CSCI 2500: Computer Organization

Lab 11 – Exercises Memory/Caching

1) 'The Matrix': Recall matrices are 2-dimensional data structures wherein each data element is accessed via two indices. To multiply two matrices, 3 nested loops are used. The following code snippet assumes that matrices A, B, and C are all n-by-n (square matrices) and stored in one-dimensional column-major arrays:

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                C[i+j*n] += A[i+k*n] * B[k+j*n];
```

Matrix multiplication operations are commonplace in linear algebra algorithms, and efficient matrix multiplication is critical for many applications within the applied sciences. In the above code, note that the loops are ordered i, j, k. If we examine the innermost loop (k), we see that it moves through B with stride 1, A with stride n and C with stride 0 (stride is the jump from element to the next).

To compute the matrix multiplication correctly, the loop order doesn't matter. However, the order in which we choose to access the elements of the matrices can have a large impact on performance. Caches perform better (more cache hits, fewer cache misses) when memory accesses exhibit spatial and temporal locality. Optimizing a program's memory access patterns is essential to obtaining good performance from the memory hierarchy.

Download lab11_ex1.c from LMS; Labs → Lab 11 (12/2/2015). This C program file contains multiple implementations of matrix multiply with 3 nested loops. Compile this code with the following command: `gcc -O3 -o ex1 lab11_ex1.c`

Note that it is important here that we use the '-O3' flag to turn on compiler optimizations. Now run the program twice:

First run – Default Matrix Multiply (this part varies the size of matrices used in a multiple and times the results)
`./ex1` (Note: no arguments)

Second run - Runs all 6 different loop orderings in the matrix multiplication (ijk, kji, etc.)
`./ex1 partb`

Based on the program outputs, answer the following questions:

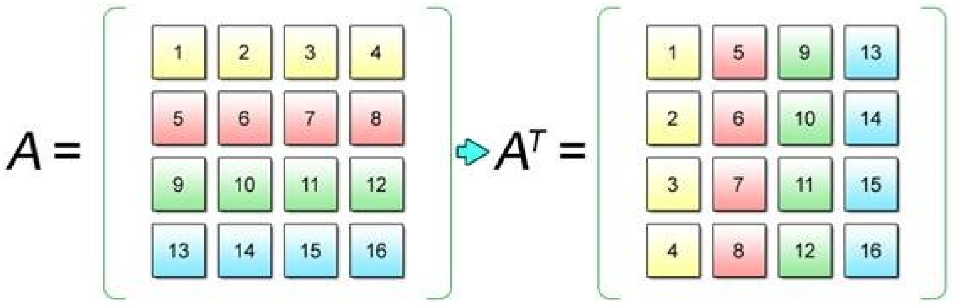
- First Run: Why does performance drop for large values of n (number of matrix elements)?

- Second Run: Which ordering(s) perform best for the 1000-by-1000 matrices?

  Which ordering(s) perform the worst?

  How does the program strides through the matrices with respect to the innermost loop affect performance?

2) 'The Matrix Reloaded': Another common routine in linear algebra libraries is the swapping of the rows and columns of a matrix. This operation is called a transpose and an efficient implementation can be very useful when performing more complicated linear algebra operations.



In exercise 1 above (matrix multiplication), we are striding across the entire A and B matrices to compute a single value of C. By operating in this fashion, the program is constantly accessing new values from memory and thus there is very little reuse of cached data! The amount of data reused in the caches can be increased by implementing a technique called cache blocking. Cache blocking is a technique that attempts to reduce the cache miss rate by improving the temporal and/or spatial locality of memory accesses.

With a blocking scheme, we can significantly reduce the magnitude of the working set in cache at any one point in time. This (if implemented correctly) will result in a substantial improvement in performance. For this exercise, you will implement a cache blocking scheme for matrix transposition and analyze its performance.

Download lab11_ex2.c from LMS. This C program file contains code to perform a matrix transposition. Compile this code with the following command:

```
gcc -O3 -o ex2 lab11_ex2.c
```

Again, it is important here that we use the '-O3' flag to turn on compiler optimizations. Now run the program:

```
./ex2
```

The program will print the time required to perform the naive transposition for a matrix of size 2000-by-2000.

Modify the function called transpose in lab11_ex2.c to implement a single level of cache blocking. More specifically, change the function to loop over all matrix blocks (not individual elements) and transpose each into the destination matrix. Make sure to handle the edge cases of the transposition(what if we tried to transpose the 15-by-15 matrix above with a blocksize of 4?)

Try block sizes of 2-by-2, 100-by-100, 200-by-200, 400-by-400 and 1000-by-1000.

Which performs best on the 2000-by-2000 matrix?

Which performs worst?

3) 'The Memory Mountain': (Optional Exercise) In this exercise, you will learn about the memory throughput of an X86-compatible microprocessor. When you have completed the lab, you will have a better appreciation for bandwidth between the main memory, caches, and the microprocessor.

Download lab11_ex3.zip from LMS. The zip file contains a C program (mountain.c) that contains code that will stride differently through multiple size arrays. The zip file also contains the the files clock.c, and fcyc.c, along with the two associated header files clock.h and fcyc.h. The clock.c and fcyc.c files contain code that will be used to for high resolution timing of the array operations in mountain.c.

For the code provided, you will need to use a Linux/Cygwin system running on top of a machine with a X86 architecture. The laboratory can be compiled by issuing the command:

```
gcc -m32 -O2 -o mountain *.c
```

Then, the program can be run by just typing:

```
./mountain
```

The output of this program will be a matrix of values that present the memory bandwidth of the computer for different sized working sets and different stride sizes. In looking at the results answer the following two questions (it helps to plot the results in Excel or Gnuplot):

What is the impact of different stride sizes?

How does the performance scale (i.e. improve/degrade) with increasing array size?

To complete this exercise you will need to modify the mountain.c file to take an average of all the different stride results (average each row in the original output). Once complete answer the following questions:

What are the major change points (increases and decreases) in the averaged throughput results?

Write down the model and speed of the microprocessor that you use to get the bandwidth results. Lookup the model information on the internet and figure out how much L1,L2, and L3 cache your system has. Do the major change points you identified in the previous question correspond to cache levels on your system?