Name: Andrew Tee
SID: 862174480
Github Account Name: atee001
Github Repo for HW1: https://github.com/CS211-Fall2023/hw1-atee001/tree/main

HW#1

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 will it take for your computer to finish the following algorithm dgemm0 and dgemm1 respectively for n = 1000? (if less than 4 flop continuously -> 1 clock cycle)

dgemm0:

Inner Loop:
  LD C → 100 cycles
  LD A → 100 cycles
  LD B → 100 cycles
  MULT A,B, ADD C, A,B -> 1 cycle
  STR C → 100 cycles
= 401 clock cycles in inner loop
=> 401n^3 clock cycles in algorithm

If n = 1000: 401(1000)^3 = 401 Giga cycles
1 cycle => 2 GigaHz
401 Gigacycles => (0.5 * 10^(-9)) * 401 * 10^9 = **200.5 seconds**

dgemm1:
Outer Loop
  LD C → 100 cycles
  Inner Loop:
    LD A → 100 cycles
    LD B → 100 cycles
    MULT A,B, ADD C, A,B -> 1 cycle
    STR C → 100 cycles
  STR C → 100 cycles
= 201 clock cycles in inner loop
= (201n + 200)(n^2)
=> 201n^3 + 200n^2 clock cycles in algorithm

If n = 1000: 201(1000)^3 + 200(1000)^2 = 2.012 * 10^11 clock cycles

2.012 * 10^11 clock cycles = **100.6 seconds**

How much time is spent on reading/writing operands from/to memory.

dgemm0: 400 Giga cycles * 0.5 * 10^(-9) seconds per cycle = **200 seconds**

dgemm1: (200 Giga cycles + 200 Mega cycles)* 0.5 * 10^(-9) seconds per cycle = **100.1 seconds**

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.

Check github for implementation

dgemm0:

2 flops in inner loop

=> 2(n^3) flops in triple loop

n	time	flops	gflops	**gflops per sec**
6.40E+01	1.12E-03	5.24E+05	5.24E-04	**4.68E-01**
1.28E+02	9.44E-03	4.19E+06	4.19E-03	**4.44E-01**
2.56E+02	1.06E-01	3.36E+07	3.36E-02	**3.17E-01**
5.12E+02	1.21E+00	2.68E+08	2.68E-01	**2.22E-01**
1.02E+03	1.03E+01	2.15E+09	2.15E+00	**2.08E-01**
2.05E+03	2.14E+02	1.72E+10	1.72E+01	**8.03E-02**

dgemm1:

2 flops in inner loop

=> 2(n^3) flops in triple loop

n	time	flops	gflops	**gflops per sec**
6.40E+01	7.42E-04	5.24E+05	5.24E-04	**7.07E-01**
1.28E+02	6.11E-03	4.19E+06	4.19E-03	**6.87E-01**
2.56E+02	7.27E-02	3.36E+07	3.36E-02	**4.62E-01**

5.12E+02	7.44E-01	2.68E+08	2.68E-01	**3.61E-01**
1.02E+03	5.97E+00	2.15E+09	2.15E+00	**3.59E-01**
2.05E+03	1.56E+02	1.72E+10	1.72E+01	**1.10E-01**

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.

Check github for implementation

dgemm2				
n	time	flops	gflops	**gflops per sec**
6.40E+01	2.77E-04	5.24E+05	5.24E-04	**1.89E+00**
1.28E+02	2.42E-03	4.19E+06	4.19E-03	**1.73E+00**
2.56E+02	2.58E-02	3.36E+07	3.36E-02	**1.30E+00**
5.12E+02	2.38E-01	2.68E+08	2.68E-01	**1.13E+00**
1.02E+03	2.07E+00	2.15E+09	2.15E+00	**1.04E+00**
2.05E+03	4.17E+01	1.72E+10	1.72E+01	**4.12E-01**

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.

Check github for implementation (Implemented with 15 registers as professor said 16 is not possible)

Dgemm3

Inner loop:
54 flops

=> (54)*(n/3)^3 flops in total algorithm

dgemm3				
n	time	flops	gflops	**gflops per sec**
6.60E+01	2.14E-04	5.75E+05	5.75E-04	**2.69E+00**
1.29E+02	1.52E-03	4.29E+06	4.29E-03	**2.82E+00**
2.58E+02	1.53E-02	3.43E+07	3.43E-02	**2.24E+00**
5.13E+02	1.32E-01	2.70E+08	2.70E-01	**2.05E+00**
1.03E+03	8.78E-01	2.16E+09	2.16E+00	**2.46E+00**
2.05E+03	6.92E+00	1.72E+10	1.72E+01	**2.48E+00**

Dgemm3 has the best computing density (gflops per second) and fastest run time among versions dgemm0 to dgemm3. This is because the 15 registers store more intermediate values reducing the need to fetch data from memory.

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?

IJK && JIK

For 10^2 size = **30 misses**.
All elements fit into cache so 30 misses need to load 100 elements of A, B, and C into Cache.

Number of reads = ((2)n + 1)(n^2) = 2100 reads

Miss/Read = 30/2100 = 0.01428571428 = **1.43% miss to read ratio**

Inner loop misses

For A: 1 miss per 10 elements
For B: 1 miss per 1 element
For C: no misses

Outer loop misses
For A: no miss
For B: no miss
For C: 1 miss per 1 element (This is because cache is overfilled after loading block of A & B)

Total Misses = ([(1/10) + 1 + 0]n + 1)n^2

For 10000^2 size = **1.1 * 10^12 misses**

Number of reads = (2n + 1)n^2 = 2.0001 * 10^12 reads

Miss/Read = (1.1 * 10^12) / (2.001 * 10^12) = **55% miss to read ratio**

IKJ && KIJ

For 10^2 size = **30 misses**.
All elements fit into cache so 30 misses need to load 100 elements of A, B, and C into Cache.

Number of reads = ((2)n + 1)(n^2) = 2100 reads

Miss/Read = 30/2100 = 0.01428571428 = **1.43% miss to read ratio**

Inner loop misses

For A: no miss
For B: 1 miss per 10 element
For C: 1 miss per 10 elements

Outerloop misses:

For A: 1 miss per 1 element (Cache gets overfilled need to retrieve A from memory again).
For B: no misses
For C: no misses

Total Misses = ([(1/10) + (1/10) + 0]n + 1)n^2
For 10000^2 size = **2.001 * 10^11 misses**

Number of reads = (2n + 1)n^2 = 2.0001 * 10^12 reads
Miss/Read = (2.001 * 10^11 misses) / (2.0001 * 10^12 reads) = **10% miss to read ratio**

JKI && KJI

For 10^2 size = **30 misses.**

All elements fit into cache so 30 misses need to load 100 elements of A, B, and C into Cache.

Number of reads = ((2)n + 1)(n^2) = 2100 reads

Miss/Read = 30/2100 = 0.01428571428 = **1.43% miss to read ratio**

Inner loop misses

For A: 1 miss per 1 element
For B: no misses
For C: 1 miss per 1 element

Outerloop misses:

For A: no misses.
For B: 1 miss per 1 element
For C: no misses

Total Misses = ([(1) + (1) + 0]n + 1)n^2
For 10000^2 size = **2.0001×10^12 misses**

Number of reads = (2n + 1)n^2 = 2.0001 * 10^12 reads
Miss/Read = (2.0001 * 10^12 misses) / (2.0001 * 10^12 reads) = **100% miss to read ratio**

Problem 5

For block matrix multiplication all three mini matrices of A, B, and C fit into cache therefore all blocked matrix multiplications have the same read cache miss.

Number of cache misses:

(B^2)/10 cache misses for A
(B^2)/10 cache misses for B
(B^2)/10 cache misses for C

For one block dot product there will be N/B number of blocks for A & B

[(B^2)/10 + (B^2)/10]*(N/B) + (B^2)/10

This is just to compute one block of C

For all blocks of C there are (N/B) * (N/B) number of blocks:

=> ([(B^2) / 5] * (N/B) + (B^2)/10) * (N/B)^2

Given Block Size is 10^2 and Matrix size = 10000^2

Total number of cache misses = **2.001 * 10^(10) misses**

```c
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;
                    }

}
```

2 reads in kk loop (for A & B)
1 read in jj loop (for C)

B^2 iterations for ii && jj loop
(N/B)^3 iterations for i, j, & k loops

(((2 * B) + 1)B^2) * (N/B)^3

Total number of reads = **2.1 * 10^12 reads**

Miss/Read = **0.95% miss to read ratio**

This is for all 6 block matrix multiplication versions.

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 analyze the performance of block and non-blocked versions of the algorithm.

See github for all 12 algorithms.

To find the theoretical best block size for L1 Cache:
The CPU model is: Model name:		Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz
Cache size is: 32KB = 32 * 1024 = 32,768 Bytes

Number of cache space per matrix = FLOOR(32KB/ 3 matrices) = 10,922 Bytes
Each element is 8 bytes => 1,365 Elements per block

Block Length/Width = sqrt(1,365 Elements per block) = 36

Therefore theoretically the block size that works best is 36^2

Compared the block size from 30 to 40 and confirmed 36 is the best with a run time of 37.068995s for blocked ijk.

Block size	Time
40.00	40.06
37.00	37.41
36.00	**37.07**
35.00	38.28
34.00	38.31
33.00	38.34
32.00	38.61
31.00	38.67
30.00	38.73
10.00	43.22

BLOCKED 36^2	ijk2	ikj2	jik2	jki2	kij2	kji2
Time	22.792445s	29.884705s	23.861618s	30.664831s	28.002248s	30.243981s

NON BLOCKED	ijk	ikj	jik2	jki2	kij2	kji2
Time	293.785486s	33.884705s	202.357588s	305.080544s	38.433066s	355.156087s

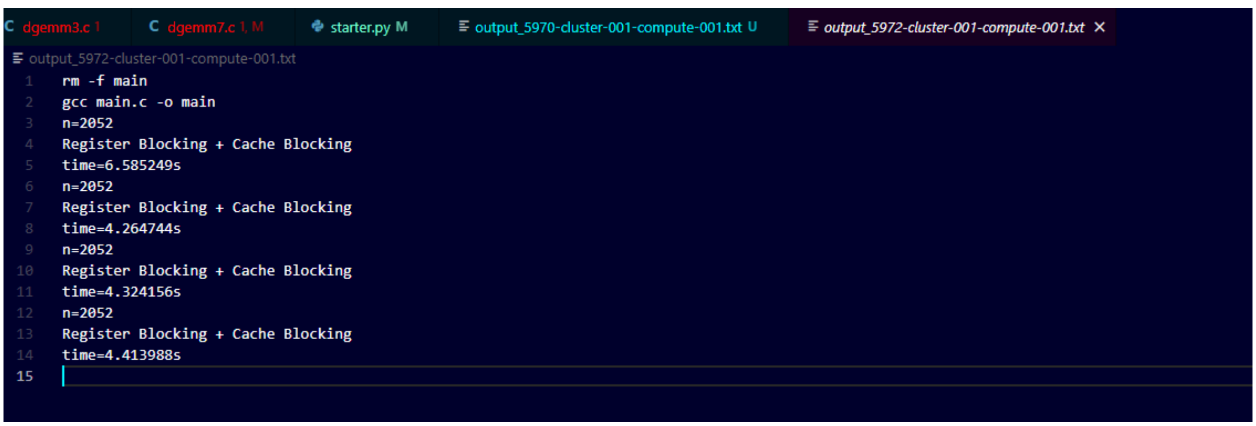
For all blocked versions the runtime is faster than all non blocked versions. The fastest blocked versions is ijk and jik while the fastest non blocked versions are ikj and kij. This is because there are less cache misses for the blocked version.

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 analyze the result.

See github for code.
I combined Cache Blocking and Register blocking using a Cache Block size of 36^2 and 15 registers.

The final result is this:




```
rm -f main
gcc main.c -o main
n=2052
Register Blocking + Cache Blocking
time=6.585249s
n=2052
Register Blocking + Cache Blocking
time=4.264744s
n=2052
Register Blocking + Cache Blocking
time=4.324156s
n=2052
Register Blocking + Cache Blocking
time=4.413988s
```

With the fastest run time with -O1 of 4.2647 seconds.

This is faster than cache blocking with all ijk versions and register blocking. The fastest achieved from the algorithms shown in lecture is dgemm3 with 6.9 seconds for a matrix size of 2048^2.

The exercises demonstrated the importance of memory hierarchy optimization and code level optimization in achieving high performance matrix multiplication.

dgemm7	
# registers	Time (seconds)
12	5.85653
15	5.392827
22	5.702219
24	5.898156

For number of optimal registers the options were 2 x 2 (12 registers), 3 x 3 (15 registers) , and 4 x 4 (24 registers)and higher. Since 4 x 4 was slower than 3 x 3 I knew there wasn't enough Floating point registers to support 4 x 4 as the more registers should mean faster performance. 3 x 3 (15 registers) was faster than 2 x 2 (12 registers) therefore I chose 15 registers.

For the best cache block refer to problem number 6.