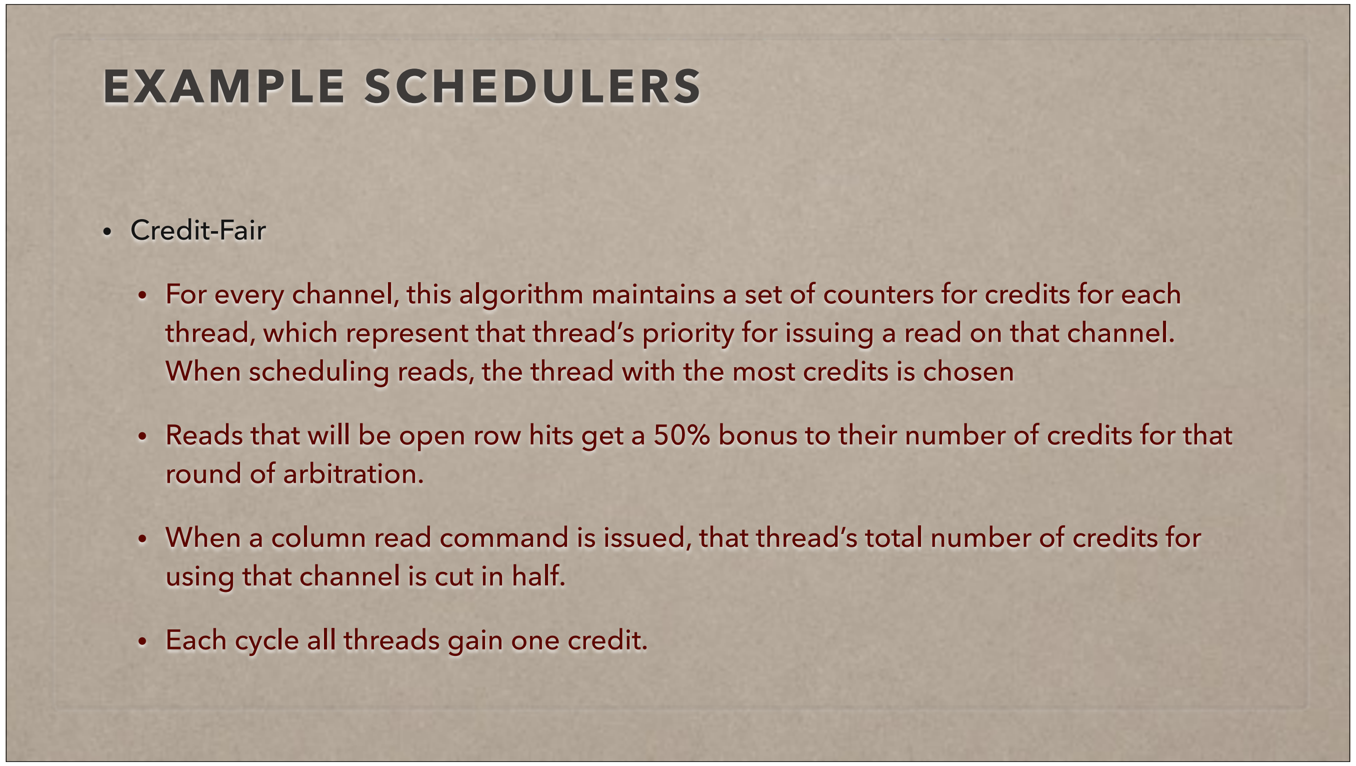


EXAMPLE SCHEDULERS

- FCFS:
  - Assuming that the read queue is ordered by request arrival time, our FCFS algorithm simply scans the read queue sequentially until it finds an instruction that can issue in the current cycle.
  - A separate write queue is maintained. When the write queue size exceeds a high water mark, writes are drained similarly until a low water mark is reached. Writes are also drained if there are no pending reads.

Next we will go through some of the schedulers that are provided as examples with USIMM.

“First come first serve”



EXAMPLE SCHEDULERS

- Credit-Fair
  - For every channel, this algorithm maintains a set of counters for credits for each thread, which represent that thread’s priority for issuing a read on that channel. When scheduling reads, the thread with the most credits is chosen
  - Reads that will be open row hits get a 50% bonus to their number of credits for that round of arbitration.
  - When a column read command is issued, that thread’s total number of credits for using that channel is cut in half.
  - Each cycle all threads gain one credit.

We won’t give details on all these schedulers, but know that there are a lot of things to try.

EXAMPLE SCHEDULERS

- Power-Down
  - This algorithm issues PWR-DN-FAST commands in every idle cycle.
  - Explicit power-up commands are not required as power-up happens implicitly when another command is issued.
- Close-Page
  - In every idle cycle, the scheduler issues precharge operations to banks that last serviced a column read/write.
  - Unlike a true close-page policy, the precharge is not issued immediately after the column read/write and we don't look for potential row buffer hits before closing the row.

EXAMPLE SCHEDULERS

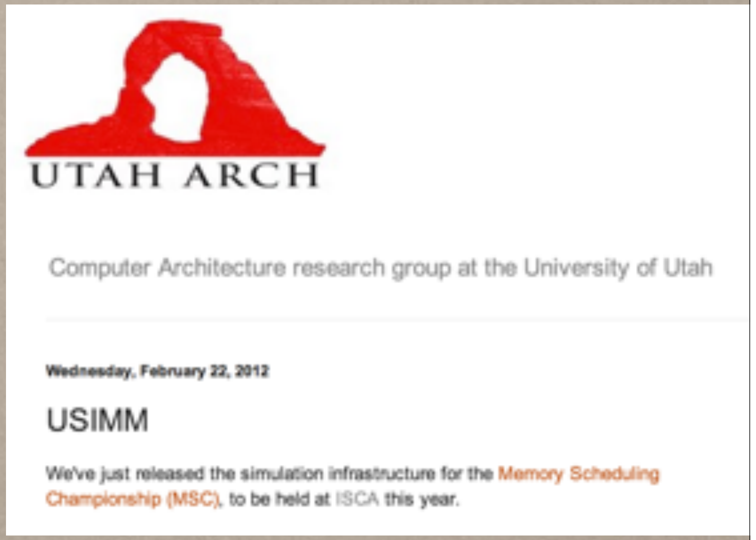
- First-Ready-Round-Robin
  - This scheduler tries to combine the benefits of open row hits with the fairness of a round-robin scheduler.
  - It first tries to issue any open row hits with the “correct” thread-id (as defined by the current round robin flag), then other row hits, then row misses with the “correct” thread-id, and then finally, a random request.

EXAMPLE SCHEDULERS

- MLP-aware
  - The scheduler assumes that threads with many outstanding misses (high memory level parallelism, MLP) are not as limited by memory access time.
  - The scheduler therefore prioritizes requests from low-MLP threads over those from high-MLP threads.
  - To support fairness, a request's wait time in the queue is also considered.

INFO ABOUT USIMM

- The most up-to-date weblink for obtaining the latest version of the simulator is:
  - http://utaharch.blogspot.com/2012/02/usimm.html



UTAH ARCH

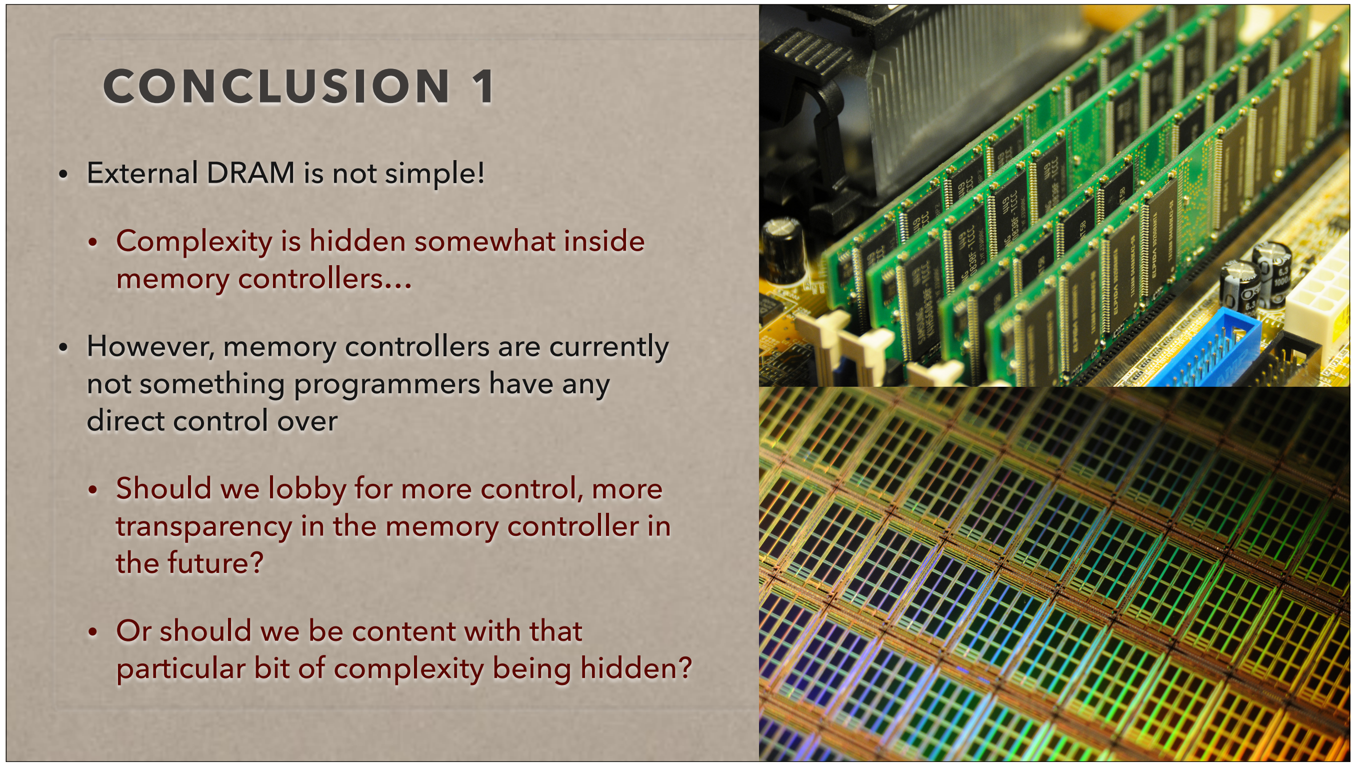
Computer Architecture research group at the University of Utah

Wednesday, February 22, 2012

USIMM

We've just released the simulation infrastructure for the Memory Scheduling Championship (MSC), to be held at ISCA this year.

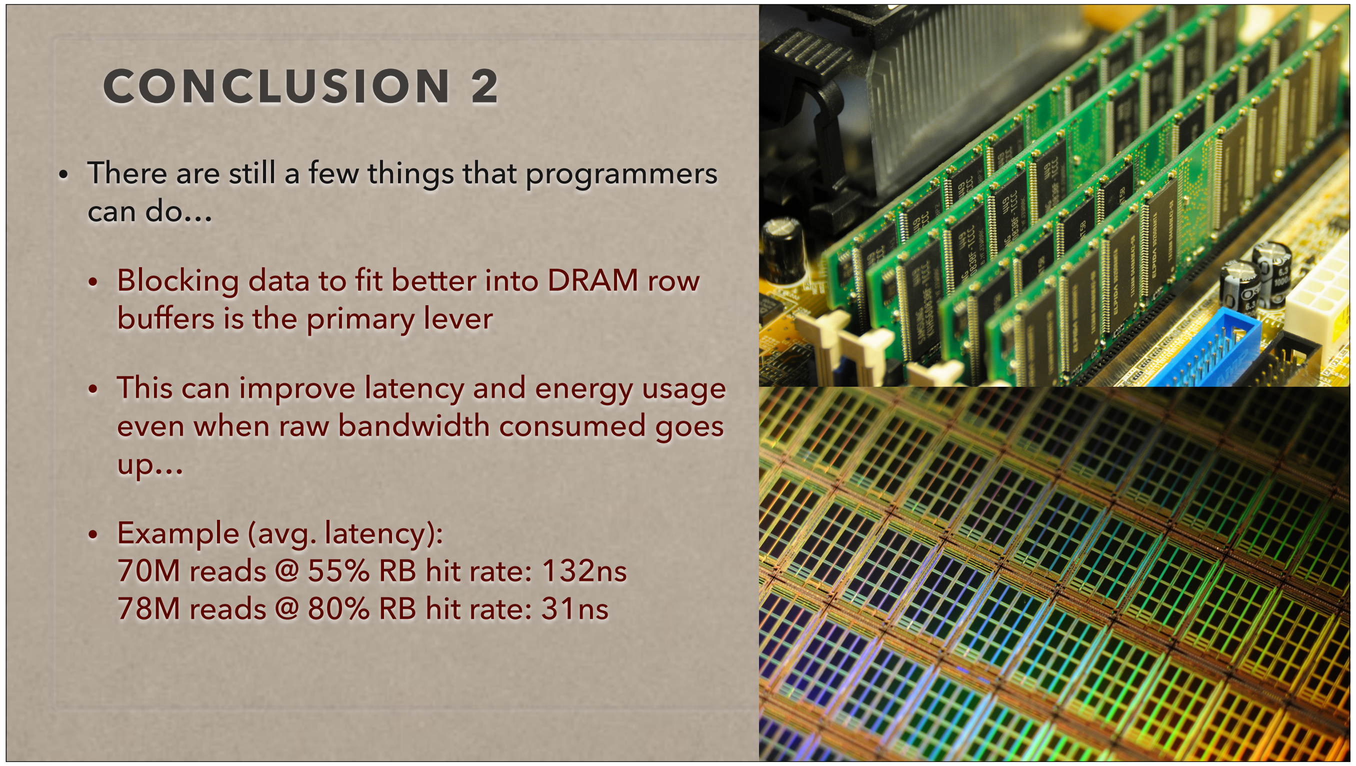
Use it, experiment!



CONCLUSION 1

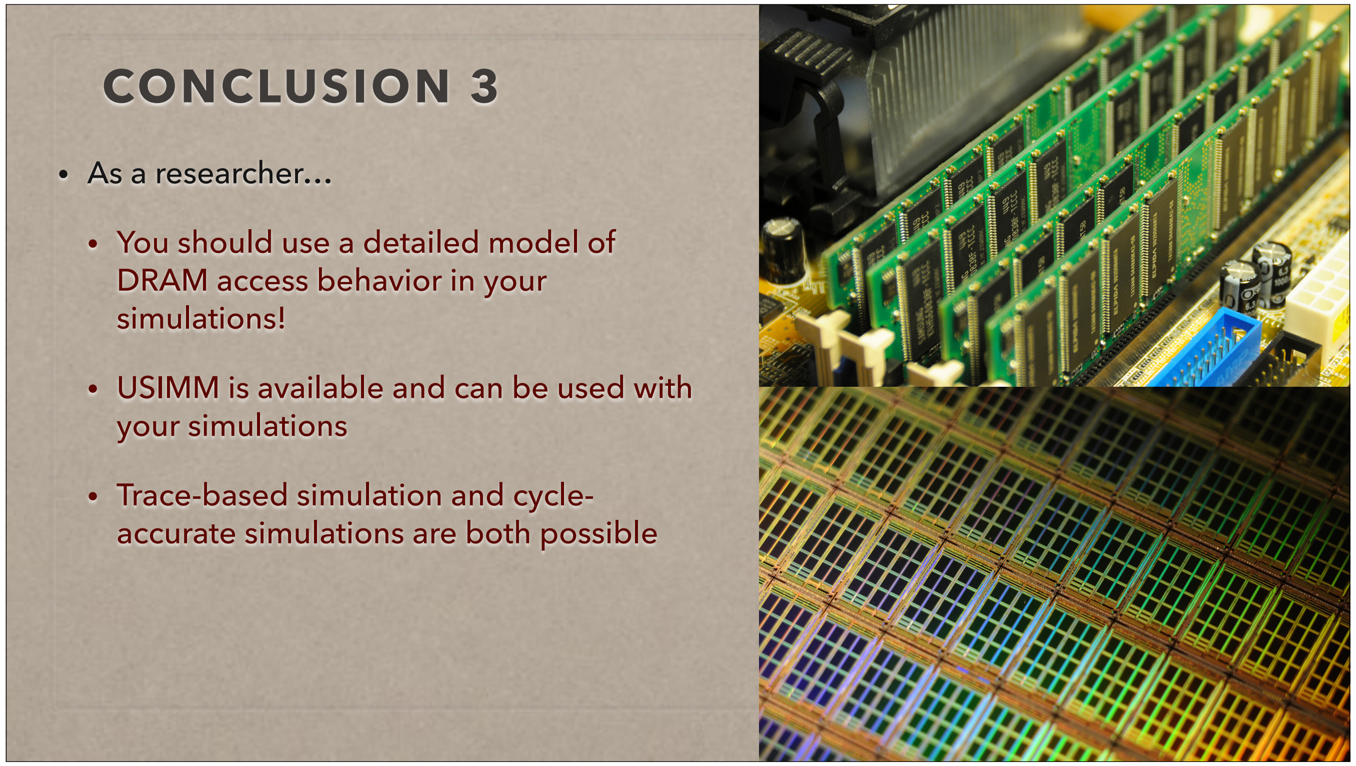
- External DRAM is not simple!
  - Complexity is hidden somewhat inside memory controllers…
- However, memory controllers are currently not something programmers have any direct control over
  - Should we lobby for more control, more transparency in the memory controller in the future?
  - Or should we be content with that particular bit of complexity being hidden?

We shouldn’t just treat DRAM as a black box that gives us data sometimes. Unfortunately, we don’t have much control. Should we lobby for more transparency?



CONCLUSION 2

- There are still a few things that programmers can do…
  - Blocking data to fit better into DRAM row buffers is the primary lever
  - This can improve latency and energy usage even when raw bandwidth consumed goes up…
  - Example (avg. latency):
    70M reads @ 55% RB hit rate: 132ns
    78M reads @ 80% RB hit rate: 31ns



CONCLUSION 3

- As a researcher…
  - You should use a detailed model of DRAM access behavior in your simulations!
  - USIMM is available and can be used with your simulations
  - Trace-based simulation and cycle-accurate simulations are both possible

As a researcher, experiment with high fidelity simulators! Make DRAM more application aware.

CONTACT

- Erik Brunvand
  University of Utah
  elb@cs.utah.edu
- Daniel Kopta
  University of Utah
  dkopta@cs.utah.edu
- Niladrish Chatterjee
  NVIDIA Corporation
  nil@nvidia.com

THE UNIVERSITY OF UTAH
NVIDIA

Thanks!