

# B4M36ESW: Efficient software

## Lecture 7: Memory, caches, allocators

Michal Sojka  
sojkam1@fel.cvut.cz



April 10, 2017

# Outline

- 1 Why is DRAM slow?
- 2 Caches
  - Architecture
  - Memory performance characteristics
  - Dynamic memory allocation
  - Matrix multiplications
- 3 Caches in multi-processor systems

# Outline

- 1 Why is DRAM slow?
- 2 Caches
  - Architecture
  - Memory performance characteristics
  - Dynamic memory allocation
  - Matrix multiplications
- 3 Caches in multi-processor systems

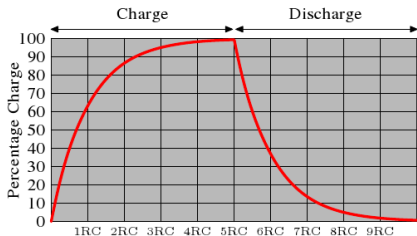
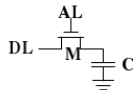
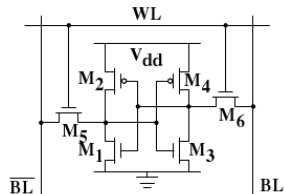
# Types of RAM

## ■ Static RAM (SRAM)

- Fast but expensive
- 6 transistors per bit

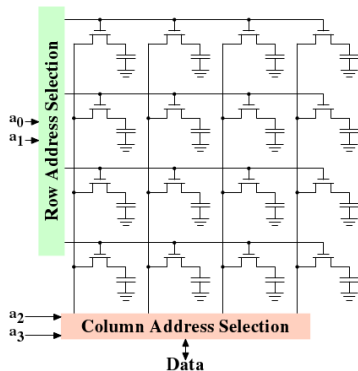
## ■ Dynamic RAM (DRAM)

- Capacitor – (Dis)Charging is not instantaneous
- Reading discharge capacitor (write after read)
- Compromise: capacity/size/power consumption



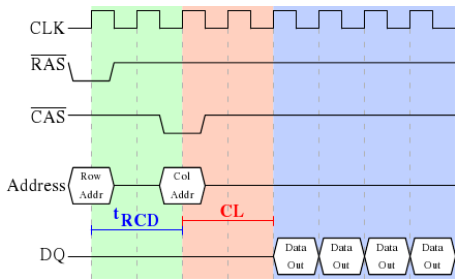
# DRAM access

- 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



# 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:  $64 \text{ bit} \times 8 \times 200 \text{ MHz} = 12.8 \text{ GB/s}$ 
  - Not reachable in reality
- Data sent in bursts!



## JEDEC standard DDR4 module

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

# Outline

- 1 Why is DRAM slow?
- 2 Caches
  - Architecture
  - Memory performance characteristics
  - Dynamic memory allocation
  - 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 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

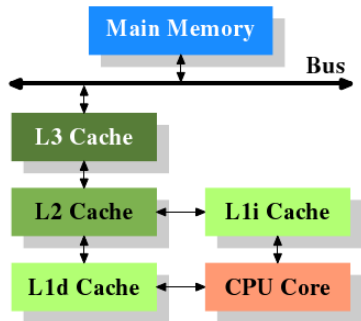
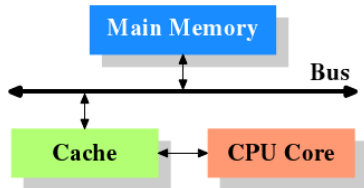
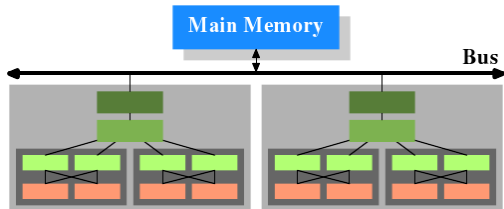
- **Spatial locality:** accessed memory objects are close to each other
  - Code: inner loops
  - Data: structures (reading of one field is often followed by of other files)
- **Temporal locality:** The same data will be used multiple times in a short period of time
  - Code: loops
  - Data: e.g. Digital 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 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

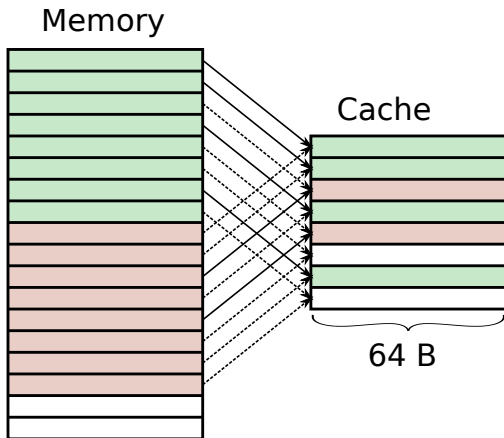
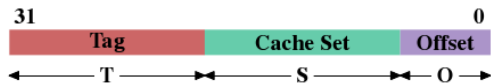
- All loads/stores go through cache
- CPU  $\longleftrightarrow$  Cache: fast connection
- Cache  $\longleftrightarrow$  Main memory: FSB Bus
- It is advantage to have separate caches for instructions and data



# Cache associativity

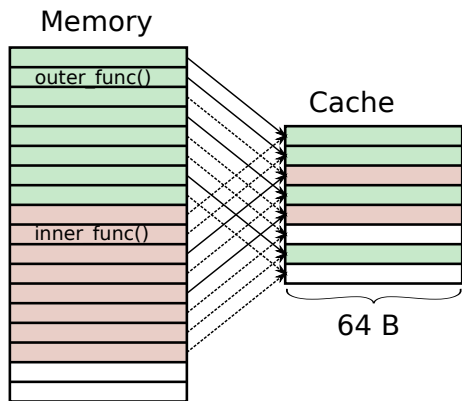
- Direct-mapped cache
  - simple
- Fully associative cache
  - ideal
- Set associative cache
  - compromise

# 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

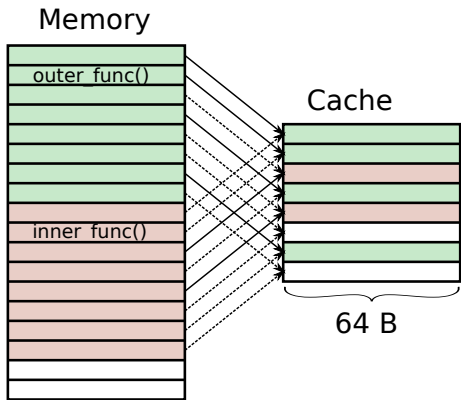


```

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

```

## Self-evicting of code



```
__attribute__((hot)) void outer_func();
__attribute__((hot)) void inner_func();
```

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

- Two cache misses every iteration (instruction fetches)!
- Solution: Improve code layout by putting related (and hot) functions together.



# Write behavior

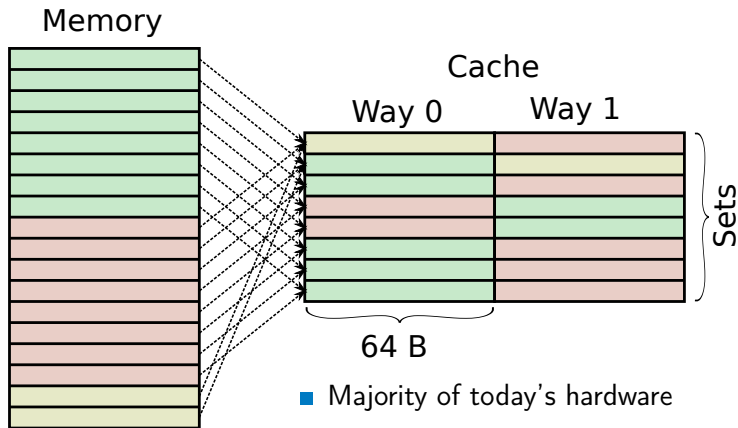
- Cache policies

- Write-through

- Write-back

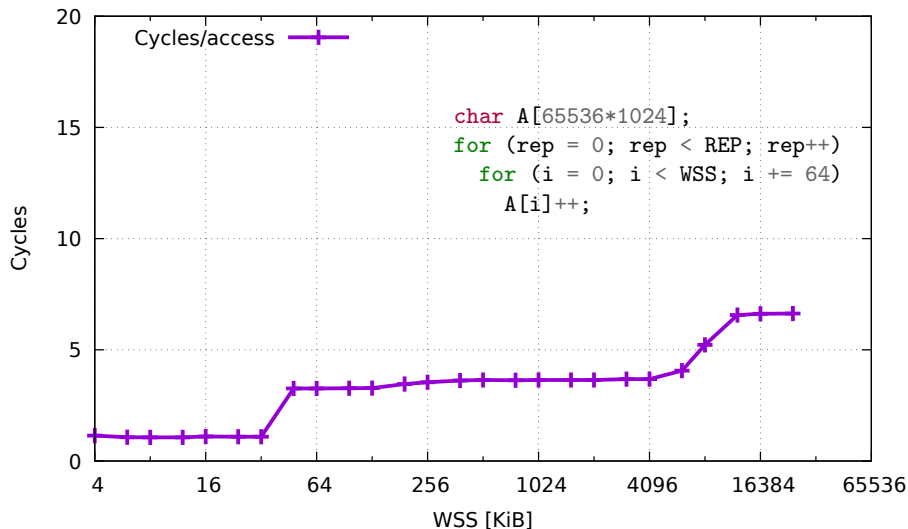
- Write-Combining

# Set associative caches



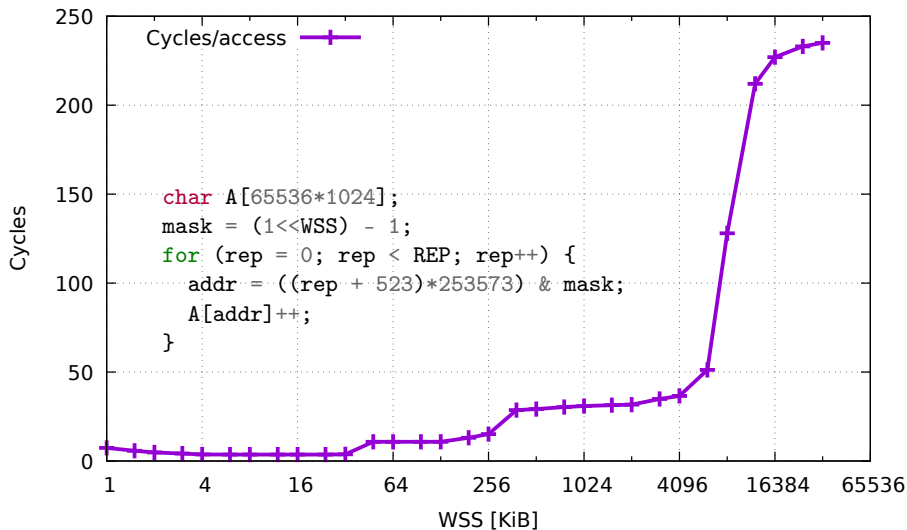
- Majority of today's hardware
- Typically 8–16 ways
- Cache replacement policy

# Sequential access



Intel i7-2600

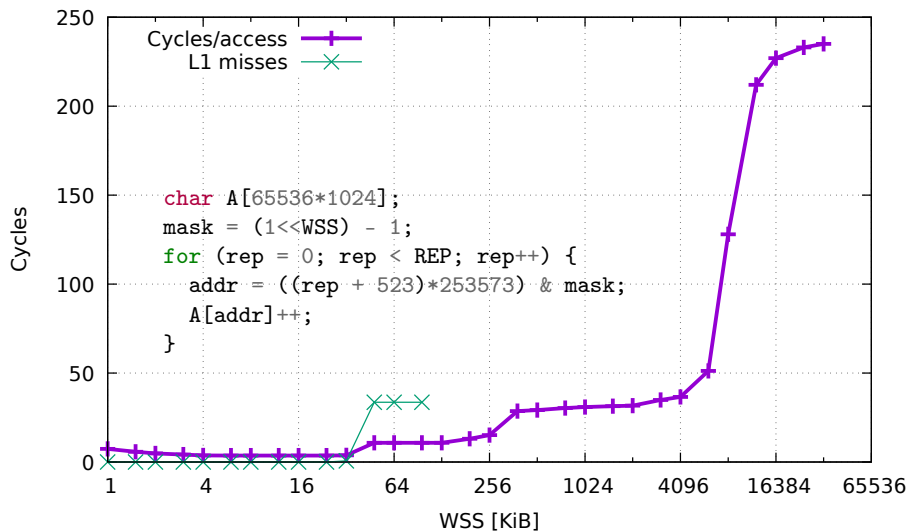
## Random access



Core i7-2600

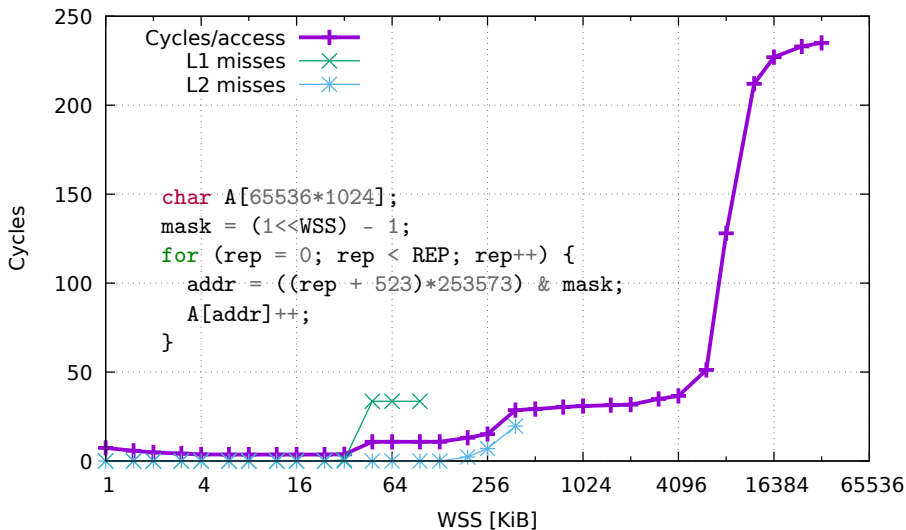
Int

## Random access



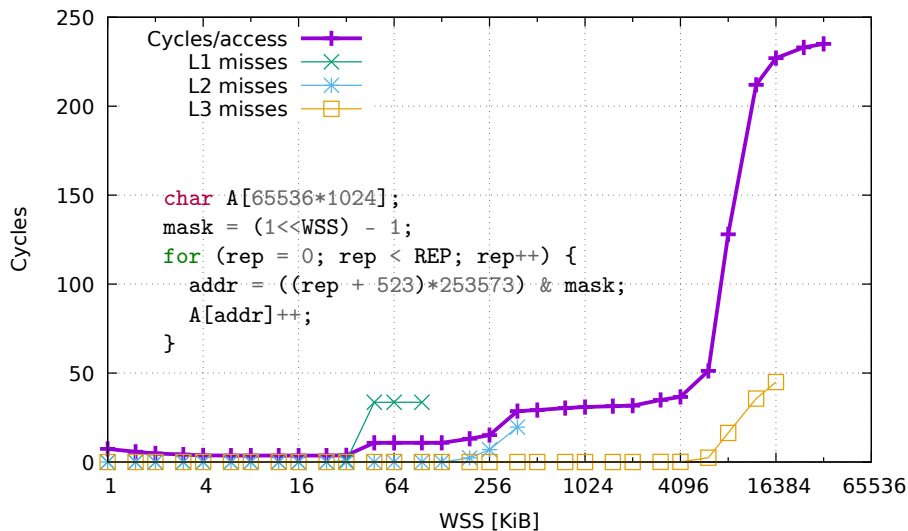
Core i7-2600, (perf counters not in scale)

## Random access



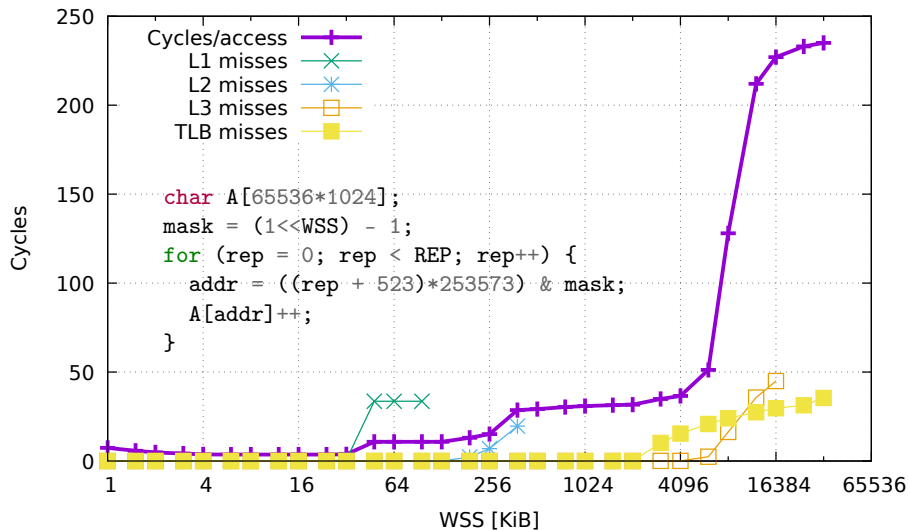
Core i7-2600, (perf counters not in scale)

## Random access



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

## Random access



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

Int



# 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  $\Rightarrow 512 \times 4 \text{ kB} = 2 \text{ MB}$
- Improvement: use so called huge pages (1 page = 2 MB, PS=1)

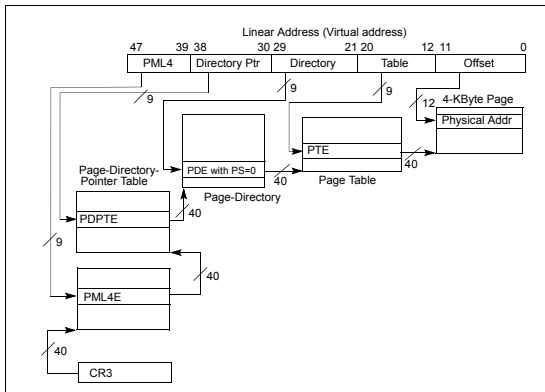


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

# Cache-related preemption delay

- 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!

# malloc, new

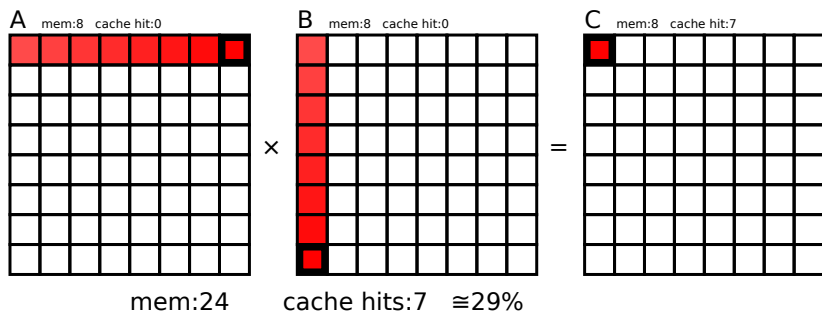
- Memory allocators try to maintain spacial locality
- Hard to achieve when heap is fragmented
  - after many new/delete operations

# 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

# Matrix multiplication

## Naive implementation

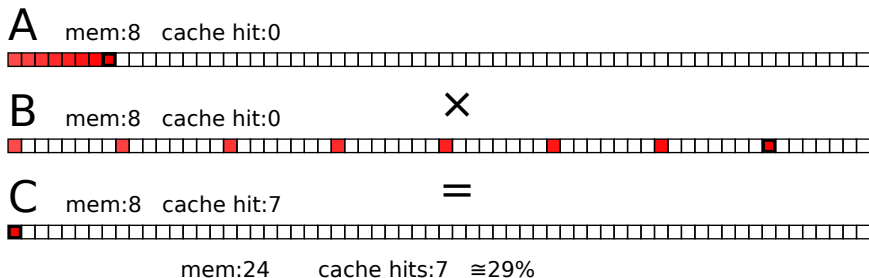


```

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];
  
```

# Matrix multiplication

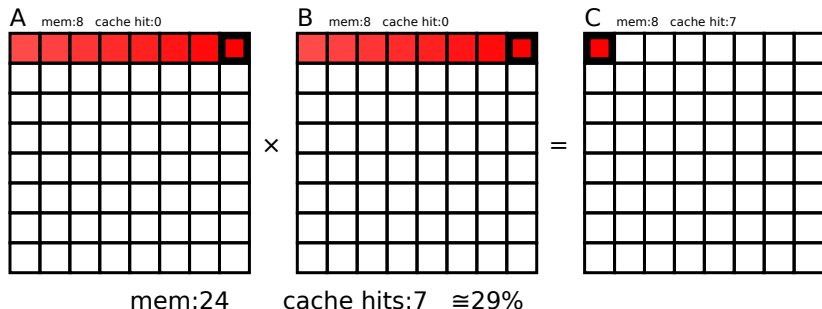
Naive implementation, memory layout



```

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



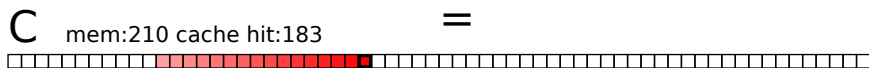
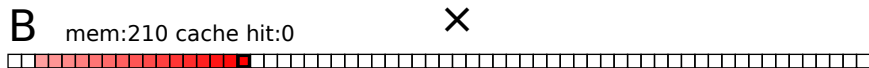
```

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



mem:630 cache hits:361  $\cong 57\%$

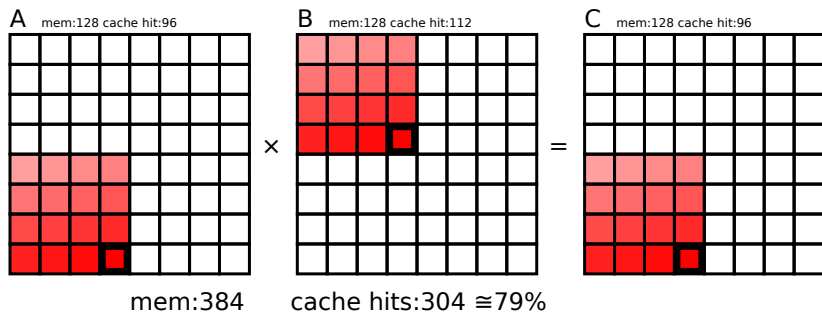
```
double B[N][N];
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        B[i][j] = Bsrc[j][i];

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



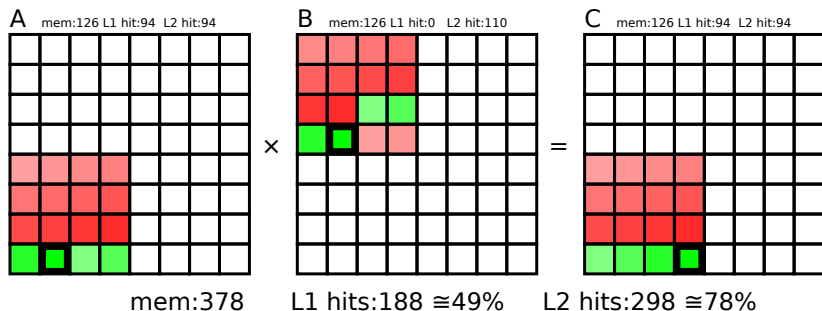
```

for (k1 = 0; k1 < N; k1 += tile)
  for (j1 = 0; j1 < N; j1 += tile)
    for (i1 = 0; i1 < N; i1 += 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];

```

- Each “tile” fits into the cache
- Performance: 17.3% of naive implementation (9.5% with vectorized operations)

## Tiled implementation and L1 cache

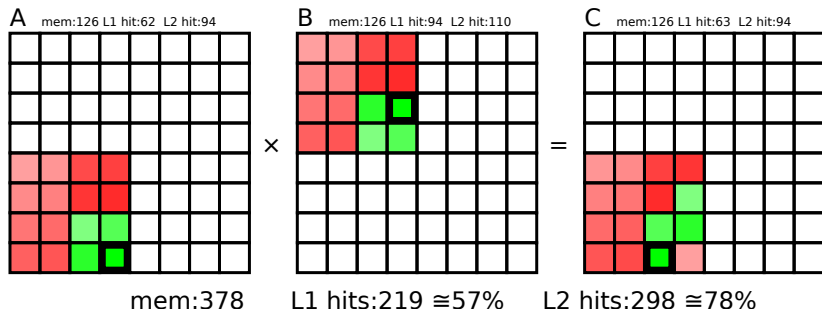


```

for (k1 = 0; k1 < N; k1 += tile)
  for (j1 = 0; j1 < N; j1 += tile)
    for (i1 = 0; i1 < N; i1 += 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



```

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; k += tile1)
        for (j1 = j2; j1 < j2 + tile2; j += tile1)
          for (i1 = i2; i1 < i2 + tile2; i += 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

$$\left[ \begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right] = \left[ \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] \times \left[ \begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right] =$$

$$\left[ \begin{array}{c|c} A_{11}B_{11} & A_{11}B_{12} \\ \hline A_{21}B_{11} & A_{21}B_{12} \end{array} \right] \times \left[ \begin{array}{c|c} A_{12}B_{21} & A_{12}B_{22} \\ \hline A_{22}B_{21} & A_{22}B_{22} \end{array} \right]$$

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

# Outline

- 1 Why is DRAM slow?
- 2 Caches
  - Architecture
  - Memory performance characteristics
  - Dynamic memory allocation
  - Matrix multiplications
- 3 Caches in multi-processor systems

# Cache coherency

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.

```
std::atomic_int32_t counter;
```

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

```
void thread_cpu1() {  
    while (true)  
        counter++;  
}
```

# True sharing

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

```
std::atomic_int32_t counter;
```

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

```
void thread_cpu1() {  
    while (true)  
        counter++;  
}
```



# True sharing

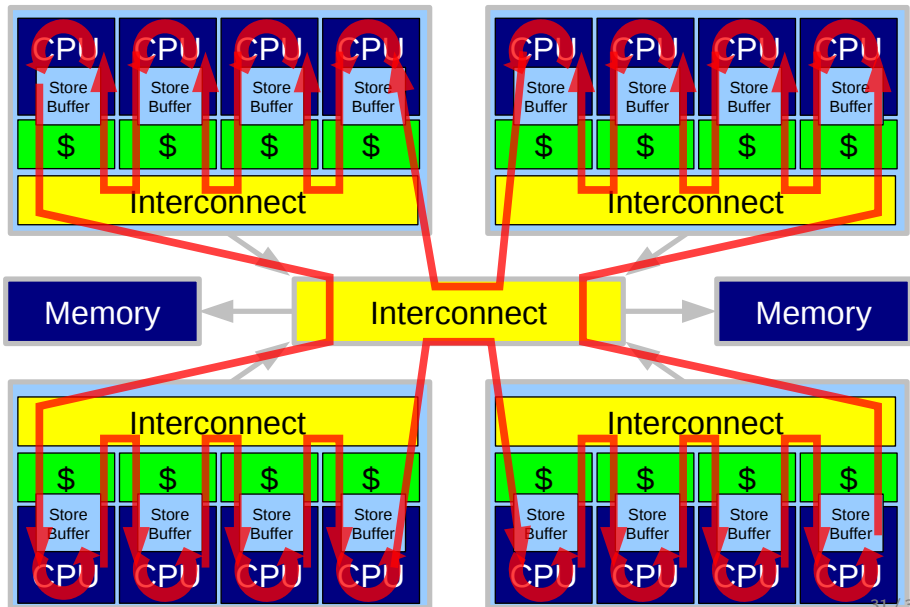
- Program is slow because cache line with shared data travel from one core to another.
- Typically, each mutex is shared between CPUs.
- When that is a problem (too much contention), make locking more fine-grained or change your data structure and/or algorithms to be more cache friendly.

```
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



# False sharing

- Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```
// 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++;
```

```
}
```

# False sharing

- Data accessed from different CPUs is not shared but happen to be stored in a single cache line.

```
// 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++;
}
```

# References

- Ulrich Drepper, “What Every Programmer Should Know About Memory”, 2007/11 [online],  
<http://people.redhat.com/drepper/cpumemory.pdf>