CISC-221 Computer Architecture
The Memory Hierarchy
Lab 8: Matrix Multiplication

Submitted by: Raghav Taneja (20058783)

Instructor: Dave Dove (dove@cs.queensu.ca).

1 Introduction

Applications involving the creation and manipulation of matrices are abundant in most scientific fields. Examples can be found in the physical sciences, computer graphics, statistics, finance, and the list goes on and on.
Matrix multiplication is an operation that takes a pair of matrices and generates a third matrix.
In this lab you will collect data relating to the performance of the Raspberry Pi memory hierarchy in the execution of a textbook matrix multiplication function and investigate how performance can be altered significantly by rearranging loops within the function to increase spatial locality. This lab is a raspberry pi version of section 6.6.2 (page 625) of the course text.

2 Logistics

You must do this lab individually and submit your solution to the associated drop-box in OnQ under your own name, before the scheduled due date and time.

3 Background

Refer to section 6.6.2 or go to en.wikipedia.org/wiki/Matrix_multiplication so review how to multiply two a pair of n x n matrices, creating a third matrix with the results. We will be multiplying two 500 x 500 (float) element matrices creating a result matrix_r[ ] [ ].

The performance of a computer program is often determined by the speed of the input data streams that drive computations. Matrix multiplication example has two input data streams of two dimensional matrix data (matrix_a and matrix_b). The program is limited by the slowest input stream.

Two-dimensional arrays in C are structured in *row major* order. The sequence of elements in memory starts with the first row, followed by the second row, followed by the third row, and so on. Within the space allotted to a row, we find all of the column data associated with that row. The space between rows is therefore equal to the #columns x element size. We will be using a square array with 500 rows and 500 columns. All elements are floats. The space between rows is therefore 500 x 4 = 2000 bytes.

The ijk loop order of the naive code below accesses matrix_a in an efficient manner as it accesses a sequence of data elements in the same order as they are stored (row major order). The access pattern has a stride of 1element (or 4 bytes).

The sequence of matrix_a accesses is:

a[i=0][k=0], a[i][k +1], a[i][k +2] . . . a[i++][k = 0], a[i][k +1], a[i][k+2] . . .

The sequence of matrix_b accesses is not row order. It is:

b[k=0][j=0], b [k +1] [j], b [k +2] [j] . . . b[k = 0] [j++], b [k +1] [j], b [k+2] [j] . . .

The stride of matrix_b is the distance between row elements or 2000 bytes. It only accesses one element from a row before jumping to the next row. This means that only one element of a 4 element (32 byte) cache line will be used before the cache line is replaced. (Do you understand why? If not, review the notes and text references to cache memory)

With a 2000 byte distance between accessed elements, only two or three elements per 4k virtual memory page are used before the corresponding entries in the TLBs will be replaced by new entries. The next accesses to the current page will require an additional memory access to access the page table in memory before additional elements from that page can be read.

The key to improving performance will be to change the memory access pattern of matrix_b.

You will use two versions of a matrix multiplication function in this lab, implemented as two separate programs: ***naïve*** and ***interchange***.
The *naïve* program implements the standard textbook algorithm for matrix multiply:

```
/*naïve: ijk sequence*/
void multiply_matrices()
{
  int i, j, k ;

  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      float sum = 0.0 ;
      for (k = 0 ; k < MSIZE ; k++) {
        sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      }
      matrix_r[i][j] = sum ;
    }
  }
}
```

The second program, *interchange* simply reverses the middle and inner loop into an ikj sequence instead of an ijk sequence, and also removes the sum local variable and writes the result array directly.

Both the *naive* and the *interchange* programs produce the same result. Although the execution order is changed, the final sum in each result is the same because addition is both commutative and associative.

Now the program steps through both arrays (matrix_a and matrix_b) in row-major order. We should see a substantial increase in performance.

```
/*Interchange*/
void multiply_matrices()
{
 int i, j, k ;
   for (i = 0 ; i < MSIZE ; i++) {
     for (k = 0 ; k < MSIZE ; k++) {
        for (j = 0 ; j < MSIZE ; j++) {
           matrix_r[i][j] = matrix_r[i][j] + (matrix_a[i][k] * matrix_b[k][j]) ;
        }
     }
  }
}
```

5 Helpful Details

References:
onQ: Archived Lecture Notes: Storage Management, Raspberry Pi Memory
Course text: Chapter 6 Section6.6.2.

6 What you need to do:

Part 1

1. Log into the raspberry pi assigned to your group and create a new directory for this lab.
2. Complete Table 1 below by running 3 versions of the naïve program. Then repeat the process and run 3 versions of the interchange program, following this sequence:
   i. Run each program with the –i option and enter the Elapsed time, Scaled cycles, and Executed Instructions numbers from the resulting naïve.txt or interchange.txt files.
   ii. Run each program with the –c option and enter the Data cache access and the Data cache miss numbers from the resulting naïve.txt or interchange.txt files.
   iii. Run each program with the –t option and enter the MicroTLB miss and Main TLB miss numbers from the resulting naïve.txt or interchange.txt files.
3. Using the numbers from 2, calculate the remaining values for both the NAÏVE and INTERCHANGE using the formulae above Table 1.
4. Complete the Summary Comparison naïve vs interchange programs calculations below the table.
5. Submit your completed document by uploading it to the Lab 8 Submission item on the course web site.

CPI = Scaled cycles * 64/Instructions (Scaled cycles were scaled by a factor of 64)
Data cache access rate = Data cache access/(Instructions / 1000) PTI (Per Thousand Instrs)
Data cache miss rate = Data cache miss/(Instructions/1000) PTI

Data cache miss ratio = Data cache miss rate/Data cache access rate * 100%
MicroTLB miss rate = MicroTLB miss rate/(Instructions/1000) PTI
Main TLB miss rate = Main TLB miss/(Instructions / 1000) PTI

Table 1. Naïve vs. interchange

naïve -<option i,c, or t>; cat naive.txt
interchange -<option i,c, or t>; cat interchange.txt

METRIC	NAÏVE	INTERCHANGE
Measured values		
Elapsed time (-i)	**15.500**	**6.450**
Scaled cycles (-i)	**169421875**	**70546875**
Instructions (-i)	**1127254000**	**1127254000**
Data cache access (-c)	**7129558**	**2938007**
Data cache miss (-c)	**1483222**	**507202**
MicroTLB miss (-t)	**518364**	**127422**
Main TLB miss (-t)	**436840**	**22895**
Calculated values		
CPI	**9.6189501213**	**4.0053084753**
Data cache access rate	**6.3247129751 PTI**	**2.6063398311 PTI**
Data cache miss rate	**1.3157833106 PTI**	**0.449944733 PTI**
Data cache miss ratio	**20.8038422578 %**	**17.2634714621 %**
MicroTLB miss rate	**0.4598466716 PTI**	**0.113037523 PTI**
Main TLB miss rate	**0.3875257928 PTI**	**0.0203104181 PTI**

Reasons for Calculated Values

Data cache access counter. The reported number of data cache accesses comes from a hardware counter in the CPU but instead of counting every cache access, it only counts non-sequential cache accesses. This

means that sequential accesses to the same cache line only counts as a single access. This also distorts the data cache miss number. A more useful metric might be data cache accesses in relation to the total number of instructions executed. The same holds true for the data cache misses. The resulting data cache access ratio and data cache miss ratio numbers are given as *per thousand instructions* (PTI).

Extending the calculations to the TLB figures reveals the largest gain of all – the Main TLB miss rate reduction.

Summary Comparison of the *naive* vs. *interchange* programs.

Please complete the following. The formula for calculating the performance improvement is:

(metric before improvement – metric after improvement)/metric before improvement * 100%

The changes to the matrix multiplication code for the *interchange program* version resulted in:

i. a decrease in the execution time of the program of: **58.3871**%
ii. a decrease in the CPI of: **58.3602**%
iii. a decrease in the cache access rate of: **58.7912**%
iv. a decrease in the cache miss rate of: **65.8040**%
v. a decrease in the MicroTLB miss rate of: **75.4184**%
vi. a decrease in the Main TLB miss rate of: **94.7590**%