CS 250B: Modern Computer Systems

Cache And Memory System

Sang-Woo Jun

UCI

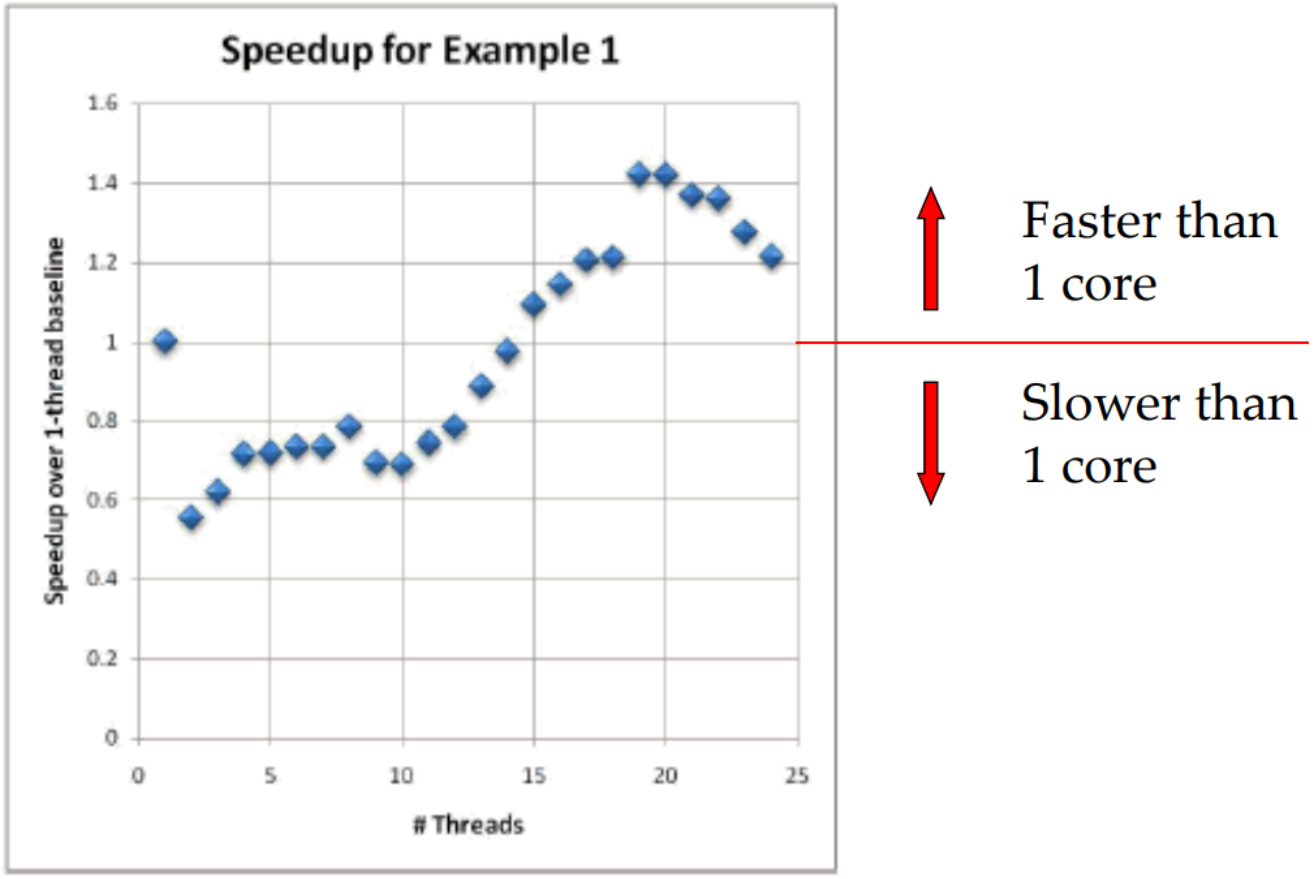
Motivation Example:
An Embarrassingly Parallel Workload

- A very simple example of counting odd numbers in a large array

```
int results[THREAD_COUNT];
void worker_thread(…) {
      int tid = …;
      for (e in myChunk) {
            if ( e % 2 != 0) results[tid]++;
      }
}
```

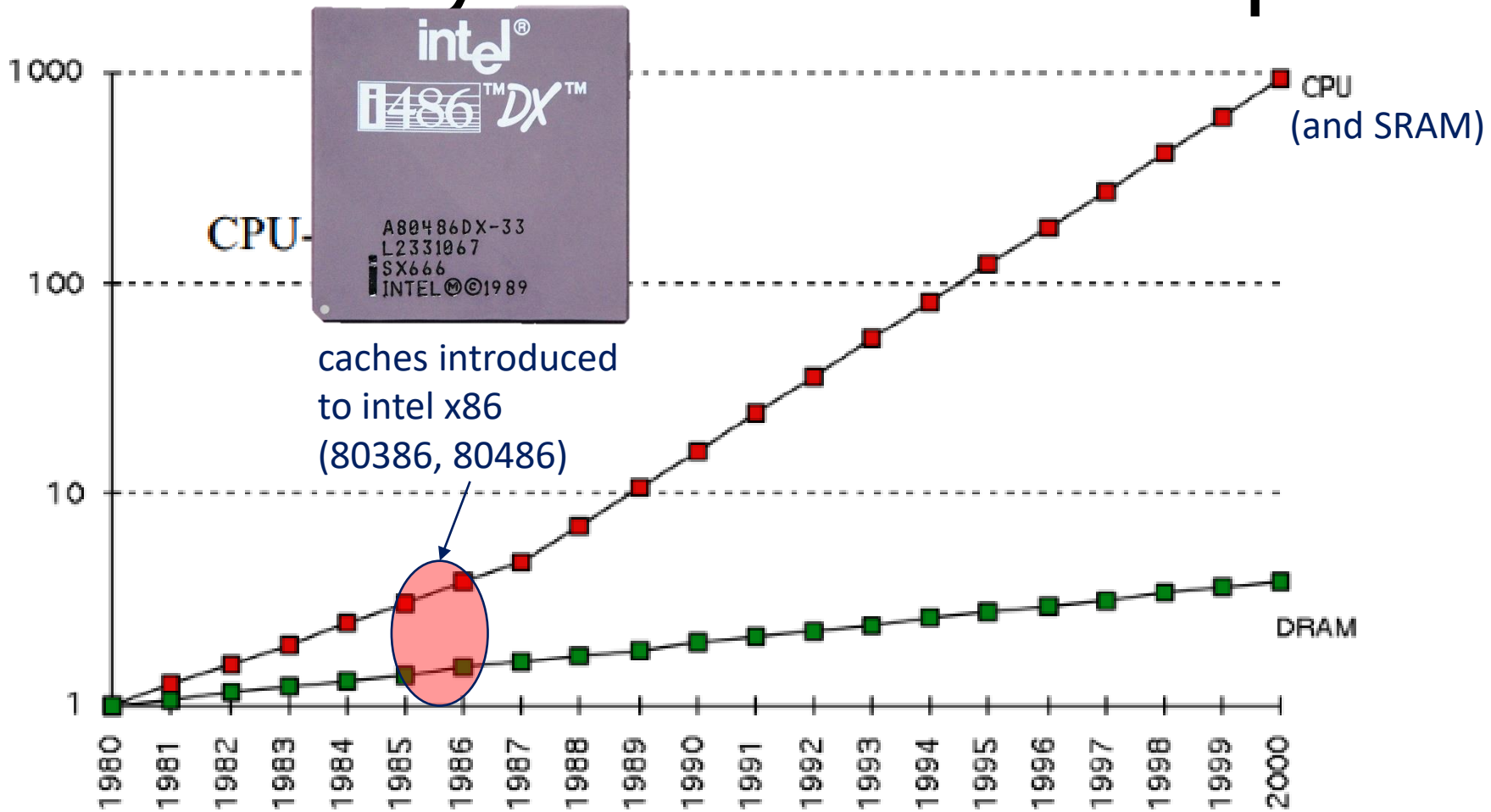
Do you see any performance red flags?

Scalability Unimpressive



Scott Meyers, “CPU Caches and Why You Care,” 2013

History of The Processor/Memory Performance Gap



What is the Y-axis? Most likely normalized latency reciprocal

Source: Extreme tech, “How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips,” 2018

Purpose of Caches

- The CPU is (largely) unaware of the underlying memory hierarchy
  - The memory abstraction is a single address space
  - The memory hierarchy automatically stores data in fast or slow memory, depending on usage patterns
- Multiple levels of “caches” act as interim memory between CPU and main memory (typically DRAM)
  - Processor accesses main memory through the cache hierarchy
  - If requested address is already in the cache (address is “cached”, resulting in “cache hit”), data operations can be fast
  - If not, a “cache miss” occurs, and must be handled to return correct data to CPU

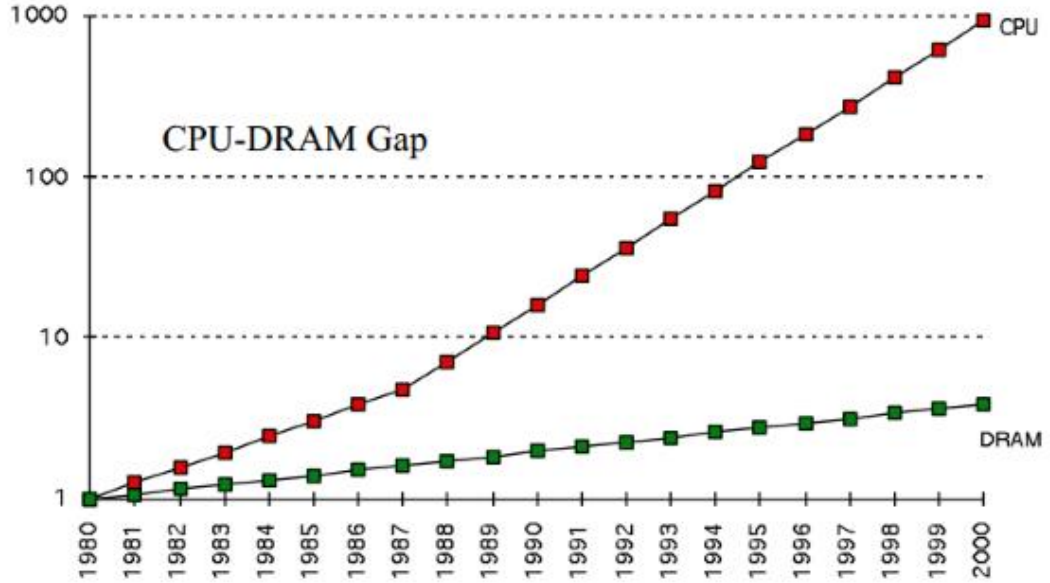
Caches Try to Be Transparent

- Software is (ideally) written to be oblivious to caches
  - Programmer should not have to worry about cache properties
  - Correctness isn't harmed regardless of cache properties

- However, the performance impact of cache affinity is quite high!
  - Performant software cannot be written in a completely cache-oblivious way

History of The Processor/Memory Performance Gap

- Processor vs Memory Performance



1980: no cache in microprocessor;
1995 2-level cache

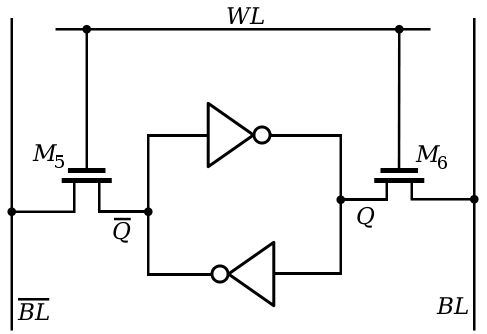
What is the Y-axis? Most likely normalized latency reciprocal

- 80386 (1985) :
  Last Intel desktop CPU with no on-chip cache
  (Optional on-board cache chip though!)
- 80486 (1989) : 4 KB on-chip cache
- Coffee Lake (2017) :
  64 KiB L1 Per core
  256 KiB L2 Per core
  Up to 2 MiB L3 Per core (Shared)

Source: Extreme tech, “How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips,” 2018

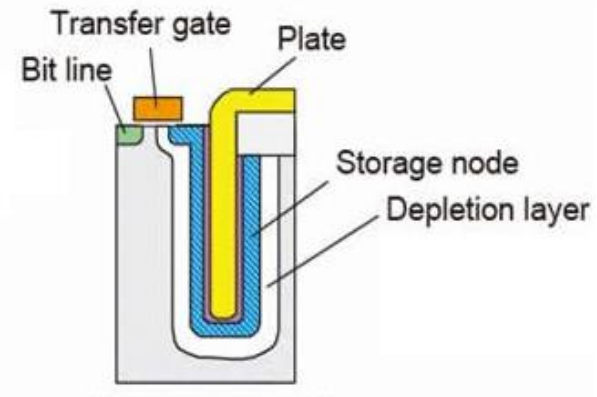
Why The Gap? SRAM vs. DRAM

- SRAM (Static RAM) – Register File, Cache
  - Built using transistors, which processor logic is made of
  - As fast as the rest of the processor



Source: Inductiveload, from commons.wikimedia.org

- DRAM (Dynamic RAM)
  - Built using capacitors, which can hold charge for a short time
  - Controller must periodically read all data and write it back ("Refresh")
    - Hence, "Dynamic" RAM
  - Requires fabrication process separate from processor
  - Reading data from a capacitor is high-latency
    - EE topics involving sense amplifiers, which we won't get into



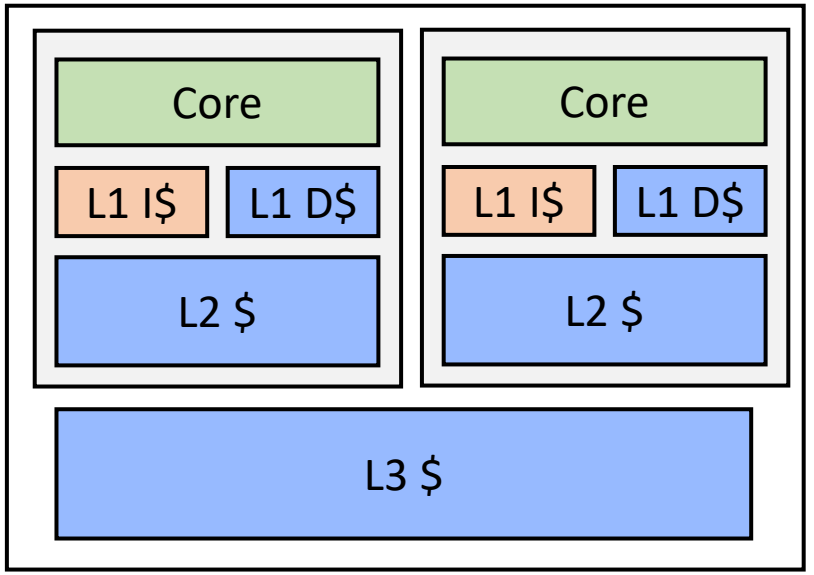
Note: Old, "trench capacitor" design

Sunami, Hideo. "The Role of the Trench Capacitor in DRAM Innovation." IEEE Solid-State Circuits Newsletter 13, 2008

Multi-Layer Cache Architecture

Numbers from modern Xeon processors (Sapphire Rapids)

Cache Level	Size	Latency (Cycles)
L1	48 KiB	< 5
L2	2 MiB	< 20
L3	~ 2 MiB per core	< 50

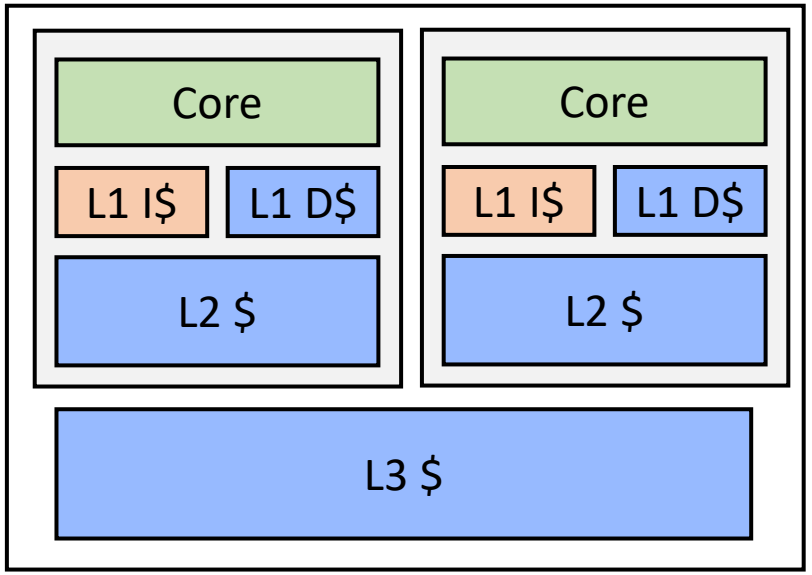


- Even with SRAM there is a size-performance trade-off
  - Not because the transistors are any different!
  - Cache management logic becomes more complicated with larger sizes
- L1 cache accesses can be hidden in the pipeline
  - Modern processors have pipeline depth of 14+
  - All others take a performance hit

Multi-Layer Cache Architecture

Numbers from modern Xeon processors (Broadwell – Kaby lake)

Cache Level	Size	Latency (Cycles)
L1	48 KiB	< 5
L2	2 MiB	< 20
L3	~ 2 MiB per core	< 50
DRAM	100s of GB	> 100*



- *This is in an ideal scenario
  - Actual measurements could be multiple hundreds or thousands of cycles!
- DRAM systems are complicated entities themselves
  - Latency/Bandwidth of the same module varies immensely by situation…

Cache Line Unit of Management

- CPU Caches are managed in units of large “Cache Lines”
  - Typically 64 bytes in modern x86 processors
- Why not smaller units?
  - Word-size management is natural to reason about. Why not this?

Reminder: Direct Mapped Cache Access

- For cache with 2^W cache lines
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, cache hit!

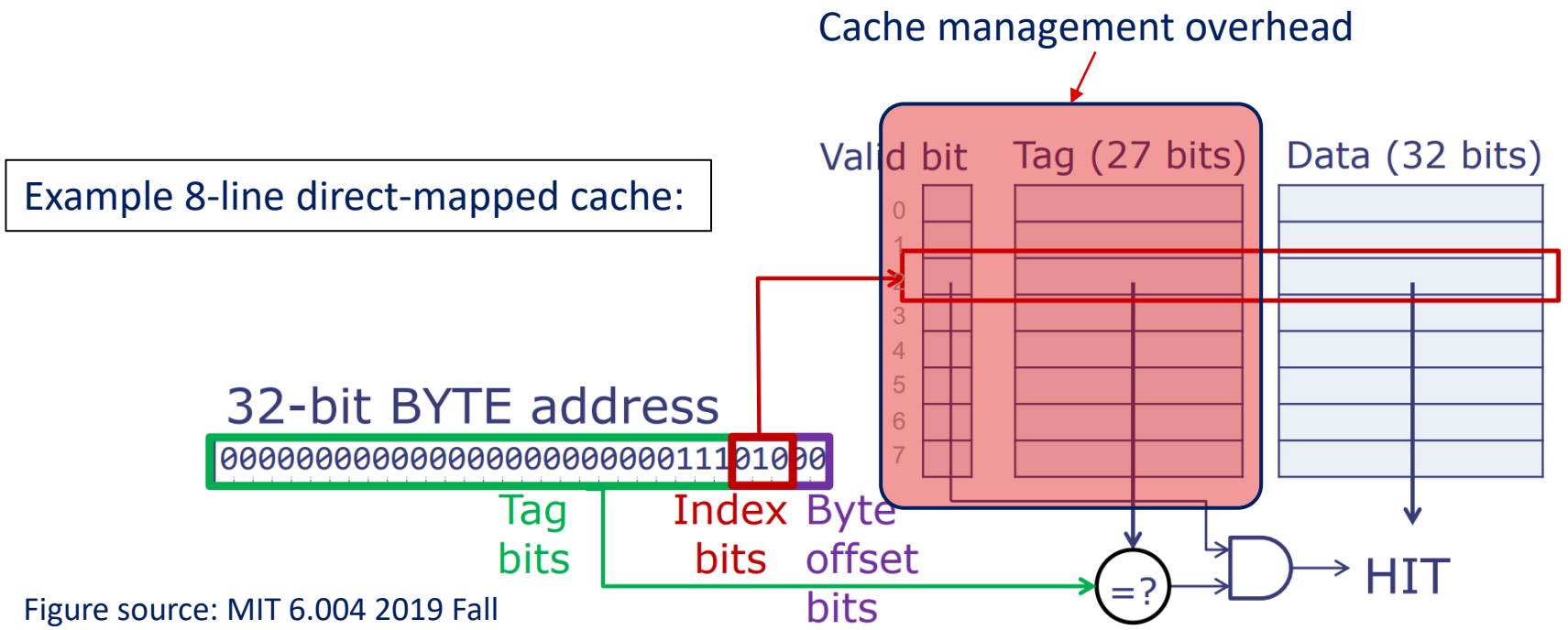


Figure source: MIT 6.004 2019 Fall

Larger Cache Lines

- Reduce cache management overhead: Store multiple words per data line
  - Always fetch entire block (multiple words) from memory
  - **+ Advantage**: Reduces size of tag memory
  - **- Disadvantage**: Fewer indices in the cache -> Higher miss rate

Example: 4-block, 16-word direct-mapped cache

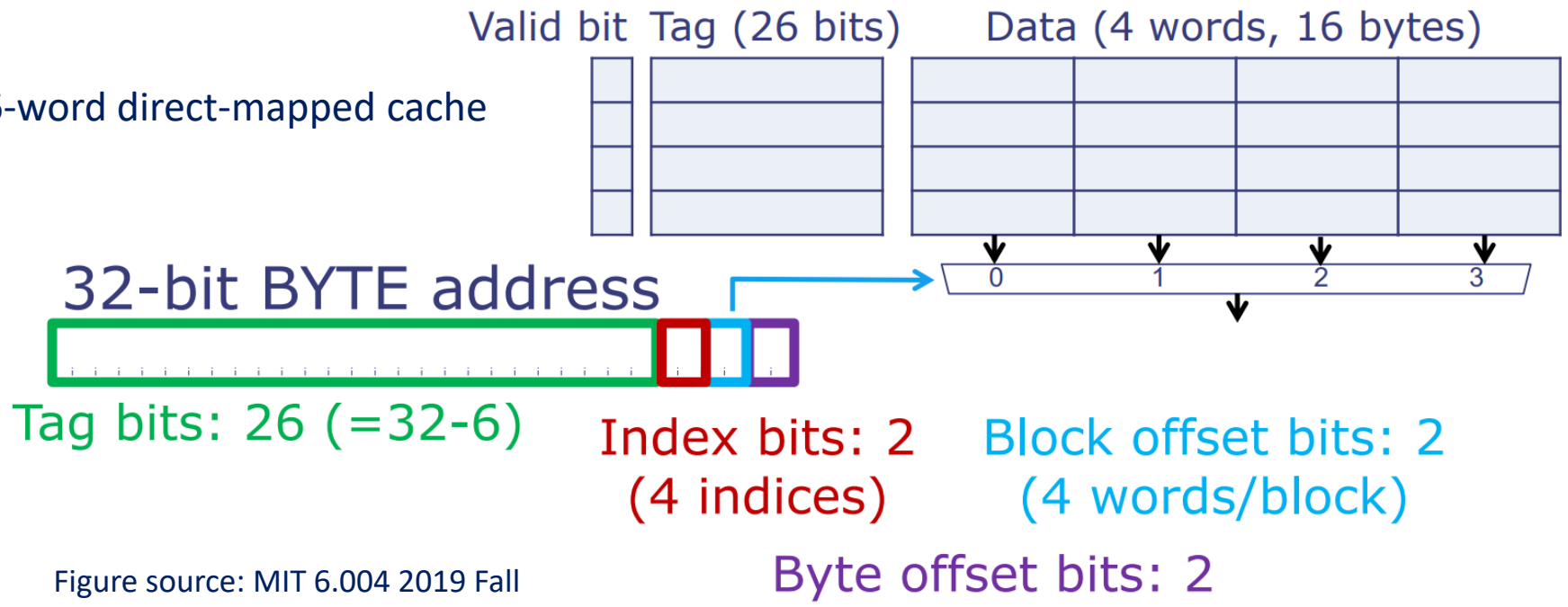


Figure source: MIT 6.004 2019 Fall

Larger Cache Lines

- Caches are managed in ***Cache Line*** granularity
  - Typically 64 Bytes for modern CPUs
  - 64 Bytes == 16 4-byte integers
  - Balance of performance and on-chip SRAM usage
- Reading/Writing happens in cache line granularity
  - Read one byte not in cache -> Read all 64 bytes from memory
  - Write one byte -> Eventually write all 64 bytes to memory
  - Inefficient cache access patterns really hurt performance!

Block Size Trade-Offs

- Larger block sizes…
  - Take advantage of spatial locality (also, DRAM is faster with larger blocks)
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time (more logic) and miss ratio (less lines)
- AMAT (Average Memory Access Time)
  - = HitTime + MissPenalty*MissRatio

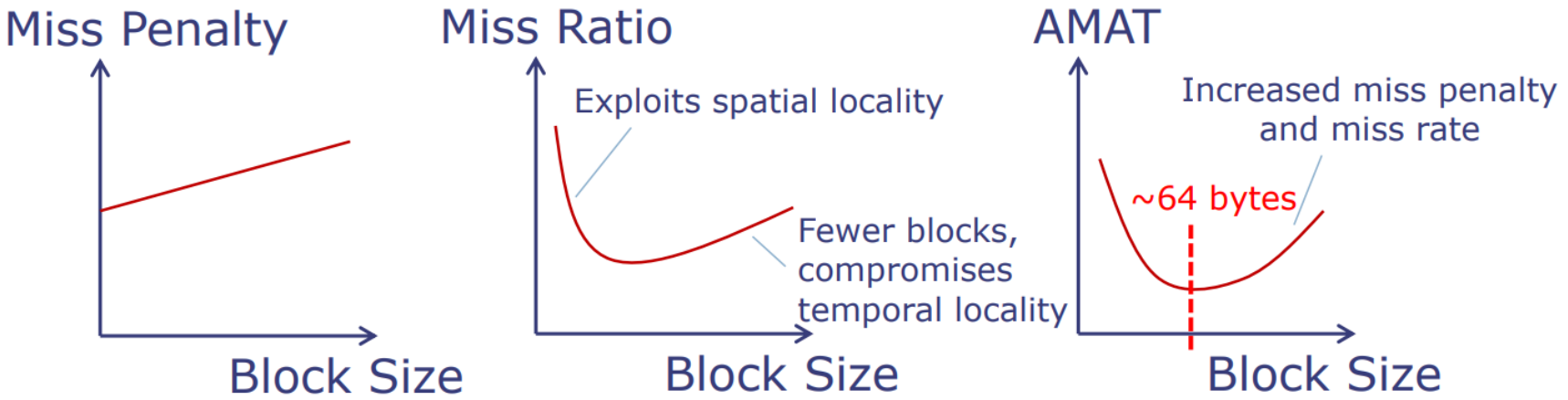


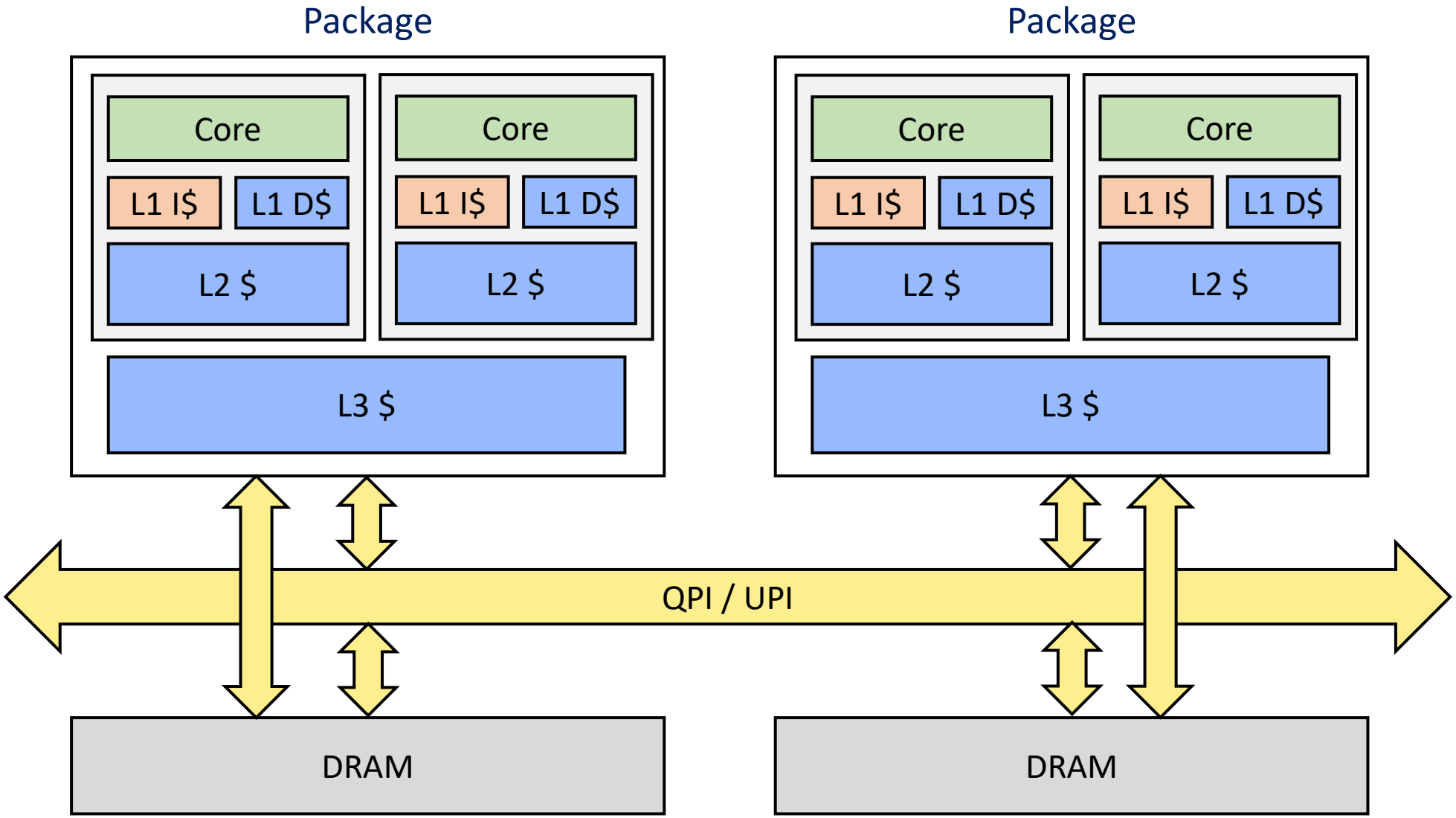
Figure source: MIT 6.004 2019 Fall

An Analytical Example: Two 4 KiB Caches

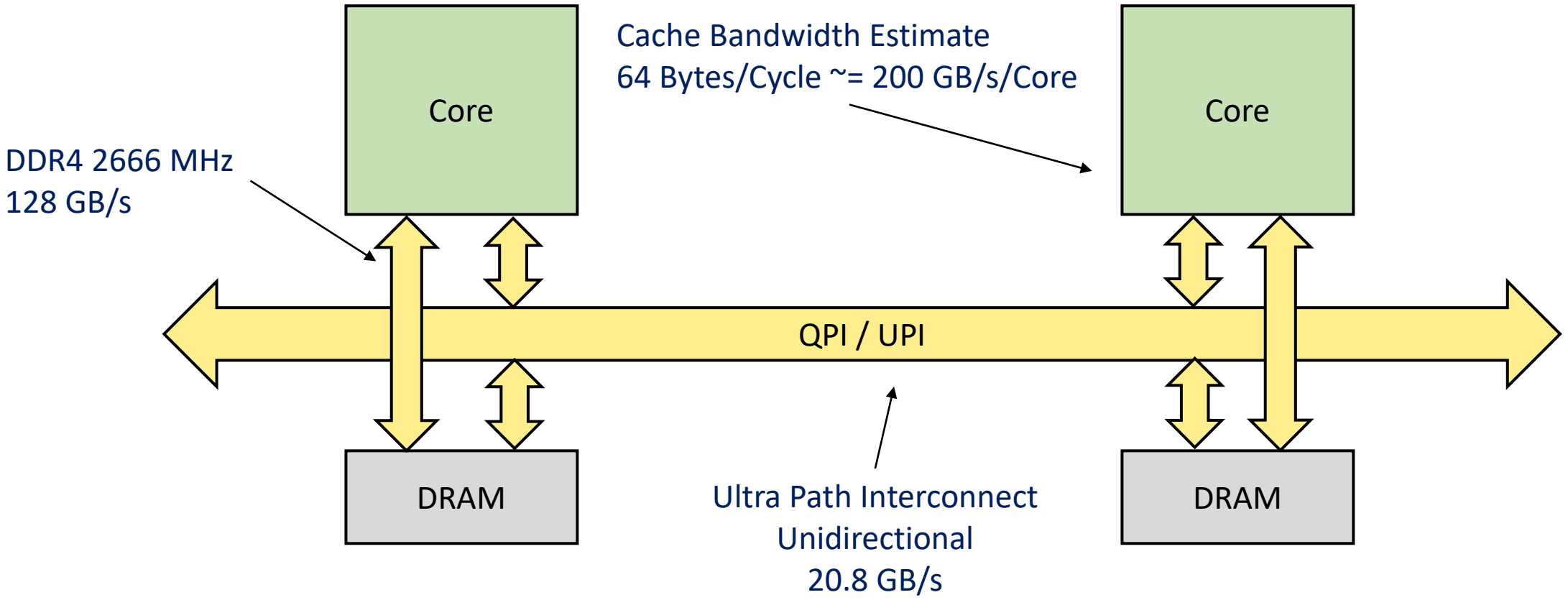
- 4-way set-associative, cache line size of 16 bytes
  - Each set == 64 bytes -> 64 sets
  - Assuming 32 bit addresses: 22 bit tag + valid + dirty = 24 bits per line
  - 768 bytes of overhead per 4 KiB cache
  - Total SRAM requirement: 4 KiB + 768 bytes = **4864** bytes
- Direct-mapped, cache line size of 4 bytes
  - Each line == 4 bytes -> 1024 lines
  - Assuming 32 bit addresses: 20 bit tag + valid + dirty = 22 bits per line
  - 2816 bytes of overhead per 4 KiB cache
  - Total SRAM requirement: 4 KiB + 2816 bytes = **6912** bytes

Memory System Architecture

Two packages make up a NUMA (Non-Uniform Memory Access) Configuration



Memory System Bandwidth Snapshot



Memory/PCIe controller used to be on a separate “North bridge” chip, now integrated on-die
All sorts of things are now on-die! Even network controllers! (Specialization!)

Reminder: Cache Coherency

- Cache coherency
  - Informally: Read to **each address** must return the most recent value
  - Typically: All writes must be visible at some point, and in proper order
- Coherency protocol implemented between each core's private caches
  - MSI, MESI, MESIF, …
  - Won't go into details here
- Simply put:
  - When a core writes a cache line
  - All other instances of that cache line needs to be invalidated
- Emphasis on ***cache line***

Cache Prefetching

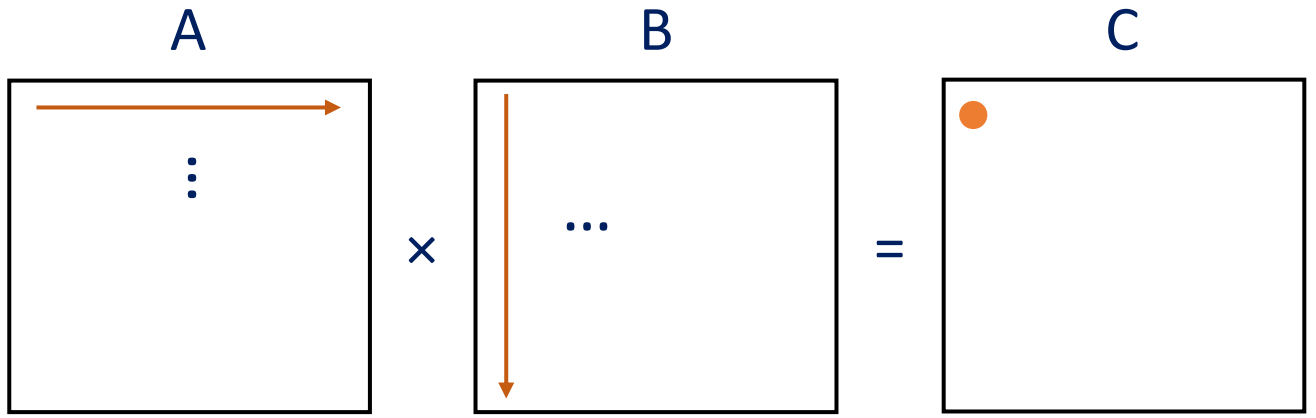
- CPU speculatively prefetches cache lines
  - While CPU is working on the loaded 64 bytes, 64 more bytes are being loaded
- Hardware prefetcher is usually not very complex/smart
  - Sequential prefetching (N lines forward or backwards)
  - Strided prefetching
- Programmer-provided prefetch hints
  - __builtin_prefetch(address, r/w, temporal locality?); for GCC
  - Will generate prefetch instructions if available on architecture

Now That’s Out of The Way…

Cache Efficiency Issue #1: Cache Line Size Matrix Multiplication and Caches

- Multiplying two NxN matrices (C = A × B)

```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * B[k][j]
```



2048*2048 on a i5-7400 @ 3 GHz using GCC –O3 = 63.19 seconds

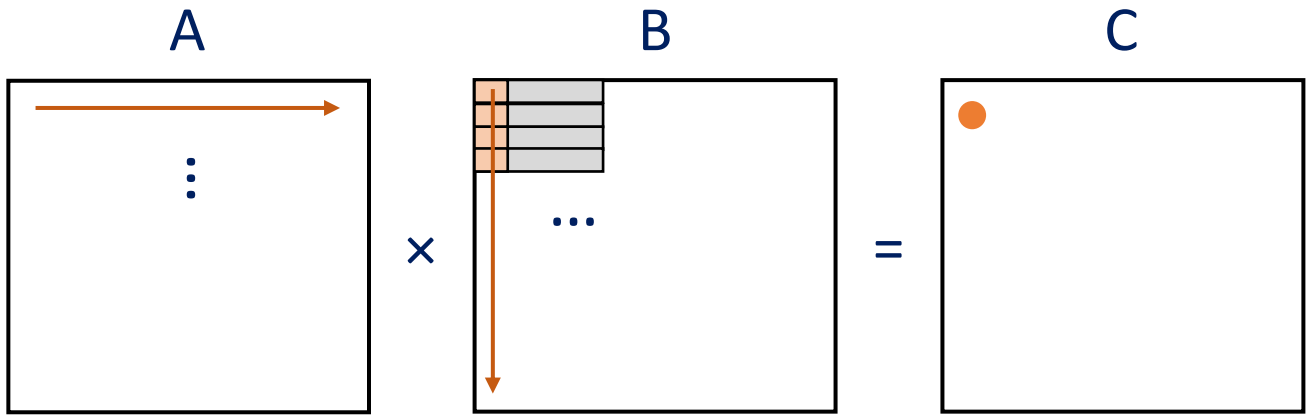
is this fast?

Whole calculation requires 2K * 2K * 2K = 8 Billion floating-point mult + add
At 3 GHz, ~5 seconds just for the math. Over 1000% overhead!

Cache Efficiency Issue #1: Cache Line Size Matrix Multiplication and Caches

- Column-major access makes inefficient use of cache lines
  - A 64 Byte block is read for each element loaded from B
  - 64 bytes read from memory for each 4 useful bytes
- Shouldn't caching fix this? Unused bits should be useful soon!
  - 64 bytes x 2048 = 128 KB … Already overflows L1 cache (~32 KB)

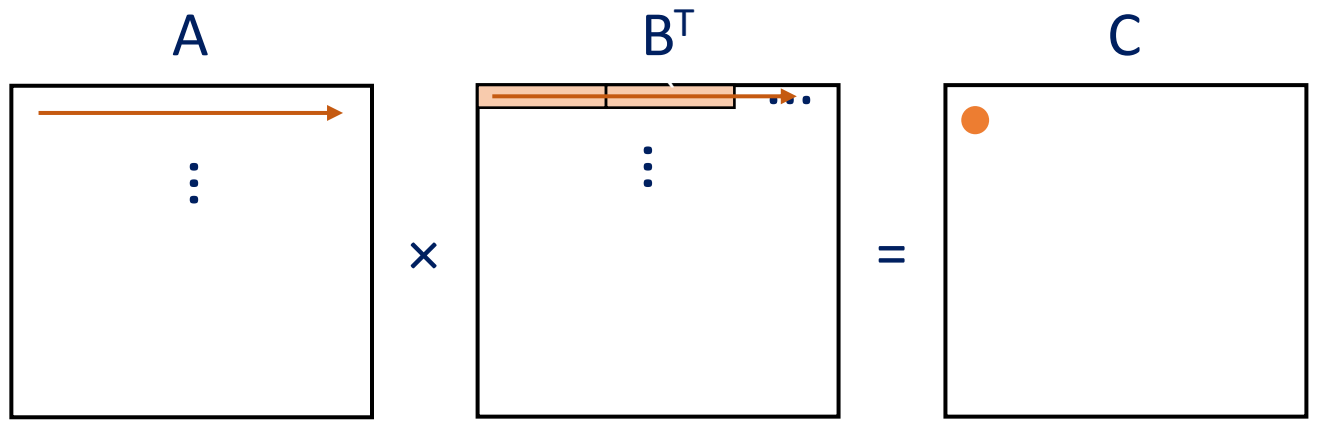
```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * B[k][j]
```



Cache Efficiency Issue #1: Cache Line Size Matrix Multiplication and Caches

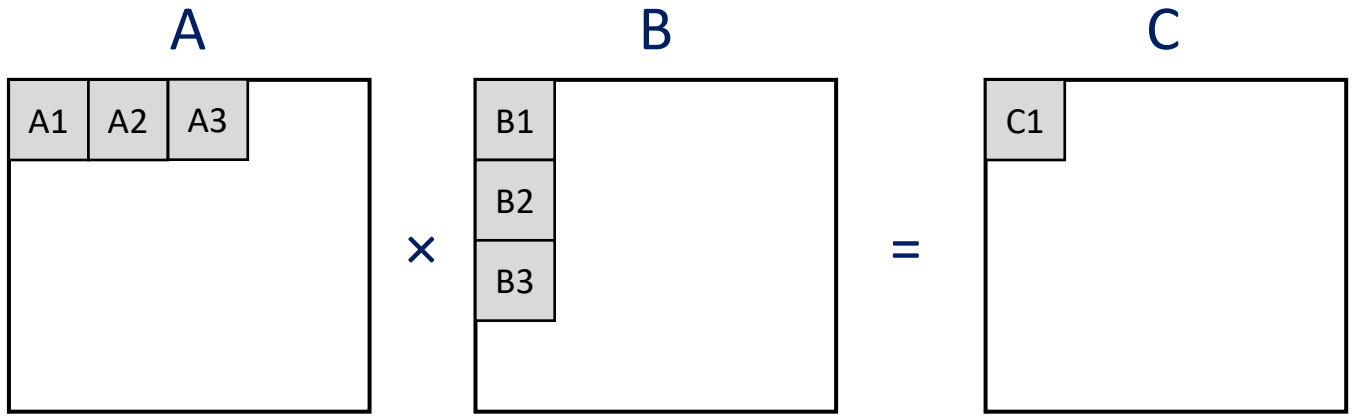
- One solution: Transpose B to match cache line orientation
  - Does transpose add overhead? Not very much as it only scans B once
- Drastic improvements!
  - Before: 63.19s
  - After: 10.39s … 6x improvement!
  - But still not quite ~5s

```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * Bt[j][k]
```



Cache Efficiency Issue #2: Capacity Considerations

- Performance is best when working set fits into cache
  - But as shown, even 2048 x 2048 doesn't fit in cache
  - -> 2048 * 2048 * 2048 elements read from memory for matrix B
- Solution: Divide and conquer! – Blocked matrix multiply
  - For block size 32 × 32 -> 2048 * 2048 * (2048/32) reads



C1 sub-matrix = A1×B1 + A2×B2 + A3×B3 …

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked Transposed	7.35	8.60

- Blocked Transposed bottlenecked by computation
  - Peak theoretical FLOPS for my processor running at 3 GHz ~= 3 GFLOPS
  - 7.35s for matrix multiplication ~= 2.18 GFLOPS
  - Not bad, considering need for branches and other instructions!
  - L1 cache access now optimized, but not considers larger caches
- This chart will be further extended in the next lectures
  - Normalized performance will reach 57 (~1 second elapsed)

Writing Cache Line Friendly Software

- (Whenever possible) use data in coarser-granularities
  - Each access may load 64 bytes into cache, make use of them!
  - e.g., Transposed matrix B in matrix multiply, blocked matrix multiply
- Many profilers will consider the CPU “busy” when waiting for cache
  - Can’t always trust “CPU utilization: 100%”

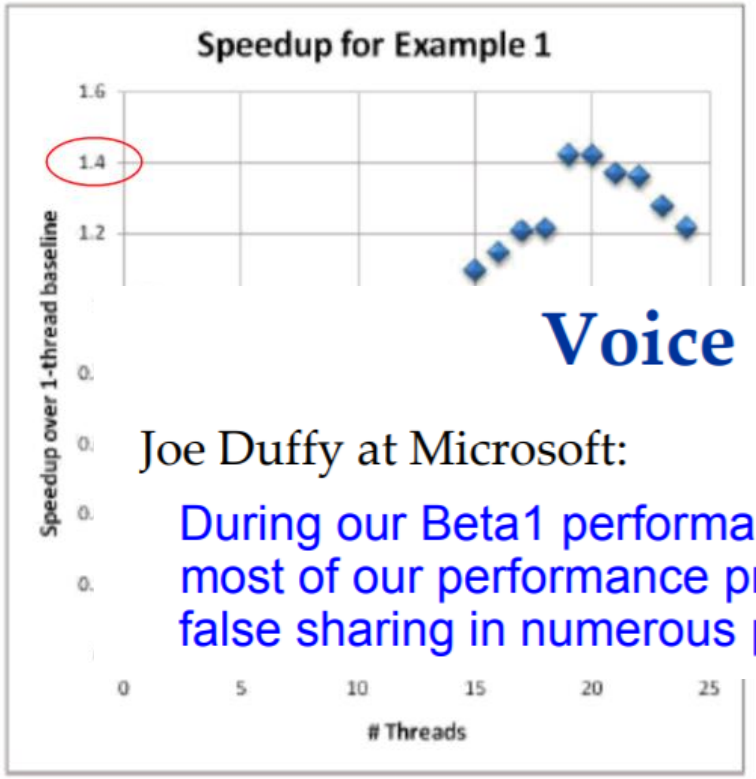
Aside: Object-Oriented Programming And Caches

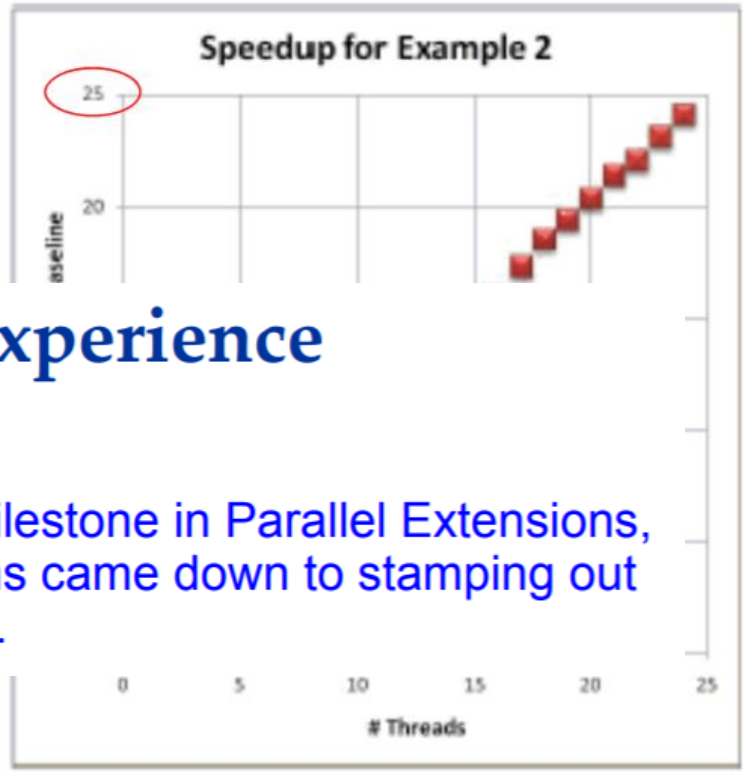
- OOP wants to collocate all data for an entity in a class/struct
  - All instance variables are located together in memory
- Cache friendly OOP
  - All instance variables are accessed whenever an instance is accessed
- Cache unfriendly OOP
  - Only a small subset of instance variables are accessed per instance access
  - e.g., a “for” loop checking the “valid” field of all entities
    - 1 byte accessed per cache line read!
- Non-OOP solution: Have a separate array for “valid”s
  - Is this a desirable solution? Maybe…

Cache Efficiency Issue #3: False Sharing

- Different memory locations, written to by different cores, mapped to same cache line
  - Core 1 performing “results[0]++;”
  - Core 2 performing “results[1]++;”
- Remember cache coherence
  - Every time a cache is written to, all other instances need to be invalidated!
  - “results” variable is ping-ponged across cache coherence every time
  - Bad when it happens on-chip, terrible over processor interconnect (QPI/UPI)
- Simple solution: Store often-written data in local variables

Removing False Sharing





Voice of Experience

Joe Duffy at Microsoft:

During our Beta1 performance milestone in Parallel Extensions, most of our performance problems came down to stamping out false sharing in numerous places.

With False Sharing

Without False Sharing

Aside: Non Cache-Related Optimizations: Loop Unrolling

- Increase the amount of work per loop iteration
  - Improves the ratio between computation instructions and branch instructions
  - Compiler can be instructed to automatically unroll loops
  - Increases binary size, because unrolled iterations are now duplicated code

Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

After loop unrolling

```
int x;
for (x = 0; x < 100; x += 5 )
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

Source: Wikipedia “Loop unrolling”

Aside: Non Cache-Related Optimizations: Function Inlining

- A small function called very often may be bottlenecked by call overhead
- Compiler copies the instructions of a function into the caller
  - Removes expensive function call overhead (stack management, etc)
  - Function can be defined with “inline” flag to hint the compiler
    - “inline int foo()”, instead of “int foo()”

- Personal anecdote
  - Inlining a key (very small) kernel function resulted in a 4x performance boost

Issue #4
Instruction Cache Effects

- Instructions are also stored in cache
  - L1 cache typically has separate instances for instruction and data caches
    - In most x86 architectures, 32 KiB each
    - L2 onwards are shared
  - Lots of spatial locality, so miss rate is usually very low
    - On SPEC, ~2% at L1
  - But adversarial examples can still thrash the cache
- Instruction cache often has dedicated prefetcher
  - Understands concepts of branches and function calls
  - Prefetches blocks of instructions without branches

Optimizing Instruction Cache

- Instruction cache misses can effect performance
  - "Linux was routing packets at **~30Mbps** [wired], and wireless at **~20**. Windows CE was crawling at barely **12Mbps** wired and **6Mbps** wireless.
  - […] After we changed the routing algorithm to be more cache-local, we started doing **35Mbps** [wired], and **25Mbps** wireless – 20% better than Linux.
    – Sergey Solyanik, Microsoft

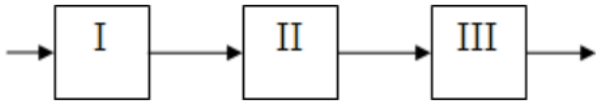
  - [By organizing function calls in a cache-friendly way, we] achieved a 34% reduction in instruction cache misses and a 5% improvement in overall performance.
    -- Mircea Livadariu and Amir Kleen, Freescale

Improving Instruction Cache Locality #1

- Careful with loop unrolling
  - They reduce branching overhead, but reduces effective I$ size
  - When gcc's –O3 performs slower than –O2, this is usually what's happening
- Careful with function inlining
  - Inlining is typically good for very small* functions
  - A rarely executed path will just consume cache space if inlined
- Move conditionals to front as much as possible
  - Long paths of no branches good fit with instruction cache/prefetcher

Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality




```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

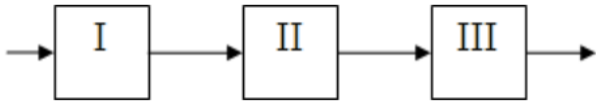
If the functions stage_I, stage_II, and stage_III are sufficiently large, their instructions will thrash the instruction cache!

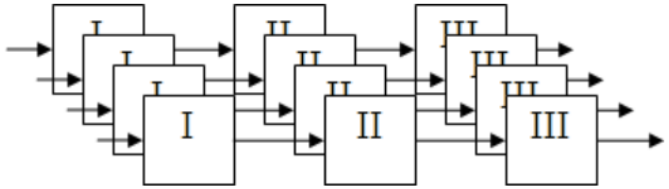
Baseline: Sequential algorithm

Livadariu et. al., “Optimizing for instruction caches,” EETimes

Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality






```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

Baseline: Sequential algorithm

```
for (i=0;i<N;i++)
        temp[i]=stage_I(input[i]);
for (i=0;i<N;i++)
        temp[i]=stage_II(temp[i]);
for (i=0;i<N;i++)
        output[i]= stage_III(temp[i]);
```

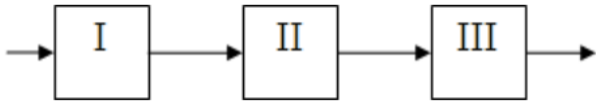
Ordering changed for cache locality

New array “temp” takes up space. N could be large!

Livadariu et. al., “Optimizing for instruction caches,” EETimes

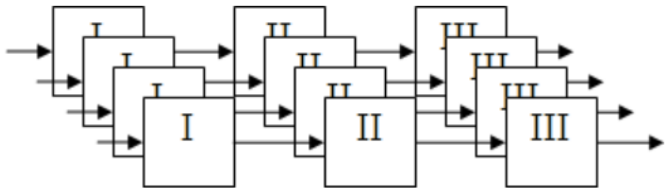
Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality




```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

Baseline: Sequential algorithm




```
for (i=0;i<N;i++)
        temp[i]=stage_I(input[i]);
for (i=0;i<N;i++)
        temp[i]=stage_II(temp[i]);
for (i=0;i<N;i++)
        output[i]= stage_III(temp[i]);
```

Ordering changed for cache locality

```
for (j=0;j<N;j+=M)
{
        for (i=0;i<M;i++)
                temp[i]=stage_I(input[j+i]);
        for (i=0;i<M;i++)
                temp[i]=stage_II(temp[j+i]);
        for (i=0;i<M;i++)
                output[i]= stage_III(temp[j+i]);
}
```

Balance to reduce memory footprint

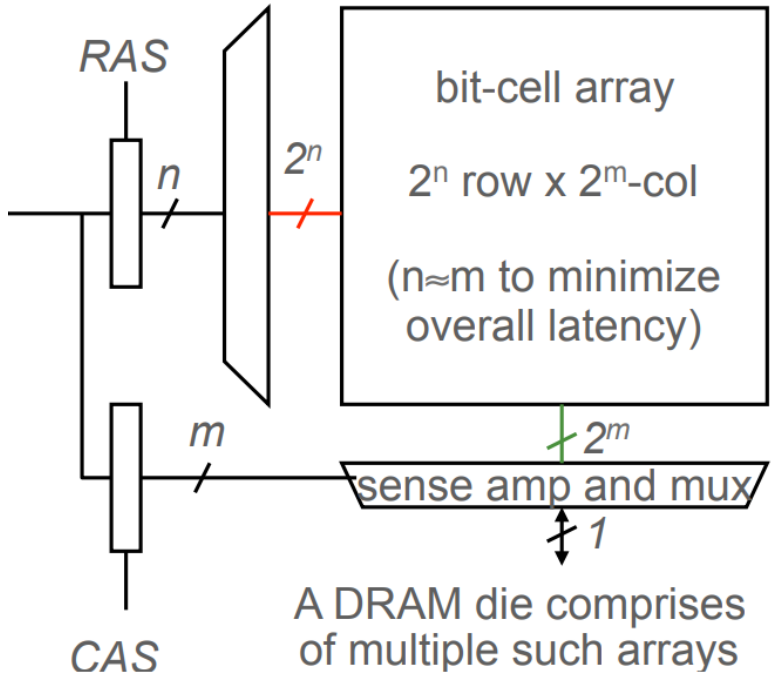
Livadariu et. al., “Optimizing for instruction caches,” EETimes

Questions?

Some Details On DRAM

- DRAM cell access latency is very high
  - Electrical characteristics of the capacitors and the circuitry to read their state
  - To mitigate this, accesses are done at a very coarse granularity
    - Might as well spend 10 ns to read 8 KiB, instead of only 4 bytes

- DRAM is typically organized into a rectangle (rows, columns), called a “bank”
  - Reduces addressing logic, which is a high overhead in such dense memory
  - Whole row must be read whenever data in new row is accessed
  - As of today, typical row size ~8 KB

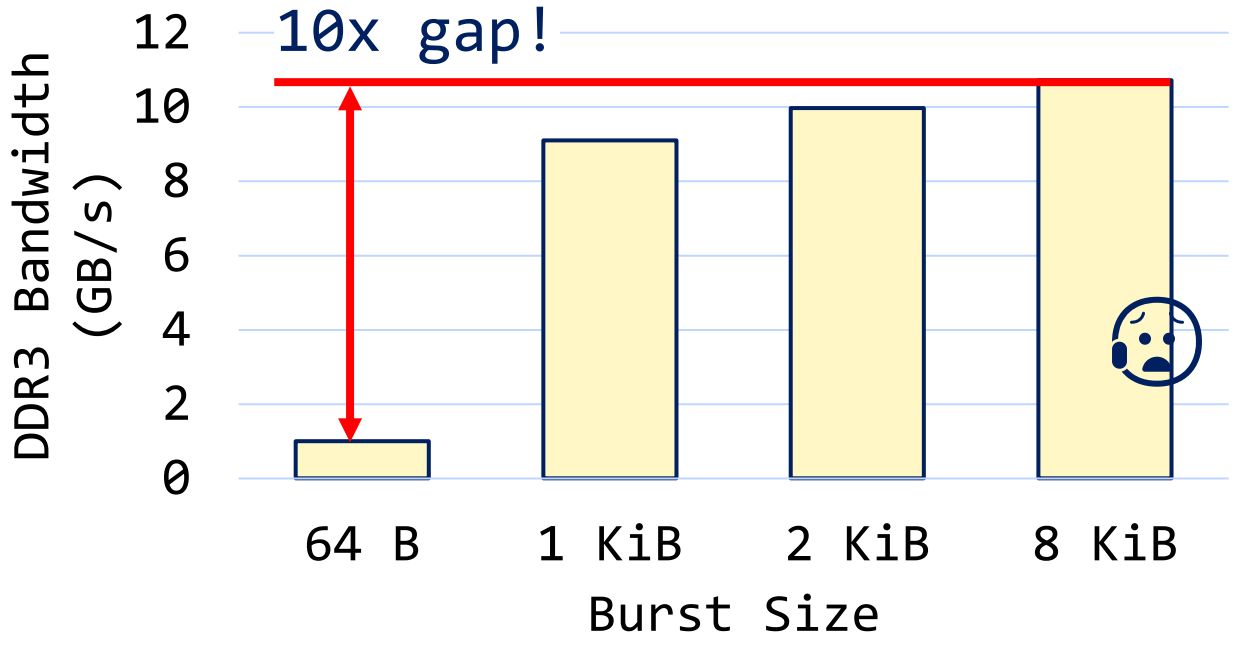


Some Details On DRAM

- Accessed row temporarily stored in DRAM “row buffer”
  - Fast when accessing data in same row
  - Much slower when accessing small data across rows
- The off-chip memory system is also hierarchical
  - A DRAM chip consists of multiple banks
  - A DRAM card consists of multiple chips
  - A memory system (typically) consists of multiple DRAM cards
- Row buffer exists for each bank
  - Total size of all row buffers in a system is quite large
  - Inter-bank parallelism

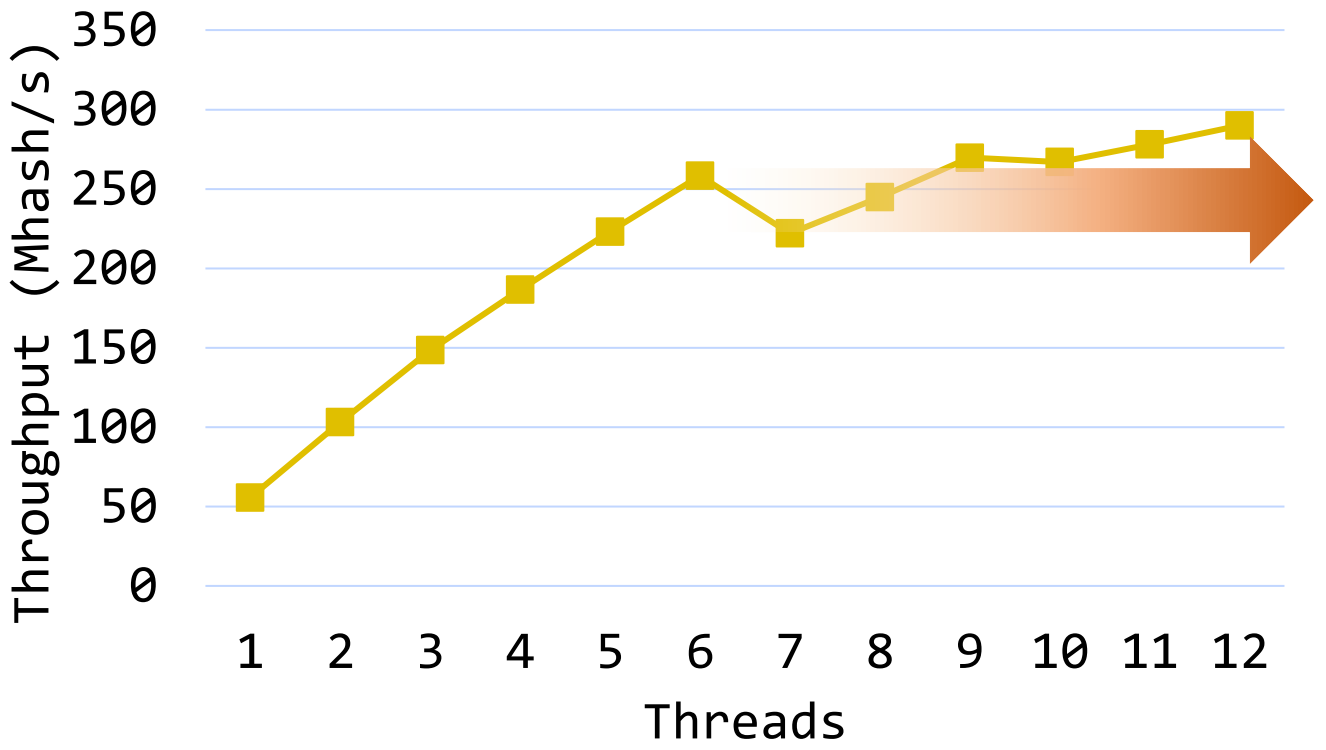
Problem: Random Access on DRAM

- DRAM is not very random-access friendly
  - (Despite the name)
  - Overhead of opening a new bank/row



Problem: Random Access on DRAM

- Random bit updates on DDR4 saturates easily
  - 4x DDR4-2133



Exploiting Inter-Bank Parallelism

- Ideally, accesses should be grouped within a row
- When this is not possible, access to the same bank must be avoided
  - Access cannot be serviced until the previous (high-latency) access is done
- The processor hardware tries to automatically handle this via address mapping
  - LSB of the address used for column index
  - MSB of the address used for row index
  - Everything in the middle spread across card/chip/bank/…

Questions?