Optimization and profiling

Matteo Caldana

04/05/2023

CPU vs. RAM

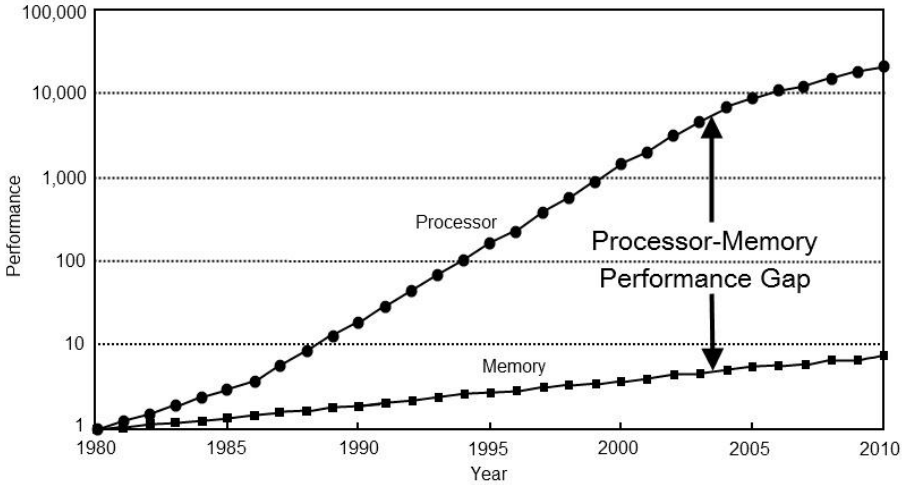

Figure: *Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau*

Memory layout

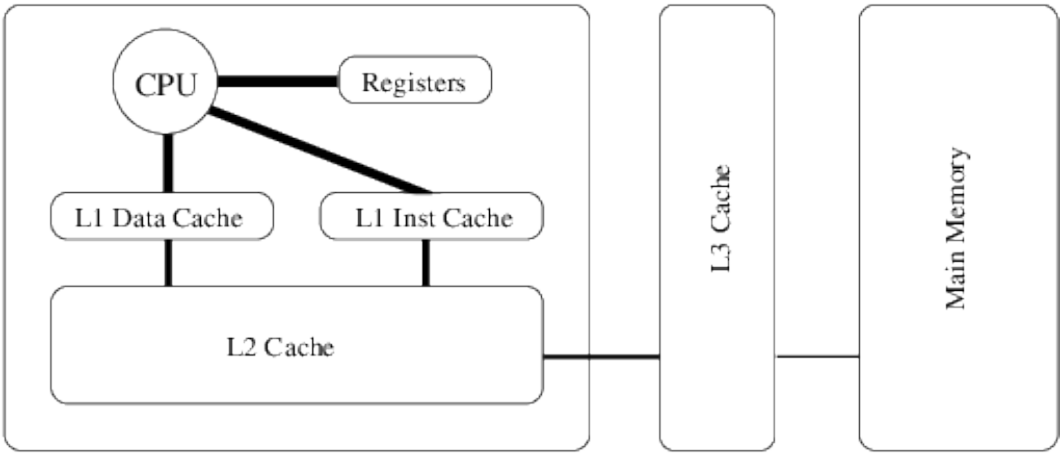

Figure: Typical memory layout of a computer.

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[1]

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.
3. Optimize the memory occupation of an object of class `Class1` by properly aligning/padding the data structure.

[1] See also: `https://github.com/ArtemKovera/code-optimization-techniques`

Data structure alignment[2]

```
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

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

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

Data structure alignment[2]

```
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.

[2]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()`.
- 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[3].
- The `#include "test_matrix_mult.hpp"` header provides timing utilities, `tic()` and `toc(x)` macros start and stop the timer.

[3] 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

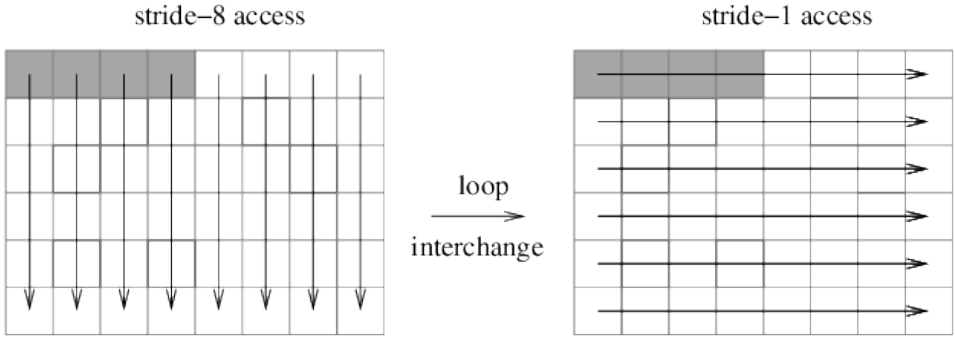

Figure: Access patterns for a **row-major** data structure, assuming that the cache can store 4 elements at once. If elements are accessed in column-wise order (left) then the loop is **not** cache-friendly: at the second iterations (following the arrow) the grey elements, that are already in the cache since their memory location is contiguous to the first one, are discarded: thus new entries – which are 8 memory locations far from the ones already in the cache – must be re-cached. This operation is costly! Vice-versa, if elements are accessed row-wise (*i.e.* following the memory pattern of the original data structure) the second iteration operates on an element (the second grey box) which is already in the cache, and so on. Analogous considerations hold for **column-major** data structures.

Matrix-matrix multiplication: loop tiling

```
1: double sum;
2: double a[n, n];
3: // Original loop nest:
4: for j = 1 to n do
5:     for i = 1 to n do
6:         sum+ = a[i, j];
7:     end for
8: end for
```

```
1: double sum;
2: double a[n, n];
3: // Interchanged loop nest:
4: for i = 1 to n do
5:     for j = 1 to n do
6:         sum+ = a[i, j];
7:     end for
8: end for
```

Figure: Loop tiling.

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`.

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`.
- Compare the execution speed with respect to the previous implementations.