B4M36ESW: Efficient software

Lecture 9: Memory, caches, algorithms

Michal Sojka
`michal.sojka@cvut.cz`

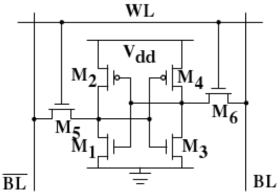
April 19, 2021

Outline

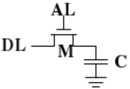
Outline

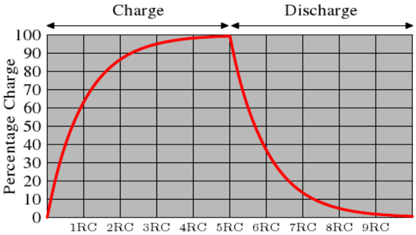
1. Why is DRAM slow?
2. Caches
   - Architecture
     - Cache associativity
     - Cache write policies
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications
3. Caches & memory in multi-processor systems
   - True and false sharing
   - NUMA
4. Conclusion

Types of RAM

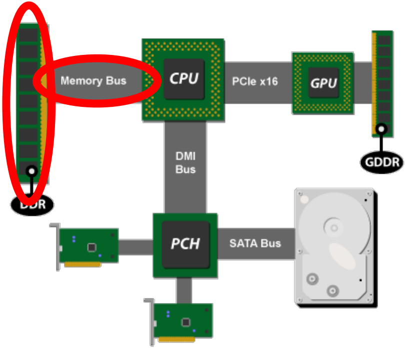


- Static RAM (SRAM)
  - Fast but expensive
  - 6 transistors per bit
- Dynamic RAM (DRAM)
  - Capacitor – (Dis)Charging is not instantaneous
  - Reading discharges capacitor (write after read)
  - Compromise: capacity/size/power consumption





DRAM in the computer

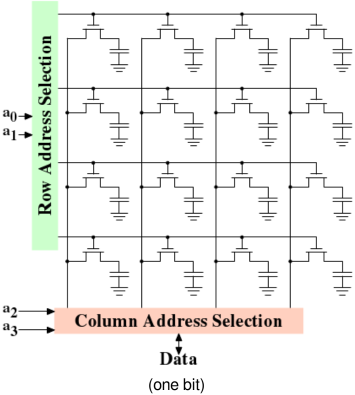


Intel’s P55 platform
Source: ArsTechnica

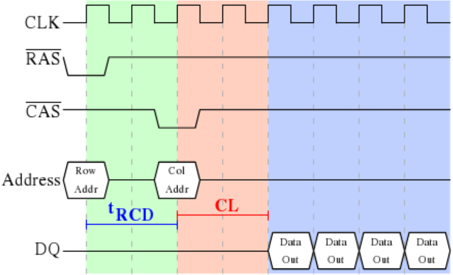
- CPU contains a memory controller (MC)
- MC talks to DRAM chips via “Memory Bus”, using a protocol
- Details on next slides

How DRAM chips work?

- 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 multiplexers (green and pink rectangles) select the lines according to address bits
- R/W operations happen in many chips in parallel to work with the whole data word (64 bits)
- Writing: New value is put on Data signal after row and column address were selected (see next slide)
  - It takes some time to charge the capacitors



SDRAM communication protocol



- Access protocol is synchronous – there is a clock signal
- SDRAM (Synchronous DRAM)
- CLK provided by memory controller (FSB frequency – typ. 800–1600 MHz)
  - Double/Quad-pumped
- Max. speed: 64 bit × 8 × 200 MHz = 12.8 GB/s
  - Not reachable in reality
- DRAM technology requires t_{RCD} and CL delays (they cannot be shortened)
- Data sent in bursts
  - Size of the burst corresponds to cache-line size
  - Sending just one word would be very inefficient due to t_{RCD} and CL delays

Timing parameters of standard DDR4 modules

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

Source: Wikipedia

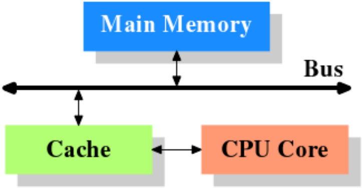
Outline

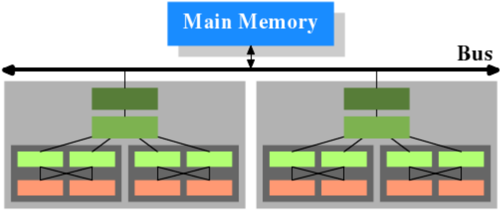
1. Why is DRAM slow?
2. Caches
   - Architecture
     - Cache associativity
     - Cache write policies
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications
3. Caches & memory in multi-processor systems
   - True and false sharing
   - NUMA
4. Conclusion

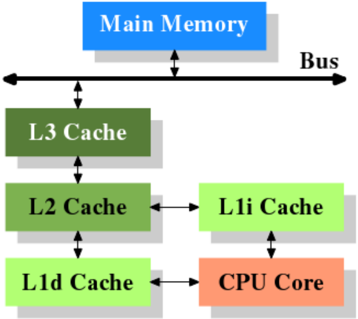
Outline

CPU caches – big picture

- All loads/stores go through cache
- CPU ↔ Cache: fast connection
- Cache ↔ Main memory: FSB bus
- It is an advantage to have separate caches for instructions and data







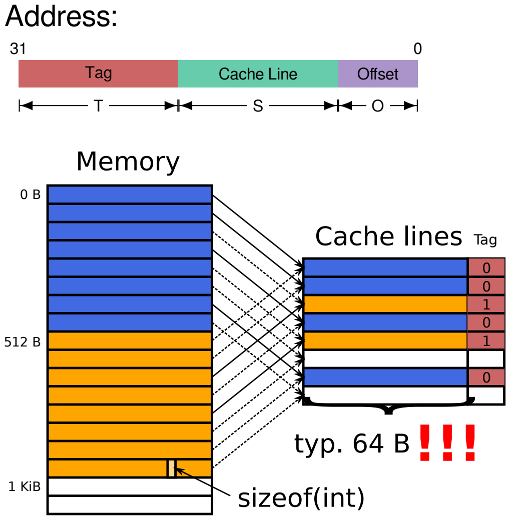
Cache terminology

- Spatial locality: accessed memory objects are close to each other
    - Code: inner loops
    - Data: structures (reading of one field is often followed by reads of other fields)
- 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 (request must go to slow main memory) Multiple possible sources:
    - 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: Least recently used (LRU), pseudo LRU, random, ...

Cache associativity

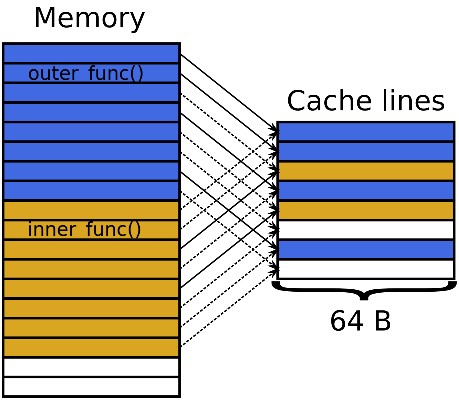
- Direct-mapped cache
  - simple
- Fully associative cache
  - ideal
- Set associative cache
  - compromise

Direct-mapped cache



- Each memory location has just one cache line associated with it
- Memory locations at multiples of cache size always collide!
  Here at multiples of 8×64 bytes.
- Besides the data, cache stores the tag

Problem: Self-evicting of code

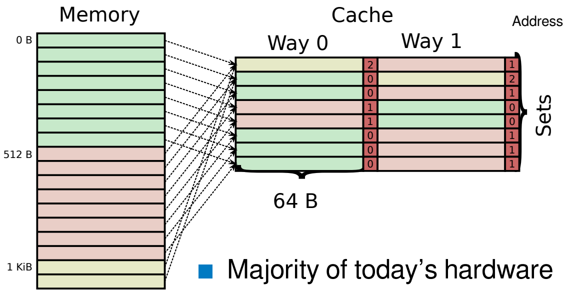


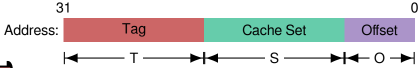
```
void outer_func() {
  for (int i = 0; i < 1000; i++)
    inner_func();
}
void inner_func() {
  // do something
}
```

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

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

Set associative caches





- Majority of today’s hardware
- Typically 8–16 ways
- Two-way set associative cache has approx. the same performance as direct-mapped cache of double size.
- Cache can be seen as hardware hash-table with limited bucket size (limit = the number of ways)
- Cache replacement policy – determines which way is evicted on conflict misses
  - Examples: Least recently used (LRU), Pseudo LRU, random, …

Self-evicting code and set associative caches

- Does the problem of self evicting code exist with set associative caches?
- Yes, but it is less likely to occur.
- Why?

Cache write policies

Goal: Avoid useless eviction of cached data.

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

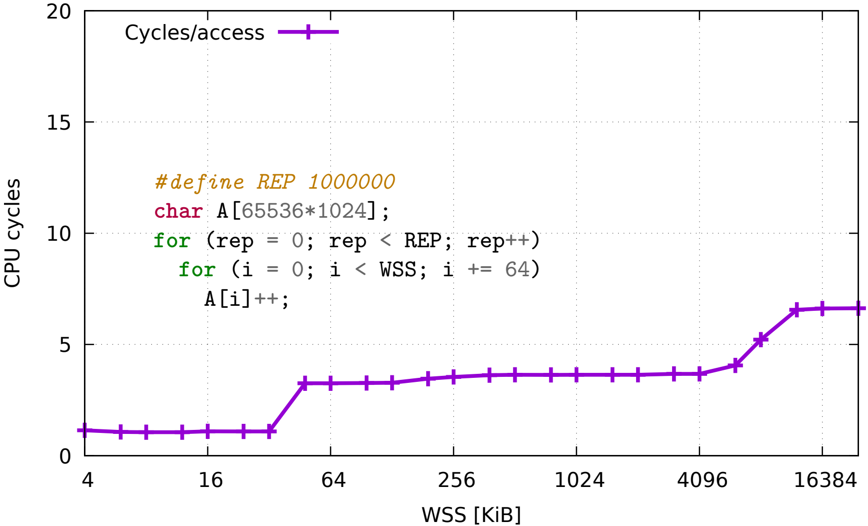
Write-through Written data bypass the cache and therefore never evicts other data from the cache. Useful when you know the data will not be needed soon.

```
#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 <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);
```

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 the screen with a color).

Outline

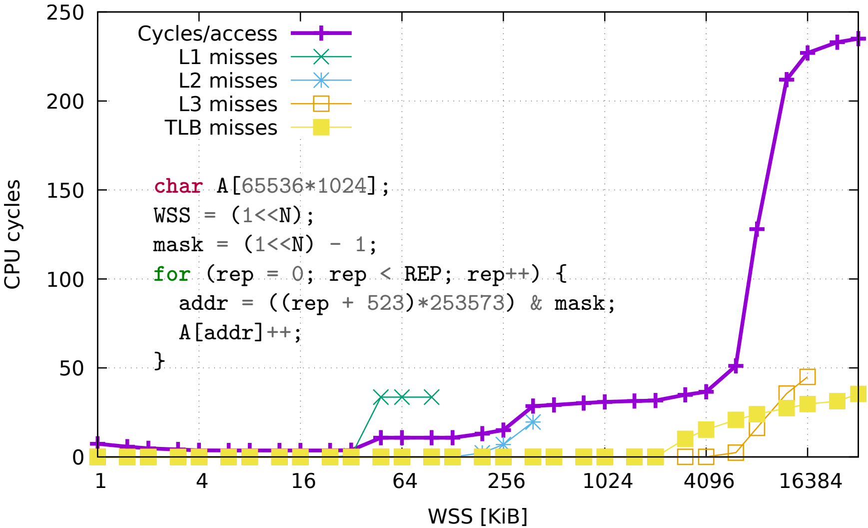
Sequential access



```
#define REP 1000000
char A[65536*1024];
for (rep = 0; rep < REP; rep++)
  for (i = 0; i < WSS; i += 64)
    A[i]++;
```

Intel Core i7-2600

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]++;
}
```

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

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 ⇒ 512×4 kB = 2 MB
- Improvement: use so called huge pages (1 page = 2 MB, PS=1)
  - Linux: in some cases automatically or explicitly via `hugetlbfs`

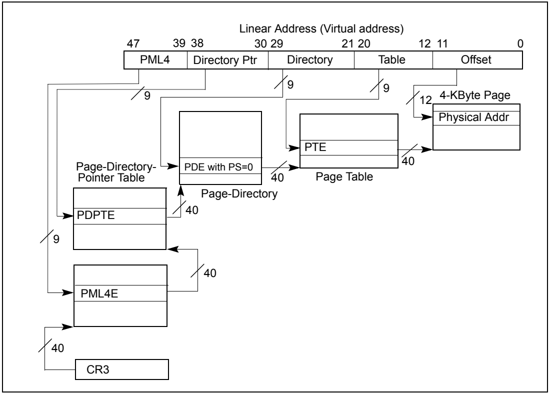


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Cache-related preemption delay

- When a thread is preempted by another thread, the preempting thread likely evicts some data from the cache.
- After preemption ends, the preempted thread continues executing and 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!

Outline

Data structures and cache friendliness

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

- For most data structures/algorithms, cache-optimized variants exist.
- These are more tricky than textbook examples.

Dynamic memory allocator (malloc(), new)

- Memory allocators try to maintain spacial and temporal locality
  - Spatial locality is hard to achieve when heap is fragmented
    - after many new/delete operations
  - Temporal locality – when memory is freed/deleted, subsequent allocation tries to use that memory because it is cache-hot.
- This is based on heuristics. If those heuristics fail for your workload, you should think about writing special allocator for your workload.

Outline

1 Why is DRAM slow?

2 Caches
- Architecture
  - Cache associativity
  - Cache write policies
- Memory performance characteristics
- Data structures and dynamic memory allocations
- Matrix multiplications

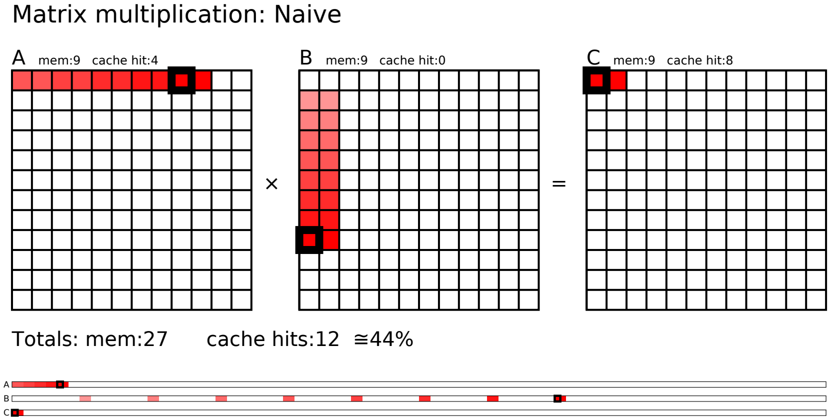
3 Caches & memory in multi-processor systems
- True and false sharing
- NUMA

4 Conclusion

Example of cache-optimized algorithm

1. $C = A \cdot B \quad A = [a_{ij}], \quad i, j = 1 \dots N$
2. $c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$
3. Cache-optimized version is 10× faster than naive implementation

Matrix multiplication – naive implementation

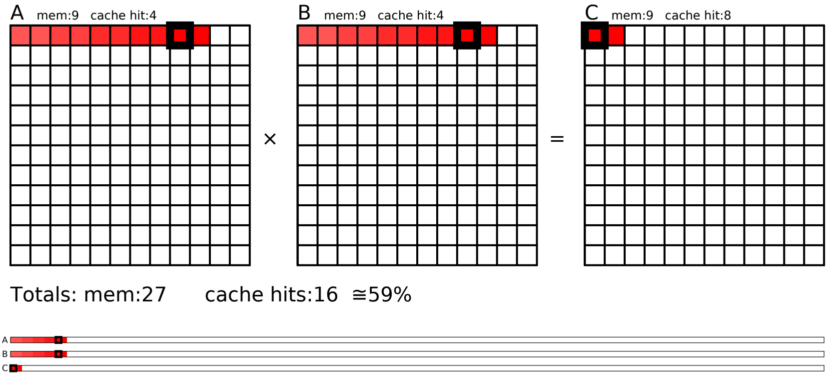


```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

- One matrix element: double (8 B)
- Cache line size: 16 B
- Fully associative caches
- L2 cache: 128 B, L1 cache: 32 B

Implementation with transposition

Matrix multiplication: B transposed



Totals: mem:27 cache hits:16 ≅59%

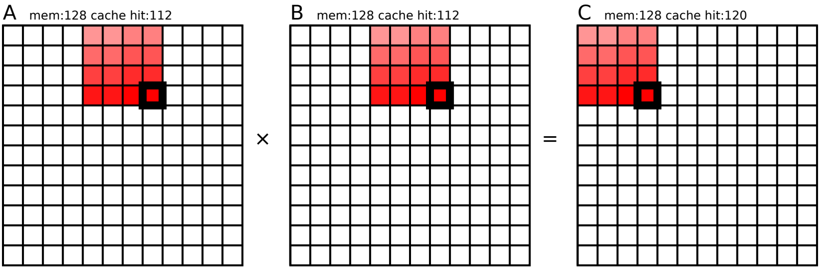
```
double B[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    B[i][j] = Bsrc[j][i];
```

```
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[j][k];
```

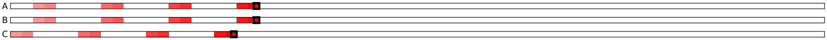
Performance (execution time): naive: 100%, transposed: 23,4%

Tiled implementation

Matrix multiplication: Tiled, B transposed



Totals: mem:384 cache hits:344 ≅89%

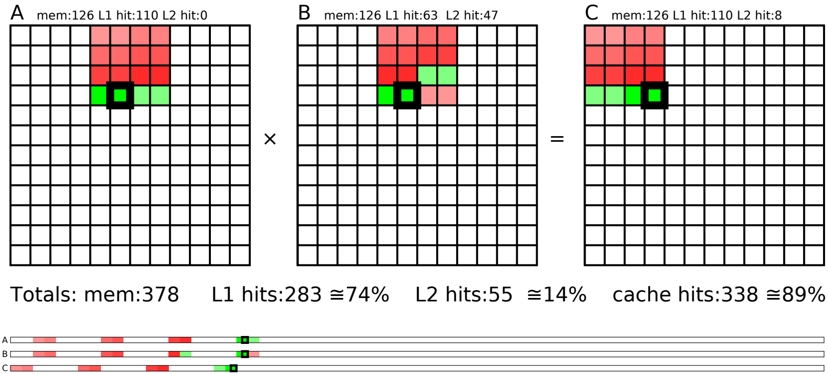


```
for (k1 = 0; k1 < N; k += tile)
  for (j1 = 0; j1 < N; j += tile)
    for (i1 = 0; i1 < N; i += tile)
      for (i = i1; i < i1 + tile; ++i)
        for (j = j1; j < j1 + tile; ++j)
          for (k = k1; k < k1 + tile; ++k)
            C[i][j] += A[i][k] * B[j][k];
```

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

Tiled implementation and L1 cache

Matrix multiplication: Tiled, B transposed



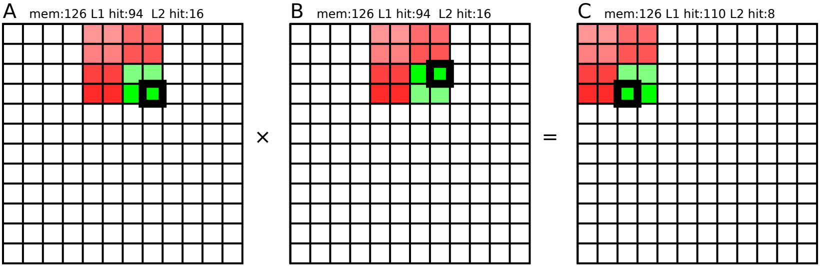
Totals: mem:378 L1 hits:283 ≅74% L2 hits:55 ≅14% cache hits:338 ≅89%

```
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[j][k];
```

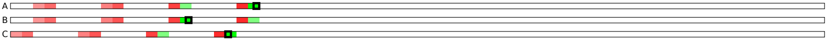
- No temporal L1 cache hit in B
- 75% L1 hits (in total)

Two-level tiled implementation

Matrix multiplication: 2-level tiled, B transposed



Totals: mem:378 L1 hits:298 ≅78% L2 hits:40 ≅10% cache hits:338 ≅89%



```
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[j][k];
```

- 79% L1 hits

Recursive matrix multiplication

- Generalization to arbitrary number of cache levels
- NN multiplication = 8 multiply-add of $(N/2)(N/2)$ multiplications

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

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

Animations

- `https://www.youtube.com/playlist?list=PLB_aWiiTt1af-dICxt6E7pNJWrfcqHE2g`
- Source code: `https://github.com/wentasah/mmul-anim`

Outline

1. Why is DRAM slow?
2. Caches
    - Architecture
        - Cache associativity
        - Cache write policies
    - Memory performance characteristics
    - Data structures and dynamic memory allocations
    - Matrix multiplications
3. Caches & memory in multi-processor systems
    - True and false sharing
    - NUMA
4. Conclusion

Cache coherency

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

- 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.
    - Remember: inter-core (inter-socket) communication is “slow”
- 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.

Cache coherency graphically

Outline

1. Why is DRAM slow?

2. Caches
   - Architecture
     - Cache associativity
     - Cache write policies
   - Memory performance characteristics
   - Data structures and dynamic memory allocations
   - Matrix multiplications

3. Caches & memory in multi-processor systems
   - True and false sharing
   - NUMA

4. Conclusion

True sharing

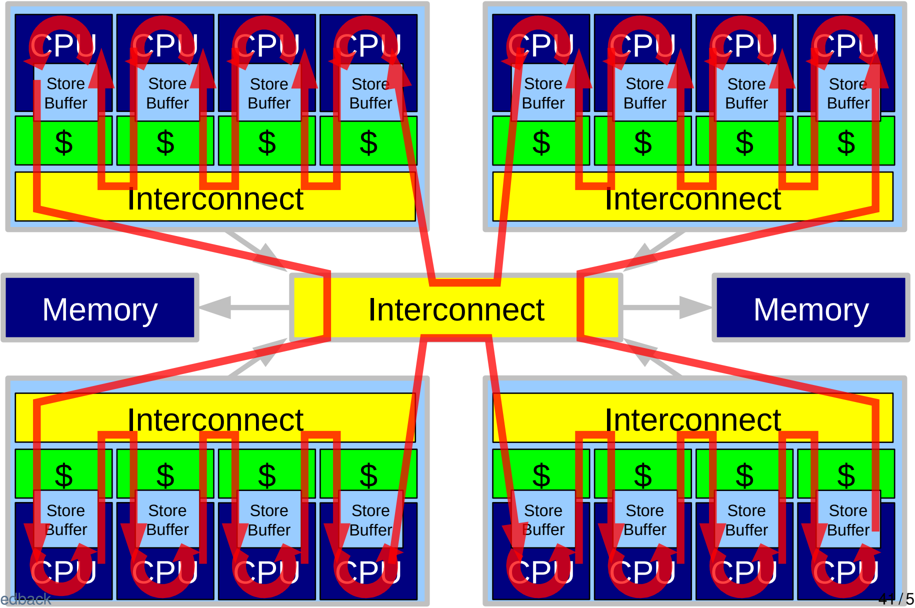
- Program is slow because cache lines with shared data travel from one CPU to another.
- Typical example of true sharing: each mutex is shared between CPUs.
- When that is a problem (too much contention):
  - make locking more fine-grained,
  - or change your data structure (e.g. per-CPU data),
  - and/or 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++;
}
```

- Detecting false sharing in your program:
  - Not visible in the source code!
  - Combining information from several HW performance counters can help
    - That's what Linux's `perf c2c` (cache to cache) subcommand does.

False sharing – solution

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

```
// Per-CPU counters, each alligned to cache line boundary
std::atomic_int32_t counter_cpu0 __attribute__((aligned(64)));
std::atomic_int32_t counter_cpu1 __attribute__((aligned(64)));
```

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

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

- How to determine cache size?
  - at run time: `sysconf(_SC_LEVEL1_DCACHE_LINESIZE);`
  - at compile time:

```
gcc -DLEVEL1_DCACHE_LINESIZE=$(getconf LEVEL1_DCACHE_LINESIZE) ...
```

Example

Code from RCU assignment

Per-thread seed all allocated by the compiler/linker:

```
__thread unsigned int seed;

void gen_rnd_key(char *key, int length)
{
    int len = strlen(charset);
    for (int i = 0; i < length; i++) {
        int r = rand_r(&seed);
        key[i] = charset[r];
    }
    key[length] = '\0';
}
```

__thread also ensures that each seed variable is located in a different cache line.

Per-thread seed all allocated by the programmer (false sharing):

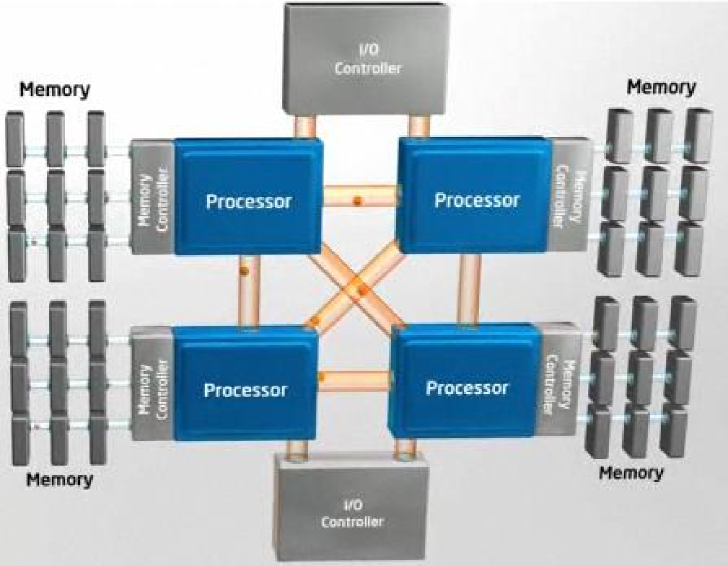
```
unsigned int seed[NUM_CPUS];

void gen_rnd_key(char *key, int length, int thread_id
{
    int len = strlen(charset);
    for (int i = 0; i < length; i++) {
        int r = rand_r(&seed[thread_idx]);
        key[i] = charset[r];
    }
    key[length] = '\0';
}
```

This version is about 40% slower on the ritchie server when running on 32 CPUs.

Outline

1. Why is DRAM slow?
2. Caches
    - Architecture
        - Cache associativity
        - Cache write policies
    - Memory performance characteristics
    - Data structures and dynamic memory allocations
    - Matrix multiplications
3. Caches & memory in multi-processor systems
    - True and false sharing
    - NUMA
4. Conclusion

Non-Uniform Memory Access (NUMA)



Thread migrations between cores

- OSes tend to do load balancing
  - By default threads are automatically migrated from overloaded to underloaded cores
  - Migrated threads loose their cache footprint (cache-related migration delay)
  - Migrated threads loose their NUMA locality
- If you do your own load balancing in the application, pin the threads to CPUs (set their CPU affinity):

```
cpu_set_t cpuset;
pthread_t thread;
thread = pthread_self();
/* Set affinity mask to include only CPU 3 */
CPU_ZERO(&cpuset);
CPU_SET(1 << 3, &cpuset);
s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
```

libnuma (Linux)

```
#include <numa.h>

int numa_available(void);

int numa_max_possible_node(void);
int numa_num_possible_nodes();

int numa_max_node(void);
//...
int numa_preferred(void);
void numa_set_preferred(int node);
void numa_set_interleave_mask(struct bitmask *nodemask);
//...
void numa_bind(struct bitmask *nodemask);
void numa_set_localalloc(void);
void numa_set_membind(struct bitmask *nodemask);
```

Outline

Conclusion

- Size matters
    - Even though we have terabytes of memory, size and layout of the data structures still matters.
    - Only few kilobytes of memory is fast, the rest is slow!
- Cache optimized algorithms can be 10–100× faster than naive implementations.
- When you profiler reports a lot of cache misses and you don't see any shared data, check for false sharing.

References

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