Optimization and profiling

Matteo Caldana

04/05/2023

CPU vs. RAM

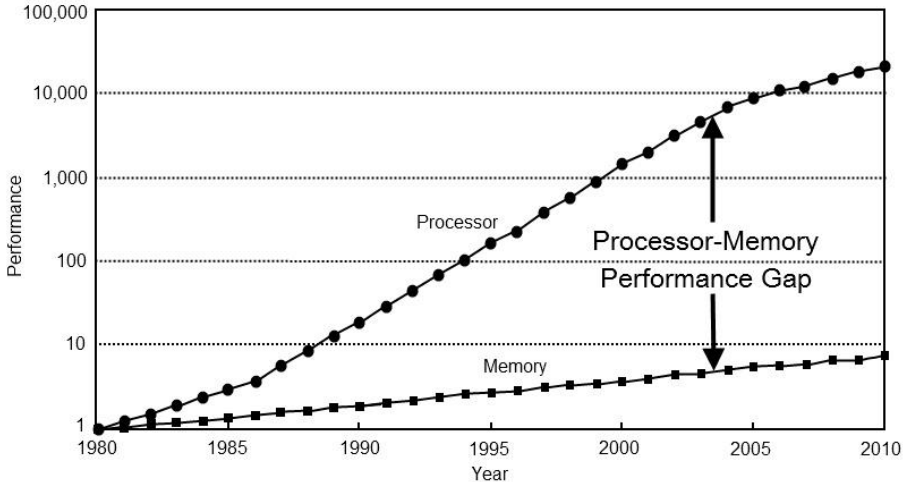


Figure: *Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau*

Memory layout

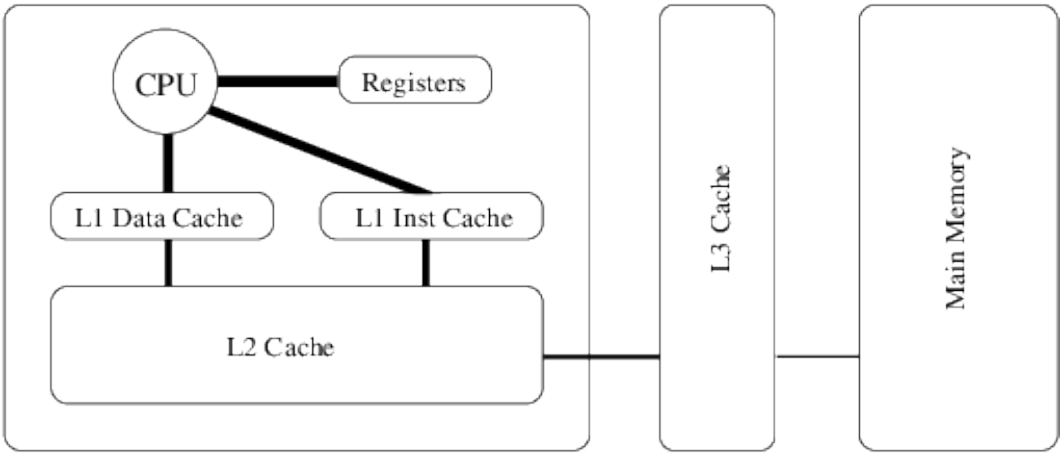
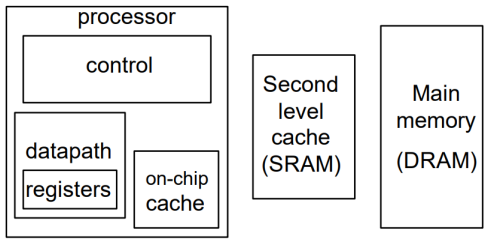


Figure: Typical memory layout of a computer.

Cache miss



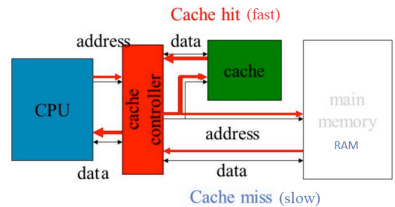


Figure: Typical access times to memory. [1]

[1] See also: https://en.wikipedia.org/wiki/Cache_performance_measurement_and_metric, https://en.wikipedia.org/wiki/Cache_hierarchy

Exercise 1 - How memory access affects performance

This exercise is inspired by this post.

The source file `01-memory-access/01-base.cpp` implements a very simple algorithm, where a `std::vector` of size equal to an integer multiple of the cache memory is filled with random numbers and applied some simple mathematical operation (function `compute()`) for a specified amount of iterations.

Compare the performance of the following three ways of accessing elements in the vector:

1. sequential value access;
2. sequential pointer access;
3. random pointer access.

Exercise 2 - Optimization techniques[3]

1. Implement a function that allocates a `std::vector` and, taking an index as an input, simply returns the corresponding value. Compare the performance with respect to declaring the vector `static`.
2. Implement a function that multiplies all elements in a `std::vector` by looping over all its elements and returns the result. Compare the performance with respect to rewriting the loop using unrolling. That is, at each iteration multiply together five elements of the vector [2].
3. Optimize the memory occupation of an object of class `Class1` by properly aligning/padding the data structure.

[2] See also: `https://en.wikipedia.org/wiki/Loop_unrolling`
[3] See also: `https://github.com/ArtemKovera/code-optimization-techniques`

Data structure alignment[4]

```
class MyClass
{
  char a;       // 1 byte.
  short int b; // 2 bytes.
  int c;        // 4 bytes.
  char d;       // 1 bytes.
}
```

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	b	b	c
c	c	c	d

Figure: How data is **not** stored.

[4] See also: https://www.geeksforgeeks.org/data-structure-alignment/

Data structure alignment[4]

```
class MyClass
{
  char a;       // 1 byte.
  short int b; // 2 bytes.
  int c;        // 4 bytes.
  char d;       // 1 bytes.
}
```

Size of 1 block = 1 byte

Size of 1 row = 4 byte

a	padding	b	b
c	c	c	c
d	padding	padding	padding

Figure: How data is actually stored.

[4]See also: https://www.geeksforgeeks.org/data-structure-alignment/

Exercise 3 - Operating with matrices

Starting from the provided implementation of the class for dense matrices (and column vectors represented as 1-column matrices) based on `std::vector`, implement the following methods:

- `transpose()`: $A = A^T$.
- `operator*`: matrix-matrix and matrix-vector multiplication.

Exercise 3 - What you need to know

- The implemented matrix class is organized as **column-major**, *i.e.* $A(i, j) =$ `data[i + j * n_rows()]`, conversion from 1D to 2D indexing is performed by the utility method `sub2ind`.
- Access to elements is implemented both in `const` and non-`const` versions, by overloading `operator()` and the `value` method.
- Data is private, *getter methods* expose what is needed to the user, both `const` and non-`const` versions are provided.
- Naive implementation of matrix-matrix multiplication is slow because it has low *data locality*, simply transposing the left matrix factor improves performance significantly[5].

[5] See also: *M. Kowarschik, C. Weiß. (2002). Lecture Notes in Computer Science. 213-232. DOI: 10.1007/3-540-36574-5_10* for further details.

Matrix-matrix multiplication: access patterns

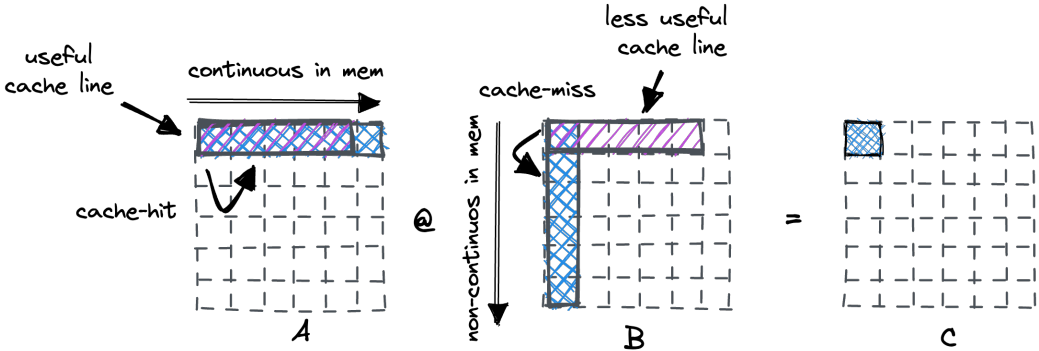


Figure: Access patterns for a **row-major** data structure, assuming that the cache can store 5 elements at once. If elements are accessed in column-wise order (blue pattern in matrix B) then the loop is **not** cache-friendly: at the second iterations the purple elements, that are already in the cache since their memory location is contiguous to the first one, are discarded: thus new entries must be re-cached (very expensive!). Analogous considerations hold for **column-major** data structures. Credits to: `https://siboehm.com/articles/22/Fast-MMM-on-CPU`

Matrix-matrix multiplication: cache aware

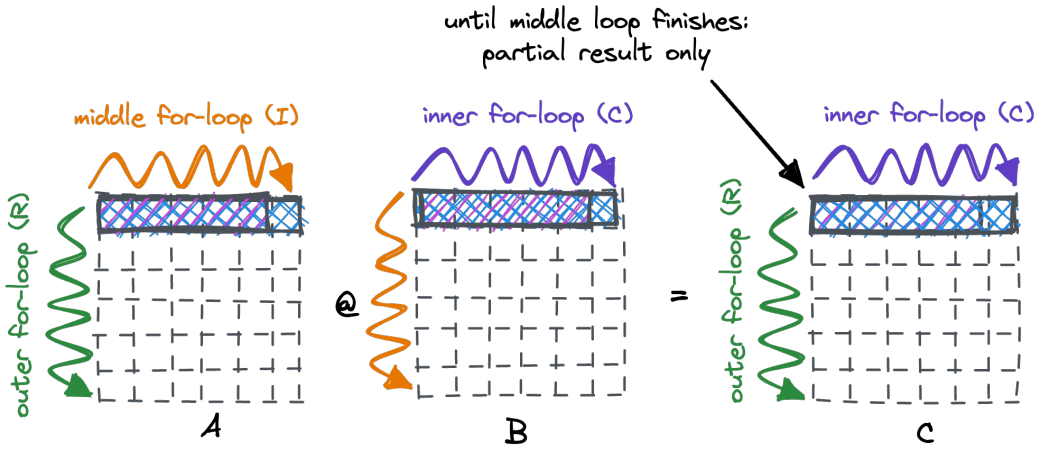


Figure: By switching the two most inner for loops we have a cache-aware algorithm. Credits to: `https://siboehm.com/articles/22/Fast-MMM-on-CPU`

Matrix-matrix multiplication: loop tiling

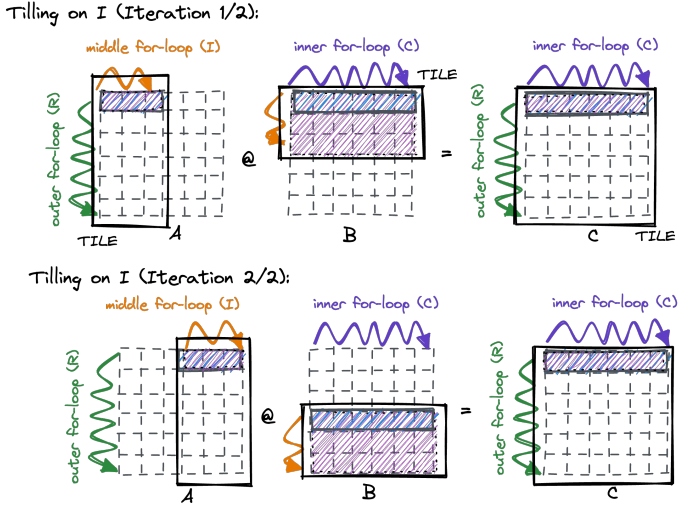


Figure: Loop tiling. Credits to: `https://siboehm.com/articles/22/Fast-MMM-on-CPU`

Exercise 3.2

- Transpose the first factor in matrix multiplication before performing the product.
- Compare the execution speed with respect to the previous implementation.
- Generate a coverage report using `lcov` and a profiling check using `valgrind` or `gprof`.

Homework

- Implement the cache-aware and loop-tiling algorithm.
- Compare the execution speed with respect to the previous implementation.

Exercise 3.3

- Include the `Eigen/Dense` header. Use the `Eigen::Map` template class to wrap the matrix data and interpret it as `Eigen::MatrixXd`.
- Include the `cblas.h` header. Use `cblas_dgemm` function to multiply the two matrices.
- Compare the execution speed with respect to the previous implementations.