1 Why is DRAM slow?

2 Caches
   - Architecture
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications

3 Caches in multi-processor systems
1 Why is DRAM slow?

2 Caches
   - Architecture
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications

3 Caches in multi-processor systems
Why is DRAM slow?

Types or RAM

- **Static RAM (SRAM)**
  - Fast but expensive
  - 6 transistors per bit

- **Dynamic RAM (DRAM)**
  - Capacitor – (Dis)Charging is not instantaneous
  - Reading discharge capacitor (write after read)
  - Compromise: capacity/size/power consumption
Why is DRAM slow?

**DRAM access**

- Addressing individual cells is impractical (many wires)
- Chip is organized in rows and columns (and banks), address is multiplexed
- In the chip, row and column multiplexors select the proper lines according to address bits
- Operations happen in parallel in many chips to work with the whole data word (64 bits)
- Writing: New value is put on data, stored when RAS and CAS are selected
  - It takes some time to charge the capacitors
Why is DRAM slow?

**DRAM access details**

- Access protocol is synchronous (SDRAM) – there is a clock signal,
- CLK provided by memory controller (FSB frequency – typ. 800–1600 MHz)
  - Double/Quad-pumped
- Max. speed: $64 \times 8 \times 200 \, \text{MHz} = 12.8 \, \text{GB/s}$
  - Not reachable in reality
- Data sent in bursts!
## Why is DRAM slow?

### JEDEC standard DDR4 module

<table>
<thead>
<tr>
<th>Standard name</th>
<th>Memory clock (MHz)</th>
<th>I/O bus clock (MHz)</th>
<th>Data rate (MT/s)</th>
<th>Module name</th>
<th>Peak transfer rate (MB/s)</th>
<th>Timings, CL-tRCD-tRP</th>
<th>CAS latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR4-1600J*</td>
<td>200</td>
<td>800</td>
<td>1600</td>
<td>PC4-12800</td>
<td>12800</td>
<td>10-10-10 11-11-11 12-12-12</td>
<td>12.5</td>
</tr>
<tr>
<td>DDR4-1600K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13.75</td>
<td></td>
</tr>
<tr>
<td>DDR4-1600L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>DDR4-1866L*</td>
<td>233.33</td>
<td>933.33</td>
<td>1866.67</td>
<td>PC4-14900</td>
<td>14933.33</td>
<td>12-12-12 13-13-13 14-14-14</td>
<td>12.857</td>
</tr>
<tr>
<td>DDR4-1866M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13.929</td>
<td></td>
</tr>
<tr>
<td>DDR4-1866N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>DDR4-2133N*</td>
<td>266.67</td>
<td>1066.67</td>
<td>2133.33</td>
<td>PC4-17000</td>
<td>17066.67</td>
<td>14-14-14 15-15-15 16-16-16</td>
<td>13.125</td>
</tr>
<tr>
<td>DDR4-2133P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14.063</td>
<td></td>
</tr>
<tr>
<td>DDR4-2133R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>DDR4-2400P*</td>
<td>300</td>
<td>1200</td>
<td>2400</td>
<td>PC4-19200</td>
<td>19200</td>
<td>15-15-15 16-16-16 18-18-18</td>
<td>12.5</td>
</tr>
<tr>
<td>DDR4-2400R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13.33</td>
<td></td>
</tr>
<tr>
<td>DDR4-2400U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

1 Why is DRAM slow?

2 Caches
   - Architecture
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications

3 Caches in multi-processor systems
Cache terminology

- **Spatial locality**: accessed memory objects are close to each other
  - Code: inner loops
  - Data: structures (reading of one field is often followed by reading of other files)

- **Temporal locality**: The same data will be used multiple times in a short period of time
  - Code: loops
  - Data: e.g. Digital filter coefficients are accessed every sampling period
Cache terminology

- **Spatial locality**: accessed memory objects are close to each other
  - Code: inner loops
  - Data: structures (reading of one field is often followed by of other files)

- **Temporal locality**: The same data will be used multiple times in a short period of time
  - Code: loops
  - Data: e.g. Digital filter coefficients are accessed every sampling period

- **Cache hit**: memory request is serviced from the cache, without going to higher level memory

- **Cache miss**: opposite of cache hit
  - cold miss, capacity miss, conflict miss
  - true sharing miss, false sharing miss

Cache line eviction: cache line is removed from the cache to make space for new data

Cache replacement policy: LRU, pseudo LRU, random
Cache terminology

- **Spatial locality**: accessed memory objects are close to each other
  - Code: inner loops
  - Data: structures (reading of one field is often followed by of other files)

- **Temporal locality**: The same data will be used multiple times in a short period of time
  - Code: loops
  - Data: e.g. Digital filter coefficients are accessed every sampling period

- **Cache hit**: memory request is serviced from the cache, without going to higher level memory

- **Cache miss**: opposite of cache hit
  - cold miss, capacity miss, conflict miss
  - true sharing miss, false sharing miss

- **Cache line eviction**: cache line is removed from the cache to make space for new data

- **Cache replacement policy**: LRU, pseudo LRU, random
CPU caches – big picture

- All loads/stores go through cache
- CPU $\leftrightarrow$ Cache: fast connection
- Cache $\leftrightarrow$ Main memory: FSB Bus
- It is advantage to have separate caches for instructions and data
Direct-mapped cache
  - simple
Fully associative cache
  - ideal
Set associative cache
  - compromise
Direct-mapped cache

- Each memory location has just one cache line associated.
- Memory locations at multiples of cache size always collide!
Self-evicting of code

void outer_func() {
    for (int i = 0; i < 1000; i++)
        inner_func();
}

void inner_func() {
    // do something
}
Self-evicting of code

Two cache misses every iteration (instruction fetches)!

Solution: Improve code layout by putting related (and hot) functions together.

```c
#include <stdio.h>

void outer_func() {
    for (int i = 0; i < 1000; i++)
        inner_func();
}

void inner_func() {
    // do something
}
```

```c
__attribute__((hot)) void outer_func();
__attribute__((hot)) void inner_func();
```
Cache write policies

**Write-back** “Common” case. Written data is cached for later reuse.
Cache write policies

**Write-back**  “Common” case. Written data is cached for later reuse.

**Write-through**Written data bypass the cache and therefore never evicts other data from the cache. Useful when we know the data will not be needed soon.

```c
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```
Cache write policies

Write-back  “Common” case. Written data is cached for later reuse.

Write-through  Written data bypass the cache and therefore never evicts other data from the cache. Useful when we know the data will not be needed soon.

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

Write-Combining  All writes to the cache line are combined together and written at once. This avoids one memory read, because when the cache line is fully overwritten, there is no point in reading the old value. Write combining is often used for frame buffer memory (e.g. filling screen with a color).
Set associative caches

- Majority of today’s hardware
- Typically 8–16 ways
- Cache replacement policy
Sequential access

```c
char A[65536*1024];
for (rep = 0; rep < REP; rep++)
    for (i = 0; i < WSS; i += 64)
        A[i]++;
```
Random access

```
char A[65536*1024];
WSS = (1<<N)
mask = (1<<N) - 1;
for (rep = 0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
    A[addr]++;
}
```
Random access

```
char A[65536*1024];
WSS = (1<<N)
mask = (1<<N) - 1;
for (rep = 0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
    A[addr]++;
}
```

Core i7-2600, (perf counters not in scale)
Random access

char A[65536*1024];
WSS = (1<<N)
mask = (1<<N) - 1;
for (rep = 0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
    A[addr]++;
}

Core i7-2600, (perf counters not in scale)
Random access

```c
char A[65536*1024];
WSS = (1<<N)
mask = (1<<N) - 1;
for (rep = 0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
    A[addr]++;
}
```

Core i7-2600, (perf counters not in scale)
Random access

char A[65536*1024];
WSS = (1<<N)
mask = (1<<N) - 1;
for (rep = 0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
    A[addr]++;
}
Translation Lookaside Buffer (TLB)

- Caches translation of virtual to physical address
- On TLB miss, page walk has to be performed (2 to 5 levels)
- Intel i7-2600 has 512 L2 TLBs ⇒ $512 \times 4 \text{ kB} = 2 \text{ MB}$
- Improvement: use so called huge pages (1 page = 2 MB, PS=1)

![Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging](image-url)
When a thread is preempted by another thread, it likely evicts some data from the cache.

After preemption ends, the threads experiences a lot of cache misses!

Certain (older) architectures has to flush TLBs when switching address spaces (processes).

- Modern architectures allow tagging TLBs with address space identifier (ASID, PCID, ...)

High-performance software tries to limit preemptions.

- Beware – limiting preemption increases response time!
Data structures and cache friendliness

- Arrays + sequential access – nice
- Dynamically allocated linked lists – depends on memory allocator (probably like random access)
- Search trees – random access
Caches » Data structures and dynamic memory allocations

Data structures and cache friendliness

- Arrays + sequential access – nice
- Dynamically allocated linked lists – depends on memory allocator (probably like random access)
- Search trees – random access

- For most data structures/algorithms, there exist cache-optimized variants.
- These are more tricky than textbook examples.
Dynamic memory allocator (malloc(), new)

- Memory allocators try to maintain spacial locality
- Hard to achieve when heap is fragmented
  - after many new/delete operations
Matrix multiplication
Naive implementation

\[
\begin{align*}
\text{for } (i = 0; i < N; ++i) \\
\text{for } (j = 0; j < N; ++j) \\
\text{for } (k = 0; k < N; ++k) \\
C[i][j] &= A[i][k] \times B[k][j];
\end{align*}
\]
Matrix multiplication
Naive implementation, memory layout

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      C[i][j] = A[i][k] * B[k][j];
Implementation with transposition

double B[N][N];
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        B[i][j] = Bsrc[j][i];

for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        for (k = 0; k < N; ++k)
            C[i][j] = A[i][k] * B[k][j];
Implementation with transposition

\[ A \times B = C \]

\[
\begin{align*}
\text{mem:210 cache hit:178} & \\
\text{mem:210 cache hit:0} & \times \\
\text{mem:210 cache hit:183} & = \\
\text{mem:630 cache hits:361} & \approx 57\%
\end{align*}
\]

double B[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    B[i][j] = Bsrc[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      C[i][j] = A[i][k] * B[k][j];

Performance: naive: 100%, transposed: 23.4%
Tiled implementation

Each “tile” fits into the cache

Performance: 17.3% of naive implementation (9.5% with vectorized operations)
Tiled implementation and L1 cache

for (k1 = 0; k1 < N; k += tile)
  for (j1 = 0; j1 < N; j += tile)
    for (i1 = 0; i1 < N; i += tile)
      for (i = i1; i < i1 + tile; ++i)
        for (j = j1; j < j1 + tile; ++j)
          for (k = k1; k < k1 + tile; ++k)
            C[i][j] += A[i][k] * B[k][j];

No L1 cache hit in B
Two-level tiled implementation

\[
\begin{align*}
A & \quad \text{mem:126 L1 hit:62 L2 hit:94} \\
\times & \quad \text{mem:126 L1 hit:94 L2 hit:110} \\
= & \quad \text{mem:126 L1 hit:63 L2 hit:94} \\
\text{mem:378} & \quad \text{L1 hits:219 \approx 57\%} \\
& \quad \text{L2 hits:298 \approx 78\%}
\end{align*}
\]

```cpp
def multiply_matrixes(A, B, C):
    for (k2 = 0; k2 < N; k2 += tile2)
        for (j2 = 0; j2 < N; j2 += tile2)
            for (i2 = 0; i2 < N; i2 += tile2)
                for (k1 = k2; k1 < k2 + tile2; k1 += tile1)
                    for (j1 = j2; j1 < j2 + tile2; j1 += tile1)
                        for (i1 = i2; i1 < i1 + tile2; i1 += tile1)
                            for (i = i1; i < i1 + tile1; ++i)
                                for (j = j1; j < j1 + tile1; ++j)
                                    for (k = k1; k < k1 + tile1; ++k)
                                        C[i][j] += A[i][k] * B[k][j];
```

- No L1 cache hit in B
Recursive matrix multiplication

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\
A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\
B_{21} & B_{22} \end{bmatrix} = \\
\begin{bmatrix}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{bmatrix} \times \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}
\]

\(N \times N\) multiplication = 8 multiply-add of \((N/2) \times (N/2)\) multiplications
Outline

1 Why is DRAM slow?

2 Caches
   ■ Architecture
   ■ Memory performance characteristics
   ■ Data structures and dynamic memory allocations
   ■ Matrix multiplications

3 Caches in multi-processor systems
In symmetric multi-processor (SMP) systems, caches of the CPUs cannot work independently from each other.

- The maintaining of uniform view of memory for all processor is called “cache coherency”
- If some processor writes to a cache line, other processors have to clean the corresponding cache line from their caches.
- Cache synchronization protocol: MESI(F)
  - A dirty cache line is not present in any other processor’s cache.
  - Clean copies of the same cache line can reside in arbitrarily many caches.
True sharing

- Program is slow because cache line with shared data travel from one core to another.

```cpp
std::atomic_int32_t counter;

void thread_cpu0() {
    while (true)
        counter++;
}

void thread_cpu1() {
    while (true)
        counter++;
}
```
Caches in multi-processor systems

True sharing

- Program is slow because cache line with shared data travel from one core to another.
- Typically, each mutex is shared between CPUs.

```cpp
std::atomic_int32_t counter;

void thread_cpu0() {
    while (true)
        counter++;
}

void thread_cpu1() {
    while (true)
        counter++;
}
```
Caches in multi-processor systems

True sharing

- Program is slow because cache line with shared data travel from one core to another.
- Typically, each mutex is shared between CPUs.
- When that is a problem (too much contention), make locking more fine-grained or change your data structure and/or algorithms to be more cache friendly.

```cpp
std::atomic_int32_t counter;

void thread_cpu0() {
    while (true)
        counter++;
}

void thread_cpu1() {
    while (true)
        counter++;
}
```
All CPUs executing atomic increment of global variable
Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```cpp
// Per-CPU counters
std::atomic_int32_t counter_cpu0;
std::atomic_int32_t counter_cpu1;

void thread_cpu0() {
    while (true)
    {
        counter_cpu0++;  
    }
}

void thread_cpu1() {
    while (true)
    {
        counter_cpu1++;  
    }
}
```
Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```c++
// Per-CPU counters (FIXME: Do not hardcode cache line size)
std::atomic_int32_t counter_cpu0 __attribute__((align(64)));
std::atomic_int32_t counter_cpu1 __attribute__((align(64)));

void thread_cpu0() {
    while (true)
        counter_cpu0++;
}

void thread_cpu1() {
    while (true)
        counter_cpu1++;
}
```
Size matters

- Even though we have terabytes of memory, size of the data structures still matters.
- Only few kilobytes of memory is fast, the rest is slow!