CSCE 513: Computer Architecture, Fall 2018

Assignment #3, ~~due 10/29/2018, Monday 11:55PM~~
Due 10/31/2018, Wednesday 11:55PM

Covered Topics: 1) Locality and Performance, 2) Cache Optimization 2) Instruction Level Parallelism (ILP)

Total Points: 100+25 (bonus) points for undergraduates, and 110+15 (bonus) for graduates.

Submission:

1. **Only electronic submissions on dropbox are accepted.**
2. **You should submit two files, the mm.c source code file, and a SINGLE PDF file that includes everything else.**
3. **Number your solutions in the same way and in the same order as the questions are numbered in this document and do NOT include the questions as part of your submission.**
4. **Include your full name in the PDF file.**
5. **Scanned copy of handwritten answers will NOT be graded.**

1) Locality and Performance (Total 40 points, 10 bonus points will be given to undergraduate if function 7 and 8 are implemented).

In this assignment, you will implement different versions of matrix multiplication and evaluate their performance and cache miss rate. The purpose is to study the locality problem of programming, its impact to memory performance and application performance, and how to optimize it. The skeleton code is given in https://passlab.github.io/CSCE513/Assignment_3/mm.c.

Implementation:
Your implementation should be based on the materials in https://passlab.github.io/CSCE513/notes/lecture10_LocalityMM.pdf and implement the following functions as it listed in the mm.c file.

1. void mm_ijk(int N, REAL * A, REAL * B, REAL *C);
2. void mm_jik(int N, REAL * A, REAL * B, REAL *C);
3. void mm_kij(int N, REAL * A, REAL * B, REAL *C);
4. void mm_ikj(int N, REAL * A, REAL * B, REAL *C);
5. void mm_jki(int N, REAL * A, REAL * B, REAL *C);
6. void mm_kji(int N, REAL * A, REAL * B, REAL *C);
7. void mm_ijk_blocking(int N, REAL * A, REAL * B, REAL * C, int bsize);
8. void mm_cb(int N, REAL * A, REAL * B, REAL * C, int bsize); /* cache oblivious algorithm */

You need to login to one of the machine using ssh/putty. Check https://passlab.github.io/CSCE513/resources/devmachine.html for details. And then download the mm.c to your folder using wget, edit to add your code that has your implementation and codes for collecting data and printing out data (see below), compile and run the code. I recommend you implement one function first and make it working and then move on to implement others.

```
–bash–4.1$ wget https://passlab.github.io/CSCE513/Assignment_3/mm.c
–bash–4.1$ vim mm.c
–bash–4.1$ vi mm.c
–bash–4.1$ gcc –O0 mm.c –o mm –lpapi
```

Data collection:
To collect performance data and L1 and L2 cache miss info, insert code in the main function. For example, for mm_ijk method, insert the following code to collect the execution time and L1/L2/L3 misses.

```
/* for mm_ijk */
__builtin___clear_cache(A, A+N*N); /* flush cache so we have cold start */
__builtin___clear_cache(B, B+N*N);
__builtin___clear_cache(C, C+N*N);
long long PAPI_Values_ijk[NUM_PAPI_EVENTS];
elapsed_ijk = read_timer();
PAPI_reset(PAPI_EventSet);
mm_ijk(N, A, B, C);
PAPI_read(PAPI_EventSet, PAPI_Values_ijk);
elapsed_ijk = (read_timer() - elapsed_ijk);
```

The following code is inserted for printing out the information at the end:

```
printf("mm_ijk:\t\t\t%4f\t%4f\t\t%.2f\t\t%.2f\t\t%.2f\n", elapsed_ijk * 1.0e3, ((((2.0 * N) * N) * N) /
(1.0e6 * elapsed_ijk)), (double)PAPI_Values_ijk[1]/(double)PAPI_Values_ijk[0],
(double)PAPI_Values_ijk[2]/(double)PAPI_Values_ijk[1],
(double)PAPI_Values_ijk[3]/(double)PAPI_Values_ijk[2]);
```

You will need to insert similar codes for each of the methods you implement and evaluate. The following screen shot shows what it would look like after you finish your implementation, and compile and execute the code. The reason to use -O0 optimization flag is to turn off compiler optimizations so we only evaluate what we intend to. Ignore L3 cache miss rate as they do not look right.

```
[yanyh@fornax ~]$ vim mm.c
[yanyh@fornax ~]$ gcc -O0 mm.c -o mm -lpapi
[yanyh@fornax ~]$ ./mm
Usage: mm <N> <bsize>, default N: 512, bsize: 32
================================================================================================
        Matrix Multiplication: A[512][512] * B[512][512] = C[512][512], bsize: 32
------------------------------------------------------------------------------------------------
Performance:            Runtime(ms)     MFLOPS          L1_DMissRate    L2_DMissRate    L3_DMissRate
------------------------------------------------------------------------------------------------
mm_ijk:                 1154.999971     232.411656              0.06            1.00            1660.99
mm_jik:                 1147.000074     234.032640              0.07            1.00            882595128.03
mm_kij:                 648.999929      413.613999              0.00            0.06            11429.86
mm_ikj:                 644.000053      416.825208              0.00            0.05            550662.64
mm_jki:                 1611.999989     166.523237              0.10            1.00            0.00
mm_kji:                 1611.999989     166.523237              0.10            1.00            0.00
mm_ijk_blocking:        638.999939      420.086826              0.06            0.01            306323.79
mm_cb:                  663.000107      404.879959              0.06            0.01            192263419.84
[yanyh@fornax ~]$ ./mm 256 64
================================================================================================
        Matrix Multiplication: A[512][512] * B[512][512] = C[512][512], bsize: 64
------------------------------------------------------------------------------------------------
Performance:            Runtime(ms)     MFLOPS          L1_DMissRate    L2_DMissRate    L3_DMissRate
------------------------------------------------------------------------------------------------
mm_ijk:                 1062.999964     252.526308              0.06            1.00            1661.00
mm_jik:                 1055.000067     254.441174              0.07            1.00            882575280.55
mm_kij:                 648.000002      414.252246              0.00            0.06            11441.63
mm_ikj:                 643.999815      416.825362              0.00            0.05            532486.13
mm_jki:                 1497.000217     179.315576              0.10            1.00            0.00
mm_kji:                 1496.999979     179.315604              0.10            1.00            0.00
mm_ijk_blocking:        651.999950      411.710853              0.06            0.00            1057971.40
mm_cb:                  657.999992      407.956625              0.06            0.00            693895490.54
```

Data analysis and evaluation

When you have your implementation finished, run your program with different matrix size and block size, and collect the performance data and L1/L2 miss rate. Data collected should be input to the Excel file provided and the file will automatically generate plot figure for you to analyze.

Submission

Undergraduates who include implementation and evaluation of function 7 and 8 receive bonus 10 points.

Your submission should include:

1). mm.c file that contains your implementation.

2). A report that contains the two figures generated from the Excel file, and your analysis and discussion about the performance different between the 8 methods. Back up your performance discussion by referring to the cache miss rate data and how the change of loop order and blocking techniques can help improve cache hit ratio. Limit your discussion to less than half-page.

2) Cache Optimization (Total 25 points)

CAQA Textbook Appendix B and Chapter 2:

B.1 a and b (10 points)

2.1 a, b and c (15 points)

3) Instruction Level Parallelism (ILP) (Total 60 points, 15 of which are bonus)

CAQA Textbook chapter 3

3.14 (30 points, each 10 points for a, b, and c)

3.15 a (15 points), b (15 points bonus for both graduate and undergraduate)

If you need, questions for 2) and 3) are copied from the textbook in the following:

Exercises by Amr Zaky

B.1 [10/10/10/15] <B.1> You are trying to appreciate how important the principle of locality is in justifying the use of a cache memory, so you experiment with a computer having an L1 data cache and a main memory (you exclusively focus on data accesses). The latencies (in CPU cycles) of the different kinds of accesses are as follows: cache hit, 1 cycle; cache miss, 110 cycles; main memory access with cache disabled, 105 cycles.

a. [10] <B.1> When you run a program with an overall miss rate of 3%, what will the average memory access time (in CPU cycles) be?

b. [10] <B.1> Next, you run a program specifically designed to produce completely random data addresses with no locality. Toward that end, you use an array of size 1 GB (all of which fits in the main memory). Accesses to random elements of this array are continuously made (using a uniform random number generator to generate the elements indices). If your data cache size is 64 KB, what will the average memory access time be?

The transpose of a matrix interchanges its rows and columns; this concept is illustrated here:

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

Here is a simple C loop to show the transpose:

```
for (i = 0; i < 3; i++) {
 for (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

Assume that both the input and output matrices are stored in the row major order (*row major order* means that the row index changes fastest). Assume that you are executing a 256·256 double-precision transpose on a processor with a 16 KB fully associative (don't worry about cache conflicts) least recently used (LRU) replacement L1 data cache with 64-byte blocks. Assume that the L1 cache misses or prefetches require 16 cycles and always hit in the L2 cache, and that the L2 cache can process a request every 2 processor cycles. Assume that each iteration of the preceding inner loop requires 4 cycles if the data are present in the L1 cache. Assume that the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically, assume that writing back dirty cache blocks requires 0 cycles.

2.1 [10/15/15/12/20] <2.3> For the preceding simple implementation, this execution order would be nonideal for the input matrix; however, applying a loop interchange optimization would create a nonideal order for the output matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <2.3> What should be the minimum size of the cache to take advantage of blocked execution?

b. [15] <2.3> How do the relative number of misses in the blocked and unblocked versions compare in the preceding minimum-sized cache?

c. [15] <2.3> Write code to perform a transpose with a block size parameter B that uses $B \cdot B$ blocks.

3.14 [25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision aX plus Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, $Y = aX + Y$, for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y:

```
        addi   x4,x1,#800 ; x1 = upper bound for X
```



```
foo: fld      F2,0(x1)   ; (F2) = X(i)
     fmul.d   F4,F2,F0   ; (F4) = a*X(i)
     fld      F6,0(x2)   ; (F6) = Y(i)
     fadd.d   F6,F4,F6   ; (F6) = a*X(i) + Y(i)
     fsd      F6,0(x2)   ; Y(i) = a*X(i) + Y(i)
     addi     x1,x1,#8   ; increment X index
     addi     x2,x2,#8   ; increment Y index
     sltu     x3,x1,x4   ; test: continue loop?
     bnez     x3,foo     ; loop if needed
```

Assume the functional unit latencies as shown in the following table. Assume a one-cycle delayed branch that resolves in the ID stage. Assume that results are fully bypassed.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	4
Integer operations and all loads	Any	2

a. [25] <3.2> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

b. [25] <3.2> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

c. [25] <3.7> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 3.20. We will compare two degrees of loop unrolling. First, unroll the loop 6 times to extract ILP and schedule it without any stalls (i.e., completely empty issue cycles), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 10 times. Ignore the branch delay slot. Show the two schedules. What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

3.15 [20/20] <3.4, 3.5, 3.7, 3.8> In this exercise, we will look at how variations on Tomasulo's algorithm perform when running the loop from Exercise 3.14. The functional units (FUs) are described in the following table.

FU type	Cycles in EX	Number of FUs	Number of reservation stations
Integer	1	1	5
FP adder	10	1	3
FP multiplier	15	1	2

Assume the following:

- Functional units are not pipelined.
- There is no forwarding between functional units; results are communicated by the common data bus (CDB).
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus, the pipeline is IF/ID/IS/EX/WB.
- Loads require one clock cycle.
- The issue (IS) and write-back (WB) result stages each require one clock cycle.
- There are five load buffer slots and five store buffer slots.
- Assume that the Branch on Not Equal to Zero (BNEZ) instruction requires one clock cycle.

a. [20] <3.4–3.5> For this problem use the single-issue Tomasulo MIPS pipeline of Figure 3.10 with the pipeline latencies from the preceding table. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop. How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:
   - Iteration (loop iteration number)
   - Instruction
   - Issues (cycle when instruction issues)
   - Executes (cycle when instruction executes)
   - Memory access (cycle when memory is accessed)
   - Write CDB (cycle when result is written to the CDB)
   - Comment (description of any event on which the instruction is waiting)

   Show three iterations of the loop in your table. You may ignore the first instruction.

b. [20] <3.7, 3.8> Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).