CS 211: High Performance Computing Project 1

Performance Optimization via Register and Cache Reuse

Due date: Oct. 18th 11:59 pm

Submission Requirements and Instructions

Your Submission to iLearn of this project should be a tar file contain 2 files: a C code file called **mygemm.c**, and a PDF report including all results asked to provide in the following problems with necessary analysis, tables and charts.

About the **mygemm.c** file: it is a part of the test framework for this project, which originally contain several blank Matrix multiplication functions which need to be implemented. In the coding part, you only need to complete the Matrix multiplication functions in the **mygemm.c** file, and all of your codes to be submitted should be written in this file, meanwhile you can test the correctness and efficiency of your codes with the whole framework. Refer to the posted manual file in the framework for more information of the usage of this framework. For the convenience of your code test, you can change the other code files in the framework, but when evaluating your codes, all the code files in the framework will be the original versions expect the **mygemm.c** that you provide.

About running your codes: The evaluation of your codes and the test framework are both based on the Tardis machine, so it is highly recommended that you always run your codes on Tardis. The manual file of the framework contains necessary information of how to run it. Moreover, there is another attached file **tardis-tutorial.pdf**, which provides additional information and instructions for running codes on Tardis. If you only use the test framework (which should be enough for this project), you will not need to deal with the scripts, just following the manual file of the test framework is OK.

About the corner cases: when the matrix size is not a multiply of the block size, you need to deal with some corner cases, which is not mandatory in this project. To avoid them, some matrix sizes in the test framework have been slightly modified. Just remind that you do not need to deal with the corner cases. However, you can emphasize it in your report if you succeed in dealing with them.

Problems

1. Register Reuse (50 points).

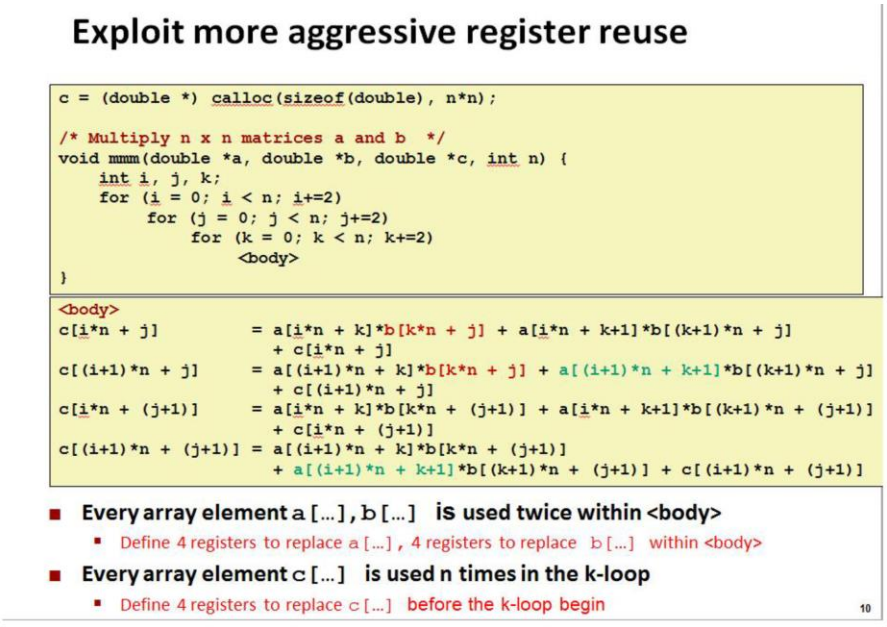
Part 1. (20 points) Assume your computer can complete 4 double floating-point operations per cycle when operands are in registers and it takes an additional delay of 100 cycles to read/write one operand from/to memory. The clock frequency of your computer is 2 Ghz. How long it will take for your computer to finish the following

algorithm ***dgemm0*** and ***dgemm1*** respectively for n= 1000? How much time is wasted on reading/writingoperands from/to memory? Implement the algorithm ***dgemm0*** and ***dgemm1*** and test them on TARDIS with n= 64, 128, 256, 512, 1024, 2048 in the framework provided. Check the correctness of your implementation with the framework, and report the time spend in the triple loop for each algorithm which is output by the framework. Calculate the performance (in Gflops) of each algorithm. Performance is often defined as the number of floating-point operations performed per second. A performance of 1 Gflops means 1 billion of floating-point operations per second.

```
/*dgemm0: simple ijk version triple loop algorithm*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i*n+j] += a[i*n+k] * b[k*n+j];

/*dgemm1: simple ijk version triple loop algorithm with register reuse*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
        register double r = c[i*n+j] ;
        for (k=0; k<n; k++)
            r += a[i*n+k] * b[k*n+j];
        c[i*n+j] = r;
    }
```

Part 2. (20 points) Let's use ***dgemm2*** to denote the algorithm in the following ppt slide from our class. Implement ***dgemm2*** and test it on TARDIS with n= 64, 128, 256, 512, 1024, 2048. Report the time spend in the algorithm. Calculate the performance (in Gflops) of the algorithm.



Part #3 (10 points). Assume you have 16 registers to use, please maximize the register reuse in your code (call this version code **dgemm3**) and compare your performance with ***dgemm0, dgemm1, and dgemm2***.

2. Cache Reuse (50 points).

Suppose your data cache has 60 lines and each line can hold 10 doubles. You are performing a matrix-matrix multiplication (**C=C+A*B**) with square matrices of size **10000X10000** and **10X10** respectively. Assume data caches are only used to cache matrix elements which are doubles. The cache replacement rule is ***least recently used first***. One-dimensional arrays are used to represent matrices with the row major order.

```
/* ijk – simple triple loop algorithm with simple single register reuse*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
        register double r=c[i*n+j];
        for (k=0; k<n; k++)
            r += a[i*n+k] * b[k*n+j];
        c[i*n+j]=r;
    }
```

```
/* ijk – blocked version algorithm*/
for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
        for (k = 0; k < n; k+=B)
        /* B x B mini matrix multiplications */
            for (i1 = i; i1 < i+B; i1++)
                for (j1 = j; j1 < j+B; j1++) {
                    register double r=c[i1*n+j1];
                    for (k1 = k; k1 < k+B; k1++)
                        r += a[i1*n + k1]*b[k1*n + j1];
                    c[i1*n+j1]=r;
                }
```

Part 1. (15 points) When matrix-matrix multiplication is performed using the *simple triple-loop* algorithm with single register reuse, there are 6 versions of the algorithm (ijk, ikj, jik, jki, kij, kji). Calculate the **number** of read cache misses for **each** element in **each** matrix for **each** version of the algorithm when the sizes of the matrices are **10000X10000** and **10X10** respectively. What is the percentage of read cache miss for each algorithm?

Part 2. (15 points) If matrices are partitioned into block matrices with each block being a 10 by 10 matrix, then the matrix-matrix multiplication can be performed using one of the 6 ***blocked version algorithms*** (ijk, ikj, jik, jki, kij, kji). Assume the multiplication of two blocks in the inner three loops uses the same loop order as the three outer loops in the blocked version algorithms. Calculate the **number** of read

cache misses for **each** element in **each** matrix for **each** version of the blocked algorithm when the size of the matrices is **10000**. What is the percentage of read cache miss for each algorithm?

Part 3. (10 points) Implement the algorithms in part (1) and (2). Report your execution time on TARDIS cluster. Adjust the block size from 10 to other numbers to see what is the optimal block size. Compare and analyze the performance of your codes for n=2048. Please always verify the correctness of your code.

Part 4. (10 points) Improve your implementation by using both cache blocking and register blocking at the same time. Optimize your block sizes. Compile your code using both the default compiler and gcc-4.7.2 with different optimization flags (-O0, -O1, -02, and -O3.) respectively. Compare and analyze the performance of your codes for n=2048. Highlight the best performance you achieved. Please always verify the correctness of your code.

3. Optional Bonus Question

1. Implement Strassen's algorithm and compare with your previous implementation.
2. Implement any other techniques you are curious about.