CS211 HW1 GEMM

Due: 10/20/2023 23:59

Guideline:

1. The template code is on github: https://classroom.github.com/a/GzxyHTUY. To complete this assignment, clone the template code to your space in **hpc-001** (refer to **CS 211 HPC Cluster.pdf**). Complete the matrix multiplication functions (including **dgemm0.c**~**dgemm3.c**, **dgemm6_xxx.c**, **dgemm6_xxx2.c**, **dgemm7.c**) and modify **starter.py** if necessary. Push them back to github.
   Complete the report (PDF) and submit on eLearn. Add your github account name in the report.
2. To test a single function, you can use:
   gcc main.c -o main
   ./main {function name} {n} {pad} (run on header node for debugging) (for example **./main dgemm0 1024 1**)
   srun main {function name} {n} {pad} (run on compute nodes)
   {function name} is the name of the matrix multiplication function you want to test.
   {n} is the length of the matrix. For example n=2048 means you want the matrix size to be 2048^2.
   {pad} is the length of padding. You can use proper padding to avoid corner cases if necessary. For example n=2048 and pad=30 means n=2070.
3. To run multiple tests at the same time, you can use **starter.py**. Some of the padding in starter.py are set as 1, and you may want to change them.
4. We use **main.c** to measure the running time and performance. You can modify your **main.c**, but this file will be in its original version when grading.
5. We use **hpc-001** to test your performance when grading. You can test and debug your code on other computers, but make sure to check them before submission on **hpc-001**.

Problem 1.1 (10 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 spent on reading/writing operands from/to memory?
Note1: If less than 4 floating-point operations are executed continuously, they also takes 1 cycle.
Note2: Assume that floating points are usually stored in the memory. When they are read they are loaded into registers. When they are written they are loaded into memory.

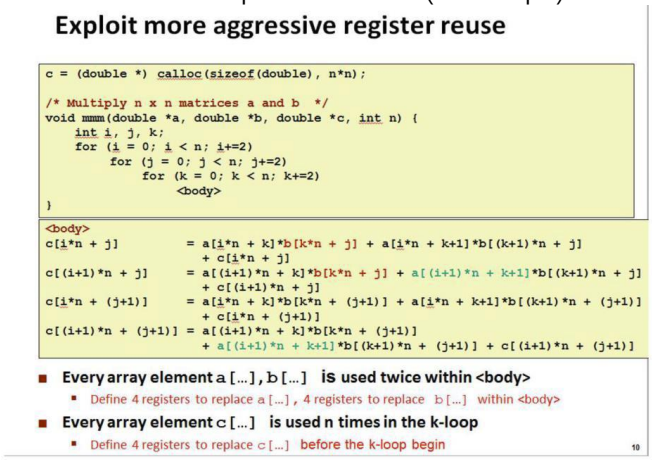
Note3: Assume integer calculations take no time.
Note4: Assume no parallelism.
Note5: Answer the time in seconds.

```
void dgemm0(double *C,double *A,double *B,int n)
{
    int i,j,k;
    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];
}
```

```
void dgemm1(double *C,double *A,double *B,int n)
{
    int i,j,k;
    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;
        }
}
```

Problem 1.2 (10 points):
Implement and test **dgemm0** and **dgemm1** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.
Check the correctness of your implementation, and report the time spent in the triple loop for each algorithm.
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 10^9 floating-point operations per second.

Problem 2 (20 points):
Implement **dgemm2** using 12 registers according to Page 10 of **optimizing-sequential-programs.pptx**.
Test **dgemm2** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.
Report the time and calculate the performance (in Gflops) of the algorithm.



Problem 3 (10 points):
Suppose you have 16 floating point registers. Implement **dgemm3** with the maximum register reuse.
Test **dgemm3** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.
Report the time and calculate the performance (in Gflops) of the algorithm.
Compare the performance of **dgemm3** with **dgemm0~2**.

Problem 4 (15 points):
Assume the cache has 60 lines. Each line can hold 10 doubles. When matrix-matrix multiplication (C=C+A*B) 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). For each version of the algorithm, each element in each matrix, calculate the number of read cache misses and number of reads.
What is the overall percentage of read cache miss for each algorithm?

```
void dgemm6_ijk(double *C,double *A,double *B,int n)
{
    int i,j,k;
    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;
        }

}
```

Note1: Matrix size can be 10000^2 or 10^2. Calculate for both sizes.
Note2: Cache only contains elements in matrixes.
Note3: The cache strategy is LRUF (least recent used first).
Note4: Each matrix is represented in a 1-dimension array in row major order.

Problem 5 (15 points):
We perform blocked matrix multiplication with single register reuse with block size 10^2. There can be 36 order of loops in all, but we just consider 6 conditions when the outer 3 loops has the same order with the inner 3 loops. For each version of the algorithm, each element in each matrix, calculate the number of read cache misses and number of reads.
What is the overall percentage of read cache miss for each algorithm?

```
void dgemm6_ijk2(double *C,double *A,double *B,int n)
{
    int i,ii,j,jj,k,kk;
    int b=1;//change b to the number you want
    for (i=0;i<n;i+=b)
        for (j=0;j<n;j+=b)
            for (k=0;k<n;k+=b)
                for (ii=i;ii<i+b;ii++)
                    for (jj=j;jj<j+b;jj++)
                    {
                        register double r=C[ii*n+jj];
                        for (kk=k;kk<k+b;kk++)
                            r+=A[ii*n+kk]*B[kk*n+jj];
                        C[ii*n+jj]=r;
                    }

}
```

Note1: Matrix size = 10000^2.

Problem 6 (10 points):
Implement all the 12 algorithms (**dgemm6_xxx** and **dgemm6_xxx2**) in problem 4 and 5 with matrix size 2048^2. Modify the block size in blocked matrix multiplication and optimize the block size (usually larger than 10^2). Compare and analysis the performance of block and non-blocked versions of algorithm.

Problem 7 (10 points):

Combine cache blocking and register blocking together and implement a matrix multiplication with size 2048^2 as fast as possible (**dgemm7**). Optimize the cache block size and register block size and list data to prove it. Compile your code with optimization flags **-O0, -O1, -O2, -O3**. Compare and analysis the result.