Matrix Multiplication and Computer Architecture

Computer Organization, Homework Assignment
Professor Ing-Jer Huang
Department of Computer Science and Engineering, National Sun Yat-Sen University

Due: Friday 17th June, 2022

1 Problem Description

You are asked to implement a matrix multiplication program in either C or C++. Given two matrices : $A_{n\times k}$ and $B_{k\times m}$, please calculate the output of $(A \times B)_{n\times m}$. You may refer to the C examples in Chapter 2 and Chapter 5 in the textbook.
Your report should be in PDF format and contain your answers to itemized assignments in Section 2 and Section 3. Submit your report to `https://cu.nsysu.edu.tw`

Input

Your program should recieve 3 positive integers n, k, m in the first line.

Next, n lines are provided for the matrix A. Each line has k **single-precision floating points**. The numbers are given in the following order:

$$A_{n\times k} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1k} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nk} \end{bmatrix}$$

At last, k lines are provided for the matrix B. Each line has m **single-precision floating points**. The numbers are given in the following order:

$$B_{k\times m} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1m} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & b_{k3} & \dots & b_{km} \end{bmatrix}$$

Constraints

- n, m, k are multiple of 32
- $n, m, k \leq 960$

- $|a_{ij}| \leq 1000,\ |b_{ij}| \leq 1000$

Output

You should output n lines for the matrix $A \times B$ with each line having m numbers:

$$C_{n\times m} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1m} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \cdots & c_{nm} \end{bmatrix}$$

The output will be accepted if the relative error is within 10^{-3}.

Small Sample

Just as an illustration, n, m, k are not multiple of 32 in this small example.

Input

```
2 3 4
-7.2715 7.8715 1.7661
4.7131 -6.9033 -3.6079
-3.4891 3.7054 -7.6667 7.6778
-9.0408 2.8584 8.3353 -6.9262
-6.4566 2.7242 -2.0897 9.1556
```

Output

```
-57.1967 0.36729 117.669 -94.179
69.2616 -12.0971 -86.1356 50.9674
```

2 Report

You should implement two versions of matrix multiplication:

1. Standard matrix multiplication, similar to the example in Chapter 2. Hint: this method should have 3 nested loops.
2. Blocked matrix multiplication, similar to the example in Chapter 4. Hint: this method should have 6 nested loops.

and answer the following questions:

1. Implementation:
   (a) How do you implement the matrix multiplication (both version)?
   (b) How do you verify the correctness of your program?

(c) Why the blocked version is correct? Answer by explaining the mathematics or program's calculation steps.

2. Experiment:
Run your program and conduct the following experiments on your computer. Before answering these questions, list the specification of your environment:

Hardware	CPU model and clock rate (e.g. i9-10900 @ 4.6 GHz)
	RAM speed (e.g. DDR4 3600MHz, LPDDR4 4333 MHz)
	Cache size (L1, L2 and L3)
Software	Compiler version
	Compile command
	Operation System

(a) Given that the size of your CPU's data cache is half of the L1 cache; 48 bits address in total; 8-way associative; and 64 bytes per cache line, draw the data cache address design based on Figure 5.18.. You should specify and explain how many bits the byte offset, index and tag takes. Note that Figure 5.18. has the cache line size of 32 bits (4 bytes), so that the number of byte offset bits is 2. If your L1 cache is not a power of 2, take the smallest number that is greater than your cache with a power of two. For instance, the cache you draw in your report will be 256 KiB if the actual data cache size is 160 KiB.

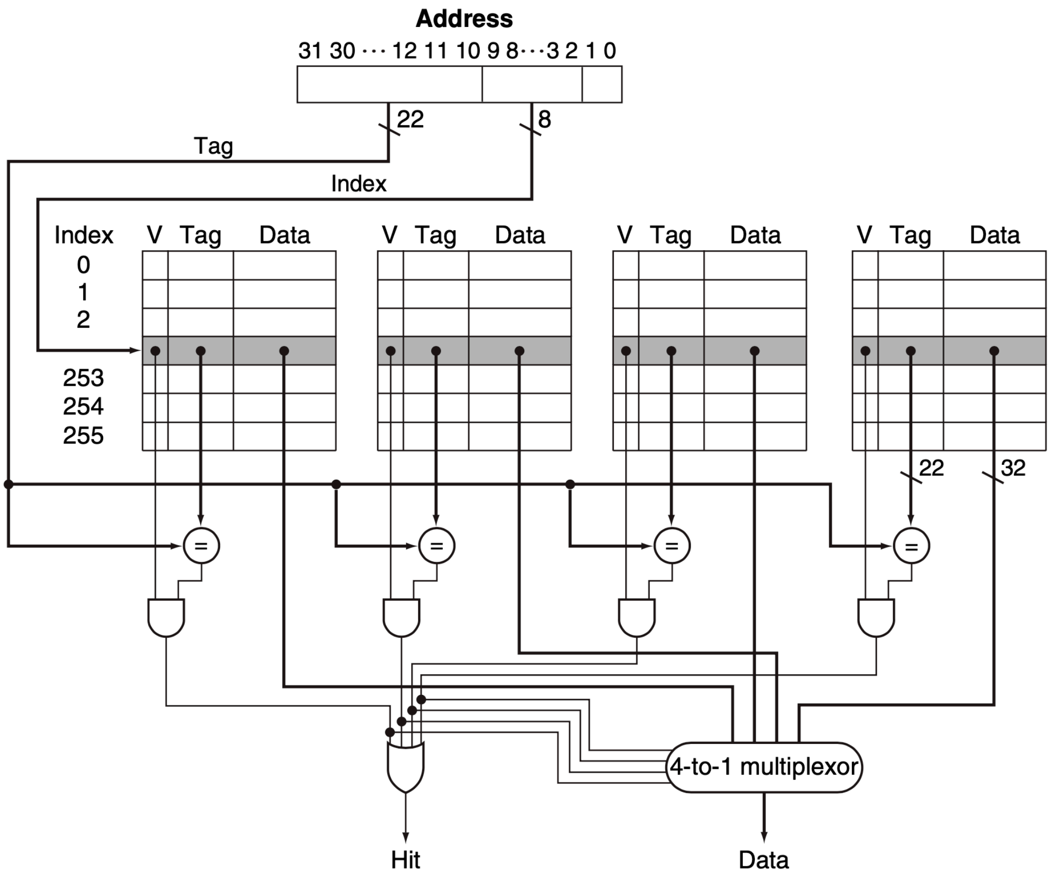

Figure 5.18.: The implementation of a 4-way set-associative cache requires four comparators and a 4-to-1 multiplexor

(b) Let's restrict n, m, k to the same value. Conduct at least 5 experiments based on different input size for your both versions of program. Explain how you produce the test data and what is the input size of those test data. The input sizes that you select should produce significantly different GFOPS values, as explained in the next paragraph.

(c) Measure the performance of your code by the 5 test cases you have created, and draw the result figure similar to Figure 5.23 (Note. This figure appears in the fifth edition of the textbook and in our class handout, but not in the sixth edition of the textbook). You should measure the performance as gigaFLOPS(GFLOPS). GFLOPS of this problem can be calculated by $\frac{n \cdot k \cdot m}{\text{execution time (ns.)}}$. Explain why can we measure GFLOPS in such calculation.

(d) Explain how you measure the execution time of the program. Is it appropriate to measure the total time of this program?

(e) Based on the result, explain the differences between the standard version and the blocked version. In addition, why different input size will have different performance in both versions?

(f) Conduct the experiments (b) again, but using the compile parameters `g++ -O2 matrix.cpp -o matrix2.exe` and `g++ -O0 matrix.cpp -o matrix0.exe`. (Assume your program name is `matrix.cpp`). Plot the result similar to the result figure in (c) and explain the differences between the flag `-O2` and `-O0`.

The program time can be measured by `std::chrono::duration` and `std::chrono::duration_cast` in high resolution in C++.

3. Advanced questions

(a) Cache line optimization:
Conventionally, array data are stored in the main memory in a row-major fashion. Data in a row are stored consecutively, and therefore they have the property of spacial locality in the cache block. As a result, accessing the array data in the row-major fasion has higher cache hit rate than accessing the array data in the column-major fasion.

Modify the code so that the B matrix is read in row major (transpose it to B^T), and compare the performance to the original code. Explain the differences.

(b) Vector instruction (bonus 15%):
In section 3.7 of the textbook, it introduced the vector instruction SSE and AVX on x86_64 platform, which provide abilities of single-instruction-multiple-data (SIMD) calculation. For example, SSE has 128-bit registers, and can perform 4 32-bit floating points calculation in an instruction. If we want to calculate $c_i = a_i \cdot b_i$ for $i = 0, 1, 2, 3$, it requires 4 multiplication; however, using vector instruction could reduce it to an instruction.

Modify your code to support AVX or SSE instruction and measure the performance. The performance should increase significantly.
Useful links:

i. Introduction-to-SSE-Programming

ii. Intel Intrinsics Guide

3 Appendix

Include your code in either C or C++ language as appendix to your report:

1. Standard matrix multiplication: `matrix.cpp`
2. Blocked matrix multiplication: `blocked-matrix.cpp`
3. Blocked matrix multiplication with cache line optimimzation: `blocked-cache-matrix.cpp`
4. Blocked matrix multiplication with vector instruction support: `vector-matrix.cpp`