CS211 HW1 GEMM

Due: 10/15/2024 23:59

Before Start:

1. Click the following link to accept your assignment on GitHub Classroom:

   https://classroom.github.com/a/wYXOtfKL

   Clone the template code to your workspace on the **hpc-001** cluster (refer to the attached guide: **CS211 HPC Cluster.pdf** for instructions on how to log in). The core functions of the general matrix multiplication (GEMM) are implemented in **dgemm0.c, dgemm1.c, dgemm2.c, dgemm3.c, dgemm6_xxx.c, dgemm6_xxx2.c, dgemm7.c**. Complete the code in these files to ensure your project works correctly.

2. To test a single GEMM function, you can compile the corresponding file using **gcc main.c -o main**, and test the code with **./main {function_name} {n} {pad}** (e.g. **./main dgemm0 1024 1**). Please note:
   - **{function name}** is the name of the GEMM function you want to test.
   - **{n}** is the axis length of the square matrix being tested. For example, n=2048 means you want the matrix size to be 2048*2048.
   - **{pad}** is the length of padding. You can use proper padding to avoid corner cases when tiling. For example, n=2048 and pad=30 means n=2070.
   - This command will run the code on the head node, which is suitable for debugging. However, for performance profiling, you must use the **srun main {function_name} {n} {pad}** command to run your code on compute nodes. During grading, we will test your code on compute nodes and compare it with the experimental results in your report.

3. To run multiple tests simultaneously, you can use **starter.py**. Some of the padding in **starter.py** are set as 1 by default, and you may need to adjust.

4. We will use **main.c** to measure running time and performance. Please do not modify the **main.c** file when submitting your code.

5. We will use **hpc-001** to test your performance during grading. You can test and debug your code on other computers, but ensure that it runs on **hpc-001** before submission.

Submission:

Following the instructions in the questions below, complete the corresponding code or theoretical analysis and answer the questions. Collect experiment data and compare the different optimization strategies when analyzing your results. You need to submit a **PDF** report for this assignment, and your code must be submitted on GitHub Classroom (push your final code to your repository in GitHub Classroom).

Report requirements: Do not include the problem description in your report; the report should only contain answers to the questions, the corresponding experimental data, data analysis, and the theoretical reasoning process.

Report naming format: **cs211_hw1_firstname_lastname.pdf**. Please include the link to your GitHub repository, your name, and your SID on the first page of your report.

Q1 (20 points, 10 for each sub question)

(a) Assume that your computer can perform 4 double-precision floating-point operations per clock cycle when the operands are stored in registers. Additionally, accessing an operand from memory incurs a delay of 100 cycles for reading or writing. The clock frequency of your computer is 2 GHz.

- How long will it take for your computer to complete the algorithms **dgemm0** and **dgemm1** respectively, for n=1000? Please provide the answer in seconds.
- How much time is spent on reading/writing operands from/to memory for **dgemm0** and **dgemm1** respectively, when n=1000? Please provide the answer in seconds.

Hint:

1. If less than 4 floating-point operations are executed continuously, they also takes 1 cycle.
2. floating points are usually stored in the memory. When they are read, they are loaded into registers. When they are written, they are stored into memory.
3. We assume integer calculations take no time; no parallelism is considered in this case.

```
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;
        }
}
```

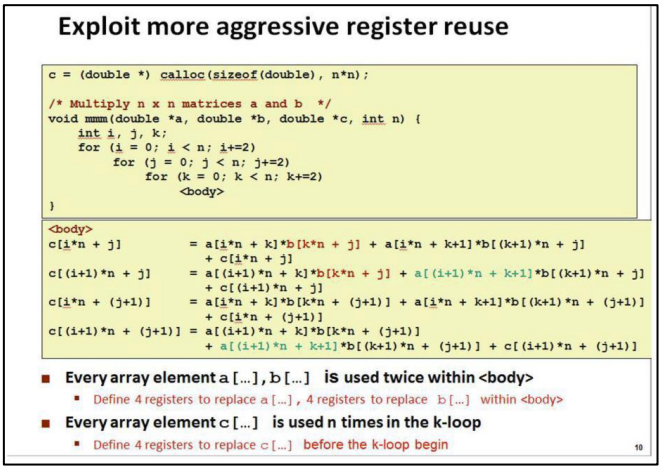
(b) Implement and test **dgemm0** and **dgemm1** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.

- Verify the correctness of your implementation and report the time spent in the triple loop for each algorithm.
- Calculate the performance of each algorithm in Gflops. Performance is typically measured as the number of floating-point operations executed per second. A performance of 1 Gflops corresponds to 10^9 floating-point operations per second.

Q2 (20 points)

Implement **dgemm2** using 12 registers as outlined on Page 10 of **optimizing-sequential-programs.pptx.** (shown in the figure below)

- Test **dgemm2** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.
- Report the execution time and calculate the performance of your code in Gflops.



Q3 (10 points)

Assuming you have 16 floating-point registers, implement **dgemm3** with the maximum possible register reuse.

- Test **dgemm3** on **hpc-001** with n=64, 128, 256, 512, 1024, 2048.
- Report the execution time and calculate the performance of your code in Gflops.
- Compare the performance of **dgemm3** with **dgemm0~2**.

Q4 (15 points)

Assuming a cache with 60 lines, each capable of holding 10 double values, is utilized in your program. You implement a simple GEMM (**C**=**C**+**A***__B__) with single cache reuse (as shown in the figure below), and you test different looping configurations: **ijk, ikj, jik, jki, kij, kji**. Please answer the following questions.

```
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;
        }
}
```

- When the matrix size is 10*10, calculate the number of cache reads required for matrix elements in **A**, **B**, and **C** respectively. Additionally, determine how many of these reads will incur a cache miss. Please calculate the read cache miss rate for matrices **A**, **B**, **C**, as well as the overall read cache miss rate for the GEMM operation. Please provide a complete calculation process in the report and present your results in a format similar to the table below.

N=10

Loop Order	A		B		C		Overall Miss Rate
	# of read	# of miss	# of read	# of miss	# of read	# of miss	
ijk							
ikj							
…							

- When the matrix size is 10000*10000, what will these values be? Please provide a complete calculation process in the report and present your results in a table format.

Hint:

1. The cache contains only the elements from the matrices.
2. The cache strategy employed is LRUF (Least Recently Used First).
3. Each matrix is represented as a one-dimensional array in row-major order.

Q5 (15 points)

On the same machine used in Q4, you implement blocked GEMM algorithms with single cache reuse and a block size of 10*10. You test 6 loop configurations where iteration order inside the blocks is the same as that at the matrix level (i.e. the outer 3 loops follow the same order as the inner 3 loops, as shown in the figure below). Please answer the following questions.

```
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;
                    }
}
```

- When the matrix size is 10000*10000, calculate the number of reads and read cache misses for all six loop orders. Please provide a complete calculation process in the report and present your results in a format similar to the table below.

N=10000, B=10

Loop Order	A		B		C		Overall Miss Rate
	# of read	# of miss	# of read	# of miss	# of read	# of miss	
ijk-ijk							
ikj-ikj							
…							

- For the **kji**-**kji** loop order, how many cache misses will occur for the element A[0][0] during the entire GEMM execution? Additionally, calculate the number of cache misses for the elements A[17][21], B[100][130], B[101][134], C[68][90], and C[2000][1297].

Hint:

1. The cache contains only the elements from the matrices.
2. The cache strategy employed is LRUF (Least Recently Used First).
3. Each matrix is represented as a one-dimensional array in row-major order.

Q6 (10 points)

Implement all 12 algorithms (**dgemm6_xxx** and **dgemm6_xxx2**) from problem 4 and 5 using a matrix size of 2048*2048. Compare and analyze the performance of block and non-blocked versions of the algorithm. Experiment with different block sizes (other than 10*10) in your implementation, and record how the performance changes as you optimize the block size.

Q7 (10 points)

Combine cache blocking and register blocking to implement a matrix multiplication of size 2048*2048 as efficient as possible (**dgemm7**). Optimize both the cache block size and register block size, and record the execution time of each block-register configuration. Analyze how performance varies with different block size and cache size. Compile your code with optimization flags **-O0**, **-O1**, **-O2**, **-O3**, and valuate how the execution time changes with each optimization level.