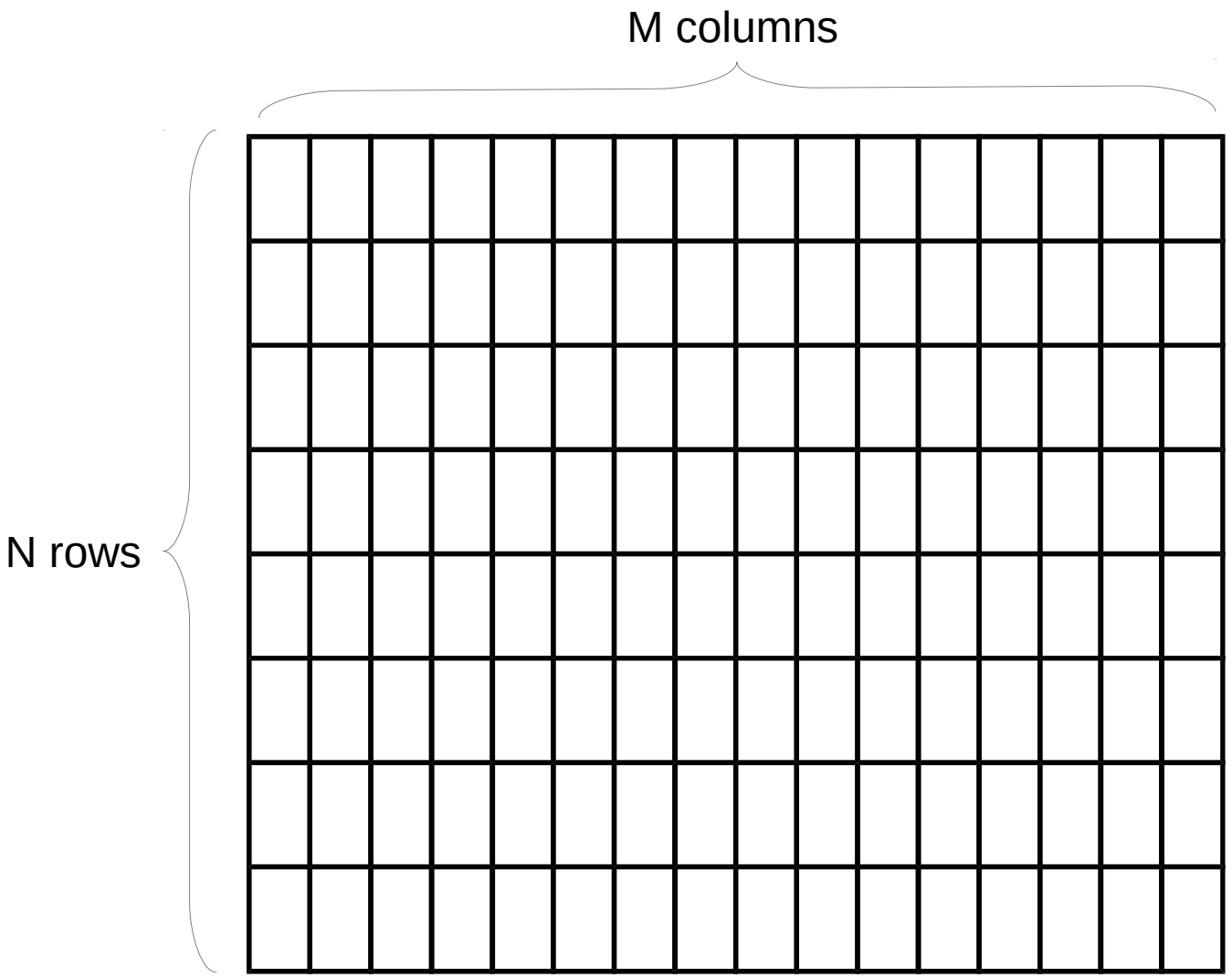
Cache Friendly Code

Assume a two dimensional array of data:

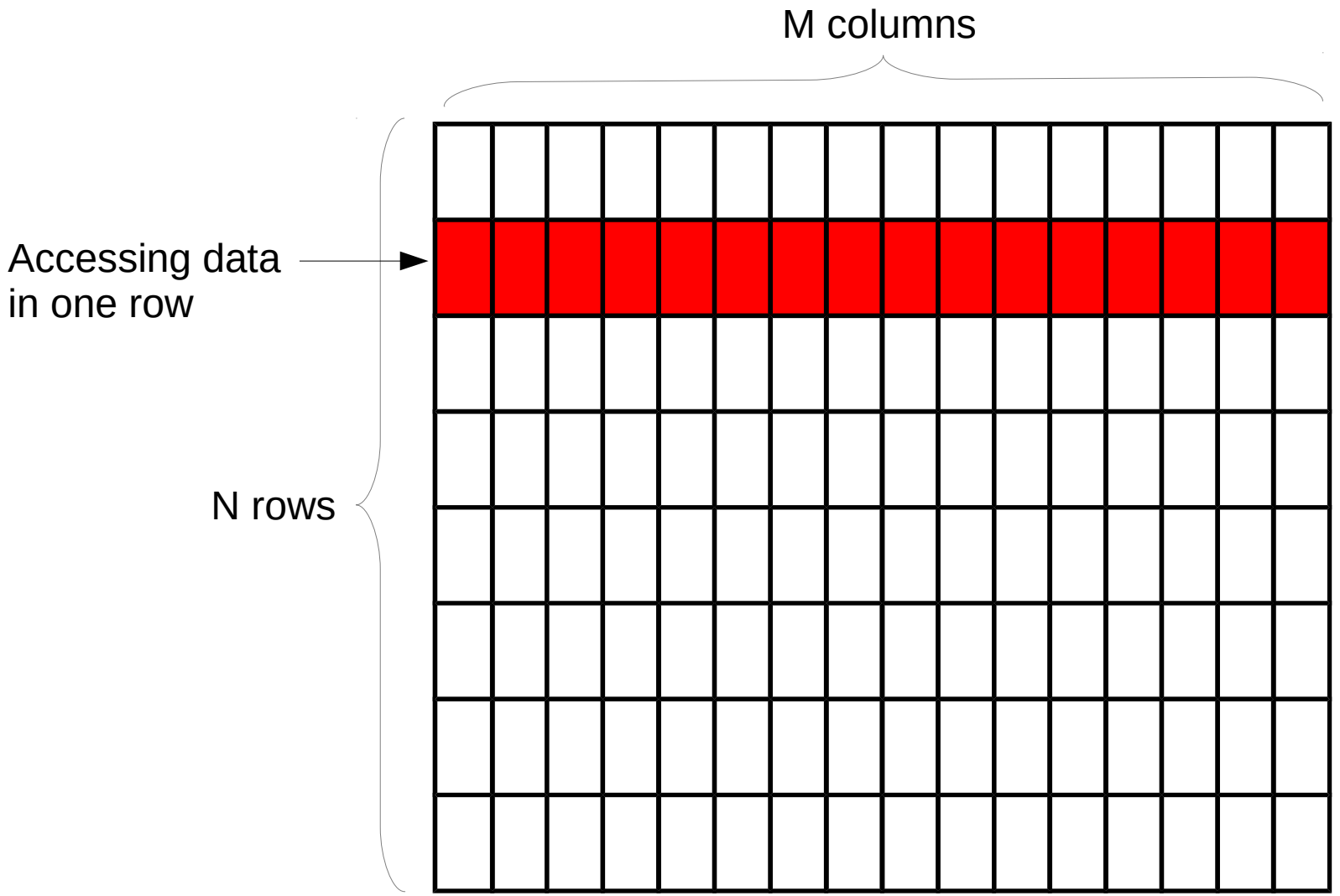


If we wish to obtain the sum of all entries in the array,
we would write a simple, double loop:

```
for( i= 0; i < N; i++ ) // iterate over rows
    for( j= 0; j < M; j++ ) // iterate over columns
        sum+= array[i][j]
```

Note that we access the variable `sum` in each iteration
of the inner loop, thereby guaranteeing temporal
locality.

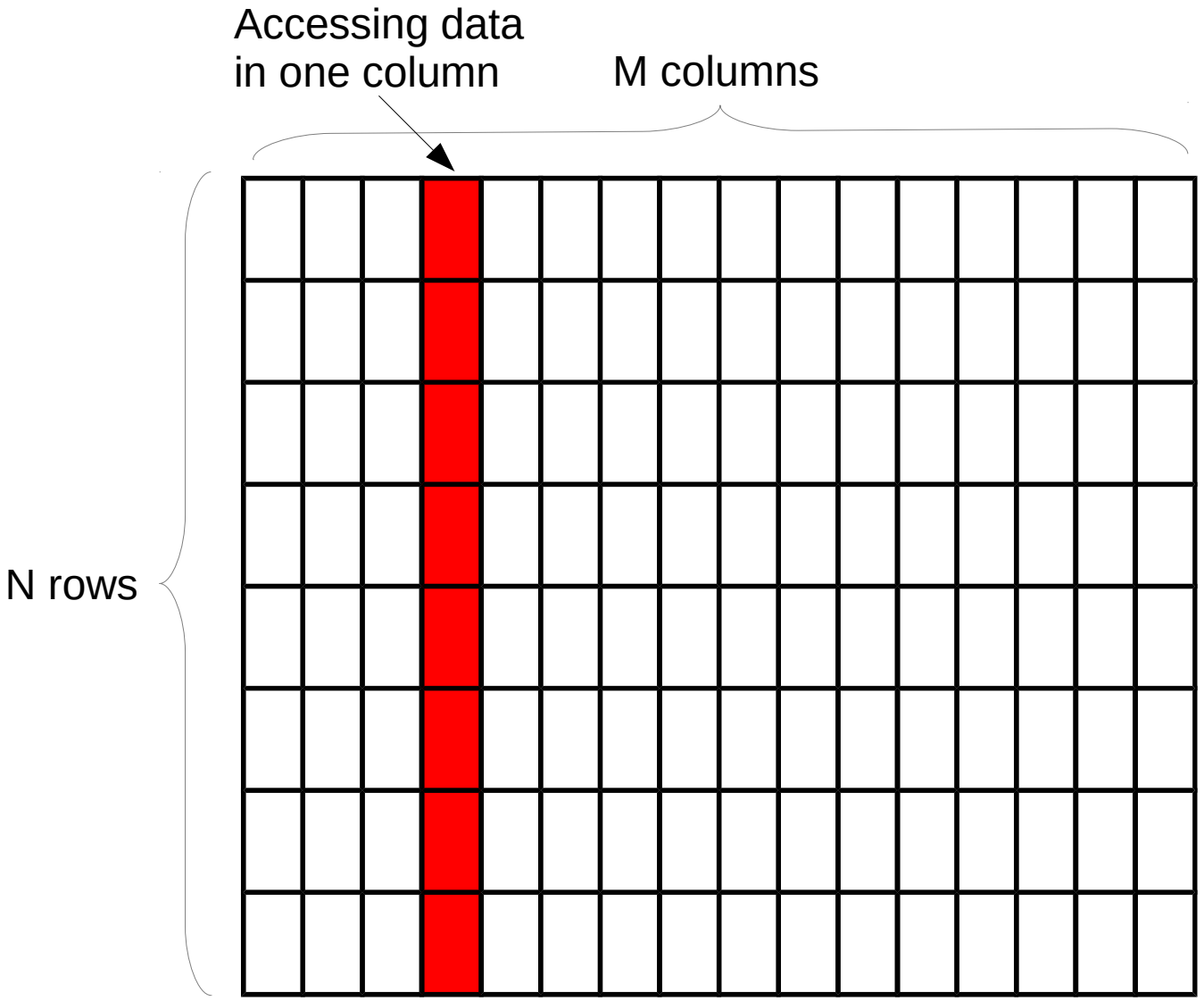
Moreover, we access data that is nearby in memory as we iterate across the columns in a single row. Thus, we also guarantee some level of spatial locality.



What would happen if we accessed the data by iterating over the columns first, then the rows?
Our double loop would appear as follows:

```
for( i= 0; i < M; i++ ) // iterate over column
    for( j= 0; j < N; j++ ) // iterate over rows
        sum+= array[j][i]
```

We have obliterated spatial locality because data in adjacent rows are not always nearby in physical memory.



Likewise, we can obliterate temporal locality by replacing our local variable `sum` with a location in memory. For example, suppose that we wished to store the sum of each cell in a row in the final column:

```
for( i= 0; i < N; i++ ) // iterate over rows
    for( j= 0; j < M - 1; j++ ) // iterate over columns
        array[i][M-1]+= array[i][j] // sum each row
```

Although we refer to the same memory location, `array[i][M-1]`, in each iteration of the inner loop, that location may be overwritten in the cache in order to make room for other data on that row, thereby destroying temporal locality.

In sum, consider temporal and spatial locality when writing code in order to optimize performance.