

Matrix Multiplication and Cache Friendly Code

COMP 273 Winter 2021 - Assignment 4, Prof. Kry

Available: 24 March - Due date: 12 April

1 Introduction

In this assignment you will write code to multiply two square $n \times n$ matrices of single precision floating point numbers, and then optimize the code to exploit a memory cache. All the functions you write in this assignment must respect register conventions and work for different sizes of square matrices. Your code must also include useful comments to make it readable.

You will need to use two MARS tools in this assignment:

- **Data Cache Simulator:** This tool allows you to set different cache sizes and types, and measures the number of memory accesses, and cache misses.
- **Instruction Counter:** This tool counts the number of true MIPS assembly instructions that execute during your program.

Each tool needs to be connected to MARS, and you will want to use a combination of breakpoints and the reset button on each tool to make careful measurements of your code performance.

You will also likely want to try the Memory Reference Visualization tool (much like the Bitmap Display), as it lets you watch the memory reference patterns generated by your program. Likewise, the bitmap display tool will also be useful for visualizing the results. Remember to **set the base address to the heap (0x10040000)**, and choose the unit and display width to match the matrix size ($N = \text{display width} / \text{unit width}$). Running some tools, may noticeably slow down the execution of your program. If ever you notice MARS running much too slow, try restarting.

2 Assignment objectives (15 marks total)

Provided code will help you get started with this assignment. The code lets you run 3 different tests by changing `TestNumber` in the `.data` section at the top of the code.

- Test 0 will help you test the first objectives (matrix subtraction and Frobenius norm).
- Test 1 will help you checking your matrix multiply-and-add procedure. It allocates memory on the heap for 4 matrices (one being the solution) and loads test matrix data from file names specified in the data segment.
- Test 2 will help you compare different matrix multiply-and-add procedures.

Remember: MARS loads data files from the directory in which you start it, and test 1 will fail if the data files are not found.

1. **subtract** (2 marks)

Implement a function that subtracts two square $n \times n$ matrices A and B , and stores the result in matrix C . That is,

$$C_{ij} \leftarrow A_{ij} - B_{ij}.$$

Use the signature `void subtract(float* A, float* B, float* C, int n)` for your function, and note that you do not need nested for loops. Instead compute n^2 with `mul` and iterate over the three matrices by stepping each pointer by 4 bytes on each loop.

2. **frobeneousNorm** (2 marks)

Implement a function that computes the Frobenious norm of a matrix,

$$\|A\|_F = \sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} A_{ij}^2}.$$

That is, compute the sum of the squares of all the entries, and then take the square root using the floating coprocessor `sqrtd` instruction. Use the function signature

```
float frobeneousNorm( float* A, int n )
```

and remember that `$f0` is used as the return register for a float. Just as in the previous question, note that you can use a single for loop to visit all n^2 matrix entries.

3. **check** (2 marks)

Implement a function that prints the Frobenious norm of the difference of two matrices. That is, your function takes two square $n \times n$ matrices A and B , computes the difference $A - B$, and stores the answer in A by calling `subtract(A, B, A, n)`, and then finally computes the Frobenious norm by calling `frobeneousNorm(A, n)`. Print the resulting single precision floating point number with `SYSCALL` number 2. Use the function signature

```
void check( float* A, float* B, int n )
```

and test your `check` function by comparing different matrices. That is, using test 0 you should compute approximately 32.38494 when you compare 64-by-64 matrices loaded from `A64.bin` and `B64.bin`. Try changing the test 0 code to also comparing a matrix with itself to see if 0.0 is printed to the Run I/O console.

Leave the test 0 code such that it compares `A64.bin` and `B64.bin` when you submit your final assignment.

4. **MADD1 - Multiply and add version 1** (4 marks)

Write MIPS code to multiply two square $n \times n$ matrices A and B , and add the result to matrix C . That is,

$$C_{ij} \leftarrow C_{ij} + \sum_{k=0}^{n-1} A_{ik} B_{kj}.$$

All matrix entries are single precision floating point numbers. Use the following function signature and implement the naive matrix multiplication algorithm with three nested loops.

```

void MADD1( float* A, float* B, float* C, int n ) {
    for( i = 0; i < n; i++ ) {
        for( j = 0; j < n; j++ ) {
            for( k = 0; k < n; k++ ) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

```

Note that this is a cache unfriendly implementation because we load and store $C[i][j]$ on every iteration of our inner loop. It would be better to compute the sum of the inner loop in a register, and then add it to $C[i][j]$ after the inner loop is complete. Moreover, the memory access patterns in this naive implementation poorly exploits cached memory.

Nevertheless, the objective of this first question is to write a simple function that works and is well commented.

Test your implementation with test 1, and make sure that `Proc` in the `.data` segment is defined to be `MADD1`.

You are provided example matrices of different sizes for testing. If your multiplication is correct then the norm should be 0.0, or at most have a very small value, e.g., $1e-7$ (recall that floating point computation is not associative). In the provided matrices, you have $AB + C = D$, where the test matrices are loaded from `A64.bin`, `B64.bin`, `C64.bin` and `D64.bin`. Try changing `Size` and the file names in the data segment to test with matrices of different sizes.

5. MADD2 - Multiply and add version 2 (4 marks)

Write a cache friendly optimized version of the multiply and add function. Breaking up the nested loops and changing the order will take advantage of matrix entries that are already in the cache.

```

void MADD2( float* A, float* B, float* C, int n ) {
    for( jj = 0; jj < n; jj += bsize ) {
        for( kk = 0; kk < n; kk += bsize ) {
            for( i = 0; i < n; i++ ) {
                for( j = jj; j < min( jj + bsize, n ); j++ ) {
                    sum = 0.0;
                    for( k=kk; k < min( kk + bsize, n ); k++ ) {
                        sum += a[i][k] * b[k][j];
                    }
                    c[i][j] += sum;
                }
            }
        }
    }
}

```

Choose `bsize` to be 4 to match the number of words per block in the cache configurations

that you are asked to use for testing below.

Again, use your `check` function to make sure your code is working, and test with different matrix sizes. Using test 2 will be easiest here, with random matrices of different sizes.

Be sure to read the bonus objective before proceeding to the next objective.

6. Measure cache performance (1 marks)

Prepare and submit a `.csv` comma separated value file with entries that summarize the compute time and cache misses of your three functions. *Collect data only for the final version of functions that you submit for grading.* That is, be sure to not change your code once you start collecting data as the TA will check and remove marks if your file is not accurate.

The filename must have the form `ID.csv`, that is, it should consist of your student number and have the file extension `csv`, for instance, `260123456.csv`. The file should only contain ASCII. Use the MARS text editor to load and edit the csv file. Take care to replace 260123456 with your student number on each line of the file.

You will test both the naive and fast versions of your multiply and add function with a variety of cache configurations. In all cases, **use 64-by-64 matrices during your measurements.**

Measure only the performance of the multiply and add function. Use test 2 to compare MADD1 and MADD2, and for each procedure call, set one breakpoint at the `jal` to the function, and another at the next line.

- (a) Ensure the cache simulator is configured correctly, then connect it to MIPS,
- (b) ensure the instruction counter is connected to MIPS,
- (c) run your code up to the breakpoint,
- (d) press the *reset* button on the cache simulator
- (e) press the *reset* button on instruction counter,
- (f) press the run button to continue execution,
- (g) once the simulation stops at the breakpoint just after the jump and link, make note of the instruction count, and the cache performance.
- (h) Repeat steps (c) through (g) for MADD2.

Note that for the cache performance, you must record the memory access count, the number of cache misses, and the hit rate. Take care to use the specified cache configurations in your tests!

Your `csv` file must exactly match the required format. To best ensure you respect the file format, rename and edit the provided csv file. You may include comments in the file by starting a line with `#`, but otherwise there are multiple lines in this file to complete with *comma separated values*, or fields, on each line. These fields consist of your student number, the test name, the matrix size, the instruction count, the number of memory accesses, the number of cache misses, and an execution time in microseconds (which you must compute, see below). Here follows an example.

#	StudentID, Case,	N,	InstCnt,	MemAccess,	Misses,	MicroSeconds,	HitRate%
260123456,	Naive8Way,	64,	3993924,	532480,	308224,	34815,	42
260123456,	Naive4Way,	64,	3993924,	532480,	273280,	31321,	49
260123456,	NaiveDirect,	64,	3993924,	532480,	274816,	32464,	48
260123456,	Fast8Way,	64,	5031254,	655360,	45312,	9562,	93
260123456,	Fast4Way,	64,	5031254,	655360,	347008,	39731,	47
260123456,	FastDirect,	64,	5031254,	655360,	108160,	15847,	83

To specify the cache configuration, we provide the six settings in the tool reading left to right and top to bottom. For instance, N-Way/16/LRU/4/8/256 is an N-way set associative cache with 16 blocks total, a LRU replacement policy, 4 words per block, 8 blocks in each set, for a total cache size of 256 bytes. Thus, use following cache configurations for each test:

8Way	N-Way/16/LRU/4/8/256
4Way	N-Way/64/LRU/4/4/1024
Direct	Direct/128/LRU/4/1/2048

Compute the time in microseconds assuming a processor that runs at 1 GHz and executes one instruction every cycle, and assuming the cache miss penalty to be 100 cycles.

For instance, on the top line above, we add 100 times the misses to the the instruction count and divide by 1000. Thus 3993924 plus a penalty of 308224×100 is 34816324 nanoseconds, and when divided by 1000 gives 34816 microseconds.

In the example above, the naive implementation uses fewer instructions and would be faster if we were only counting instructions, but there are also far fewer cache misses, which can make the fast implementation much faster than the naive implementation!

Can you do better than the fast implementation example shown above? Can you identify why there might be a performance problem for the 4-way cache configuration given the size of the matrix problem? Consider how much memory each row of the matrix consumes, and how many bits would be used in the tag, index, and offset for this matrix configuration. It may be interesting to notice that a smaller cache with larger set-associativity can be better than a larger cache!

If you are unable to complete one or both of the multiply and add functions, you will not be able to receive full marks on this objective. Leave the entries in your `csv` file as zero in this case (and you will likewise not receive full marks for this measurement objective). Also note that if you have data entry errors or do not exactly follow instructions for the file format you may not receive full marks for this objective.

7. Bonus/competition (5 marks)

Bonus marks will be awarded to **top 10% of submissions** with the best cache performance using the 100 cycle cache miss penalty described above. Your optimized cache friendly code must not only be fast but also compute the correct answer! We will test your code on different matrices of various sizes, that is, not only the same as those you have documented in your `.csv` file. In order to reduce the total instruction count, you might likewise consider following strategies. Make sure you save different working copies of your code in case you introduce serious bugs when optimizing!

- Partial loop unrolling. If you can reduce the number of times that you increment and check your loop pointer, you will ultimately execute fewer instructions.

- Be smart with your addresses. The address of $A(i, j)$ is $A + i \times n + j$, and one might naively multiply i by n , get the low part of the result assuming no overflow, add j , multiply by 4, and add this to the address of A for a total of 5 instructions. But if we just accessed $A(i, j-1)$, then we only need to add 4 to the previous address.
- Replace pseudo instructions that expand to multiple true instructions with a smaller number of true assembly instructions, and find other ways to reduce the number of instructions inside loops. If you reduce the number of true instructions in your inner loop by just one, it will reduce the total count by thousands during large matrix multiplies.
- Identify locations where one instruction can do the work of two. For instance, using `bne` or `beq` alone instead of pairing it with an `slt` instruction, and avoiding the use of `j` instructions to form loops.
- Use more registers. Loading data from the cache is fast, but if you already have values stored in registers, then there is no need to load it again.
- If it helps you with versioning, consider moving your MADD1 and MADD2 implementations to a different file, and select "Settings→assemble all files in directory" in MARS. If you want to try out small changes to your code without changing all the labels for loops then it is best to put them into a different file. Say you made a 3rd version of MADD. Then place a `.globl MADD3` directive at the top of the file containing that version, and then test your code accordingly. Of course, for the final submission, you should make sure that you have your final solution for MADD2 in the one `.asm` file that you submit.

Submission Instructions

Submit exactly two files. Do not use a zip file or any other kind of archive. Your submission consists of your `.asm` file containing assembly code, and your `.csv` file containing measurements. Include your name and student number in all files submitted. Add to comments at the top of the `asm` file anything you would want the TAs to know (i.e., treat the comments at the top of the `asm` file as a README). All work must be your own, and must be submitted by MyCourses. Double check that you have correctly submitted your assignment as you will not receive marks for work incorrectly submitted.