CDT204 -Computer Architecture

Lab Assignment 3

Prepared by Husni Khanfar, Afshin Ameri

Objectives

- Exploiting the cache memory to develop efficient code.
- Exploiting the vector instruction set to improve performance.

Introduction

In this assignment, we will perform experimental evaluations of different methods used to multiply two large (10^6 cells) matrices. Our role is to find the best method and scientifically discuss the results.

As part of the assignment you have received a C program file: `matmul.c`. We will write the C code to perform a matrix multiplication between two matrices in this file. The file contains extra code that measures the multiplication time and it can also verify the correctness of the output of each multiplication by using the function: `compare_matrices`.

The two multiplied matrices are N × N square matrices. The value of N is selected to be enough to create a large matrix that could not be fit in the cache memory of any modern computer. Setting N to a large value guarantees that the whole matrix is not loaded in the cache memory. Thus, many loadings and storing are done from/to the main memory to/from the cache memory.

If N = 1000, then the size of every matrix is 1000 * 1000 = 10^6 integers or about 4 Mega Bytes (considering that every integer = 4 bytes in Windows Platform).

Matrix Multiplication

The result of multiplying two matrices is stored in a new matrix. In this code, we have two matrices: A and B, the result is stored in matrix R. To perform the multiplication, the following condition must be satisfied:

The number of the rows of A must be equal to the number of the columns of B.

As a result, the number of rows in R matrix is equal to the number of rows in matrix A; similarly, the number of columns in matrix R is equal to the number of columns in matrix B. Since we use N × N dimension, then the dimensions of R matrix is N × N.

If every cell in any matrix is represented by ***i,j***, then we can calculate each $R_{i,j}$ by the following equation:

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

For example: A and B are 2 × 2 matrices. They can be multiplied by as follows:

$$\begin{array}{|c|c|} \hline 8 & 1 \\ \hline 7 & 4 \\ \hline \multicolumn{2}{c}{A} \end{array} \times \begin{array}{|c|c|} \hline 3 & 5 \\ \hline 1 & 0 \\ \hline \multicolumn{2}{c}{B} \end{array} = \begin{array}{|c|c|} \hline 3 \times 8 + 1 \times 1 & 8 \times 5 + 1 \times 0 \\ \hline 7 \times 3 + 4 \times 1 & 7 \times 5 + 4 \times 0 \\ \hline \multicolumn{2}{c}{R} \end{array}$$

Allocation of the Arrays

When multidimensional arrays are compiled, they are converted to a linear memory layout. Therefore a two-dimensional array is seen by the CPU as a one dimensional array. Since most C Compilers place the two-dimensional arrays in memory as row-major order, the two dimensional array is stored in the memory in a row by row manner. Accordingly, the first row is stored first, then the second row, etc.

As an example, suppose that we have the following 3x4 array:

99	54	2	-49
33	689	124	65
132	109	40	72

According to **row-major order**, this 3x4 array is stored as a one dimensional array as the following:

99	54	2	-49	33	689	124	65	132	109	40	72

If the same array is stored as a **column-major order**, then it appears like the following:

99	33	132	54	689	109	2	124	40	-49	65	72

Number of Loads

Suppose that the size of the cache memory is 512 bytes and the CPU makes continuous readings from a 512 byte × 512 byte array. Also suppose that the array is stored in the memory in row-major order. How many cache misses will happen if:

We write a code that reads all the elements of the array in a row by row manner? (We traverse all the columns of the first row, then the second row and so forth…)

512
……………………………………

We write a code that reads all the elements of the array in a column by column manner? (We traverse all the elements of the first column, and then the second column and so forth…)

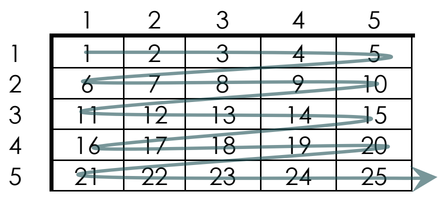
262 144
……………………………………

The Assignments

In order to perform a matrix multiplication: R = A × B, there are several ways for accessing the cells of these three matrices. We will develop different matrix multiplication algorithms using different access patterns and compare their performance.

First Algorithm (Version 1)

In this approach, you need to visit the values of the cells in matrix R according to this order:

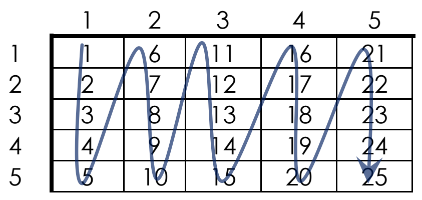


	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20
5	21	22	23	24	25

To calculate the value of $R_{(i,j)}$, you have to multiply each cell in Row ***i*** in A with its corresponding cell in the column ***j*** of B. The outcome of every multiplication is accumulated in the cell $R_{(i,j)}$. We do not move to a new cell before finishing all the calculations related to the current cell. Then we move to $R_{(i,j+1)}$.

Second Algorithm (Version 2)

In this approach, you need to visit the values of the cells in matrix R according to this order:



	1	2	3	4	5
1	1	6	11	16	21
2	2	7	12	17	22
3	3	8	13	18	23
4	4	9	14	19	24
5	5	10	15	20	25

To calculate $R_{(i,j)}$, you have to multiply each cell in Row ***i*** in A with its corresponding cell in the column ***j*** in B. The results are accumulated in the cell $R_{(i,j)}$.

Third Algorithm (Version 3)

In Version1 and Version2, the value of each cell in the result matrix (R) is calculated before moving to a new cell in R. The value of each cell in R is calculated by accumulating the outcome of many multiplications according to equation in page 1.

Version 3 has a different method than the previous two. This algorithm visits each cell in matrix A and all the calculations, which this cell has to be involved in, are performed before moving to a new cell. Notice that each cell in row ***i*** of matrix A is involved in calculating all the cells in row ***i*** in R.

For instance, the cell A(4,5) is used to calculate R(4,3) by this equation: R(4,3) += A(4,5) * B(5,3)

And it is involved to in calculating the value of R(4,7) by this equation: R(4,3) += A(4,5) * B(5,7)

Fourth Algorithm (Version 4)

Use your own ingenuity to invent a new sequential algorithm that overcomes all the previous ones.

Fifth Algorithm (Version 5)

Rewrite one of solutions above using SSE or AVX vector instructions. Which one did you choose? Why? Which one suits better for SSE/AVX?

The Results

Now that we have all of the algorithms developed, we can run some tests. In order to see the effects of compiler optimizations, we will run each test in two different compiler settings: No optimization and optimize for maximum speed. This is how you do it using visual studio:

- Goto Project >> "*Project_name*" Properties
- On the left panel traverse to node: Configuration properties >> C/C++ >> Optimization
- Now you can set the compiler setting "Optimization" to *disabled* or *maximize speed*.

For each test you will need to run the application at least 5 times and average the results. Use your results to fill the following table:

	Optimization disabled	**Maximize speed**
Algorithm 1 (row order)	6.08	0.926
Algorithm 2 (col order)	4.53	1.967
Algorithm 3	3.451	0.625
Algorithm 4 (I am a genius version)	3.085	0.39
Algorithm 5 (SSE/AVX)	1.522	0.411

Explain and discuss the results here:

The SSE-version is faster because we write parallell code. Algorithm 4 is faster then the 3 because we first transpose the second matrix and then just compute row by row.

This is Afshin.

Afshin measures the performance of his code in "Debug" mode.

Afshin won't pass the course.

Don't be like Afshin.