Easylab2: Accelerate matrix multiplication

Du Jiajun

2023.12.13

Problem

There are two matrices `m1` and `m2`, and their shapes are `N*M` and `M*P` respectively. Write the result of matrix multiplication of `m1` and `m2` into the result matrix `r`. Try to speed up matrix multiplication as much as possible.

Step by step

1. Higher cache hit rate

    1.1 Motivation : How to Access Matrix Memory

    1.2 Blocking

    1.3 Array Rearrangement

    1.4 How to Write Result Back

2. Parallel Programming

3. bonus: SSE3 Instruction Set

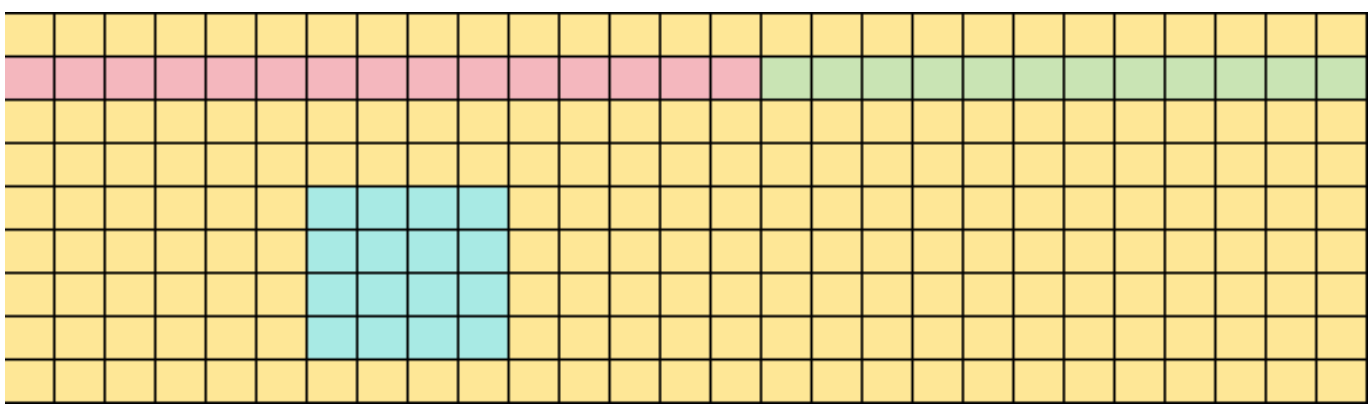
1.1 Motivation: how to access matrix memory

```
for(int row = 0; row < N; ++row)
    for(int col = 0; col < P; ++col)
        for(int mid = 0; mid < M; ++mid)
            r[row][col] += m1[row][mid] * m2[mid][col];
```

Array access jumps from row to row. Adjust loop order to: row -> mid -> col.

```
for(int row = 0; row < N; ++row)
  for(int mid = 0; mid < M; ++mid)
    for(int col = 0; col < P; ++col)
      r[row][col] += m1[row][mid] * m2[mid][col];
```

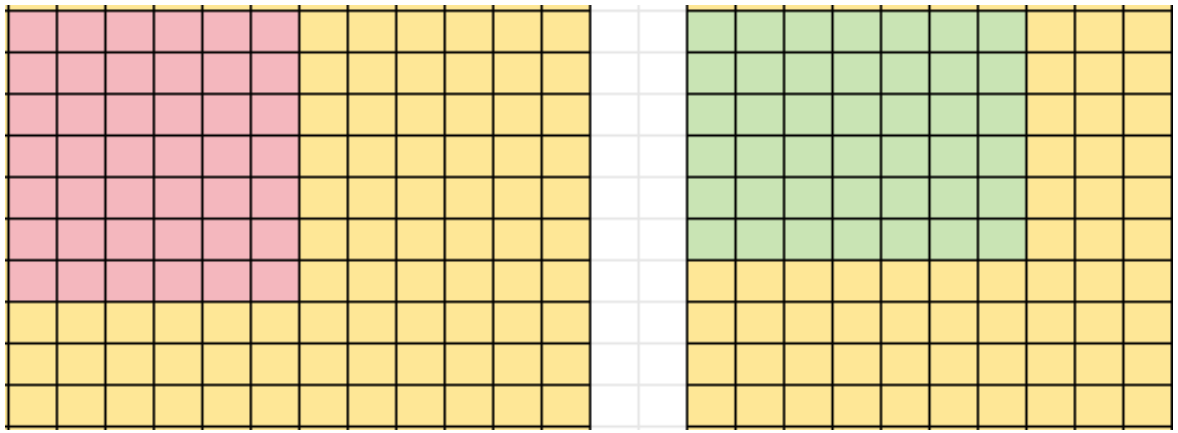
When N、M、P is large, cache misses often occur because memory access is performed in cacheline units. Even two adjacent elements are loaded in tow different cache line, a cache miss will occur. And when the matrix is too large, the cache line that was loaded in last loop will be flushed out of the cache during the next loop. So we need to divide the matrix into small blocks, access the matrix one by one block.



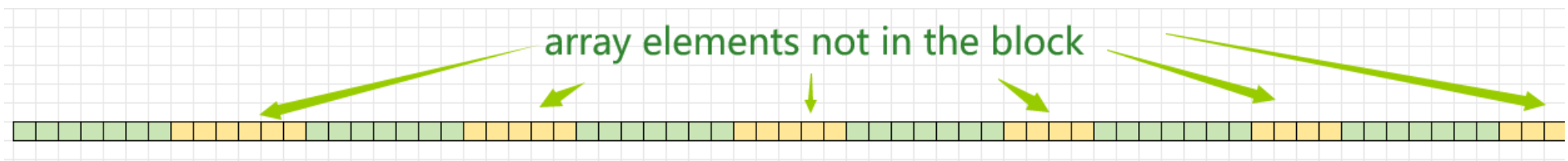
1.2 Blocking

The code looks like:

```
for(int r_o = 0; r_o < N; r_o += N_B)
  for(int m_o = 0; m_o < M; m_o += M_B)
    for(int c_o = 0; c_o < P; c_o += P_B)
      for(int row = r_o; row < min(r_o + N_B, N); ++row)
        for(int mid = m_o; mid < min(m_o + M_B, M); ++mid)
          for(int col = c_o; col < min(c_o + P_B, P); ++col)
            r[row][col] += m1[row][mid] * m2[mid][col];
```



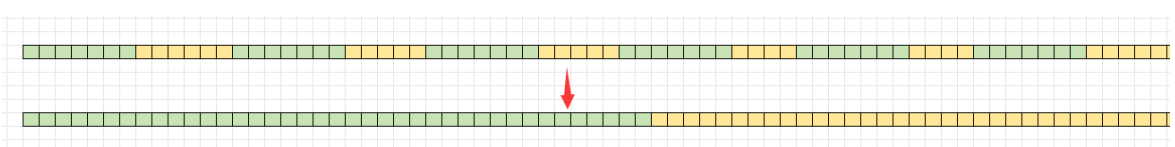
Let's think about how the matrix block on the right is accessed in memory. The real memory is one-dimensional.



Because when accessing between rows, you need to span a distance of `P` size, result in:

- When `P` is larger than the cache line size, the first access to each line in the left and right block will cause a cache miss.
- When there are many rows, every time `row+1` and `mid+1` will cause a cache miss.

So we need to rearrange the array. The memory access pattern becomes like this:



1.3 Array Rearrangement

1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4

1	1	1	1	2	2	2	2
3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2
3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2
3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2
3	3	3	3	4	4	4	4

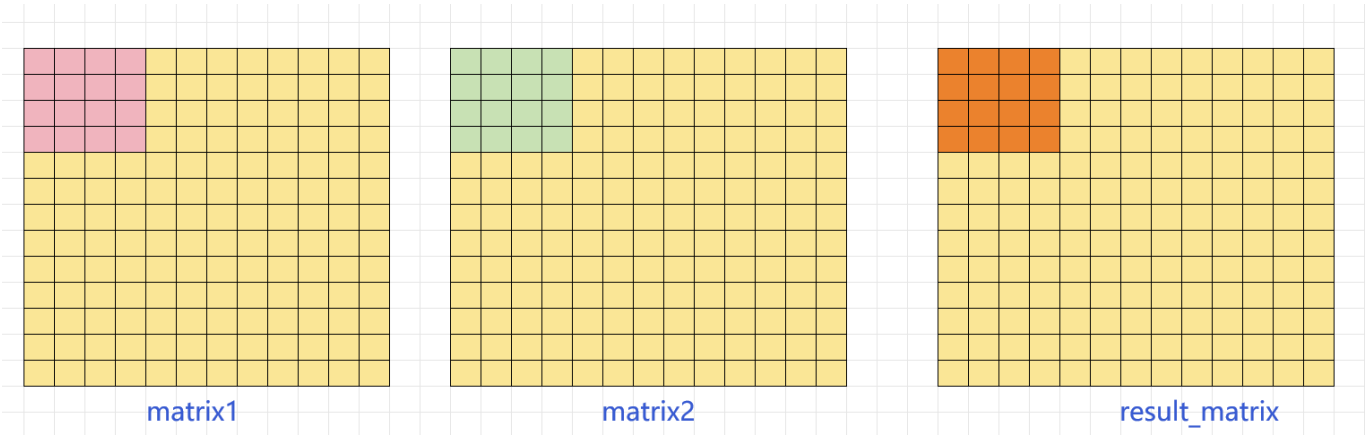
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	1	1	1	1	2	2	2	2	...

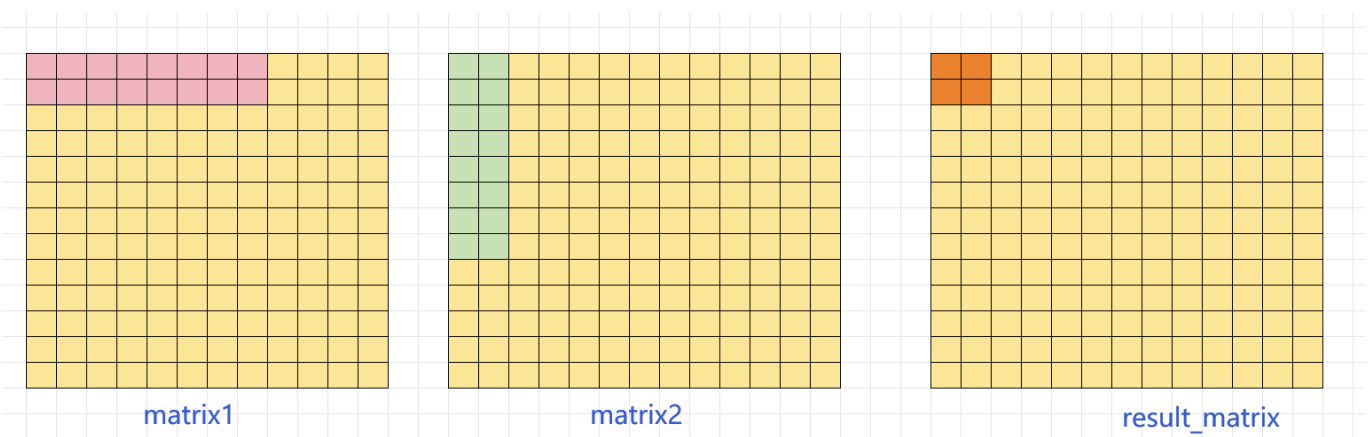
Code will give later.

1.4 How to Write Result Back

Not only do you need to consider the efficiency of reading the two multiplication matrices, but you also need to consider the efficiency of writing back the result matrix.

Let's consider two different cases of blocking.


matrix1
matrix2
result_matrix


matrix1
matrix2
result_matrix

When the blocking size `M_B` is set to be long, but `N_B` and `P_B` are small, the elements need to be written back will be less. And if `P` is divided into blocks and looped in the innermost layer, the blocking method on the right side will also reduce the number of result matrix's cache misses.

So how we blocking our matrix:

- The `N` and `P` dimension block sizes need to be small, and the `M` dimension block size needs to be large. (long and narrow shape)
- The loop of blocking `P` is in the innermost layer.

So how to set the size of blocking

Assume that the block sizes of each dimension are `N_BLOCK`, `M_BLOCK`, and `P_BLOCK` respectively.

- Setting both `N_BLOCK` and `P_BLOCK` to `4` is very appropriate(experience), because in the end only `4*4` elements need to be written back to the result matrix.
- The size of `M_BLOCK` needs to be set according to the number of threads you set. If the number of threads is small, you can set larger, because the cache will most likely not be replaced due to the operations of other threads. Otherwise, you can set `M_BLOCK` smaller, even set the block size to the size of a cache line.

2. Parallel Programming

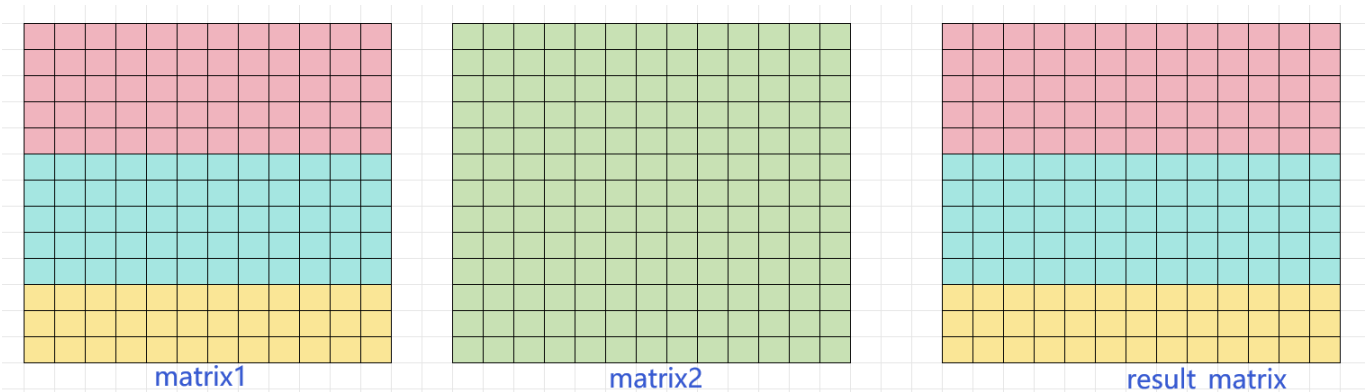
It's easy to use `pthread` and `thread` libraries from the standard library.

We don't need to use thread synchronization tools, such as locks, semaphores, etc.

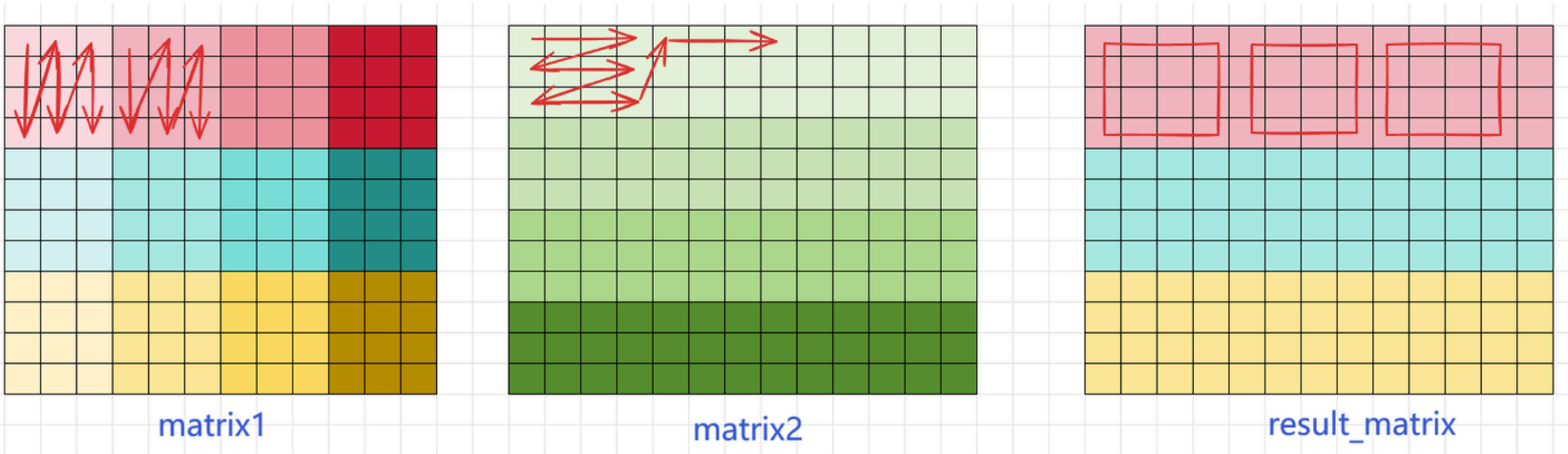
Because there is no conflict when multiple threads read data at the same time and memory place, you just need to ensure that no other threads can read and write when writing.

All writing tasks are on the result matrix, so you only need to divide the writing operations into disjoint sets.

A way to divide tasks into sub-threads:



Just divide matrix1 into several parts by rows, and then multiply them by matrix2. Each sub-thread is equivalent to performing a smaller matrix multiplication, and this ensures that there is no conflict between reading and writing when writing the result matrix.


matrix1
matrix2
result_matrix

When to do array rearrangement

Because matrix1 is divided into multiple small matrices by row and multiply them all by matrix2, the result of matrix2's rearrangement can be reused.

In matrix1, each small block with the size of `N_BLOCK*M_BLOCK` will never be used after multiplication operation with the corresponding part of matrix2, so we can rearrange each small block in matrix1 before performing small block multiplication operations. The rearranged results can be discarded directly because they will not be used in the future.

3. bonus: SSE3 Instruction Set

Some important functions that need to be used:

- _mm_setzero_pd(void): return a vector [0, 0]
- _mm_load_pd(ptr): load two consecutive doubles from the location of address ptr and return them as vectors
- _mm_loaddup_pd(ptr): a macro definition, load a double data from address ptr and set both data fields of vector to this number and return

The vector in SSE3 supports direct use of multiplication and addition, so it is relatively simple.

```c
#include <iostream>
#include <mmintrin.h>
#include <xmmintrin.h>
#include <pmmintrin.h>
#include <emmintrin.h>

typedef union {
    __m128d vector;
    double data[2];
} sse_vector;

int main()
{
    double nums[] = {
        1.2, 2.2, 3.0, 4.0
    };

    sse_vector first, second, third, sum;
    // set to [1.2, 2.2]
    first.vector = _mm_load_pd(nums);
    // set to [3.0, 3.0]
    second.vector = _mm_loaddup_pd((nums + 2));
    // set to [4.0, 4.0]
    third.vector = _mm_loaddup_pd((nums + 3));
    // set to zero
    sum.vector = _mm_setzero_pd();

    sum.vector += first.vector * second.vector;
    sum.vector += second.vector * third.vector;

    // 15.600000, 18.600000
    printf("%f, %f\n", sum.data[0], sum.data[1]);
}
```

Code Example

Read code: code.png.