Name: Alphonse Sun
ID: 862549253

1. 9.Q1
   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.
      - Reasons:
        For the code in dgemm0, since four double-precision floating-point operations can be performed in one cycle, the addition, subtraction, multiplication, and division of the variables in the innermost loop can be considered as an additional cycle. Therefore, disregarding the latency caused by data being read from memory, an extra clock cycle should be added. For the innermost code of the dgemm0 algorithm, this line of code performs four memory reads, with each read causing a delay of 100 clock cycles. Therefore, when the variable n=1000, the total time spent is given by the following formula:
        $$T = (1 + 100 * 2) * n\text{^}3 * 1 / (2 * G)$$
        As a result, the total time of dgemm0 spent is 200.5 seconds.

        On the other hand, for the code in dgemm1, Since the dgemm1 algorithm uses registers to store the variables of matrix C, the delay in clock cycles caused by the innermost loop of the code is (100*2+1)*n^3. Additionally, even though registers are used, memory still needs to be accessed, and after completing the matrix computation, the result also needs to be written back to memory. Therefore, the clock delay caused by this part is 2*100*n^2. In summary, the final time spent by dgemm1 is given by the following formula:
        $$T = (2 * 100 * n^2 + n^3 * (2 * 100 + 1))/(2 * G)$$
        As a result, the total time of dgemm1 spent is 100.6 seconds.

      - Reasons:
        To calculate the time taken for read and write operations from memory, we only need to consider the number of times memory is accessed in the dgemm0 and dgemm1 algorithms, and multiply that by the corresponding clock cycle delay and number of accesses. The principle is the same as in the previous question. Therefore, the final results are as follows: the time spent on memory read and write operations in the dgemm0 code is 200 seconds, while in the dgemm1 code it is 100.1 seconds.

   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.
      -

	N=64	N=128	N=256	N=512	N=1024	N=2048

Dgemm0	0.000996s	0.008804s	0.080969s	1.183371s	8.970921s	176.797448s
Dgemm1	0.000650s	0.005483s	0.055476s	0.765435s	5.864238s	123.289336s

- 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.
  Reasons:
  According to the instructions given in the problem, the formula for calculating Gflops is as follows.

$$Gflops = \frac{Total\ Flops}{Execution\ Time(seconds) \times 10^9}$$

  Based on the previously measured execution times, the performance table for **dgemm0** and **dgemm1** can be derived as follows:

	N=64	N=128	N=256	N=512	N=1024	N=2048
Dgemm0	0.526 Gflops	0.476 Gflops	0.414 Gflops	0.227 Gflops	0.239 Gflops	0.097 Gflops
Dgemm1	0.807 Gflops	0.765 Gflops	0.605 Gflops	0.351 Gflops	0.366 Gflops	0.139 Gflops

2. Q2
   - 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.
     Reasons:
     The table of spent time of code dgemm2:

	N=64	N=128	N=256	N=512	N=1024	N=2048
Dgemm2	0.000240s	0.002016s	0.017505s	0.224138s	1.684574s	36.722402s

     The table of performance of the code dgemm2:

	N=64	N=128	N=256	N=512	N=1024	N=2048
Dgemm2	2.185 Gflops	2.081 Gflops	1.917 Gflops	1.198 Gflops	1.275 Gflops	0.468 Gflops

     Based on the previously measured execution times, the average performance for **dgemm2** can be derived as follows:

     The average result of performance is nearly **1.520** Gflops.

3. Q3
   - Reasons:
     Since we have a maximum of 16 registers and aim to utilize register reuse to

the greatest extent, we can use 3x3 matrix blocking to accelerate matrix multiplication. By maximizing register reuse, the related variables of matrix A in the innermost loop can be limited to 3 registers, and the related variables of matrix B can also be limited to 3 registers. Outside the innermost loop, 9 registers are allocated for storing and reading the result matrix C variables. Therefore, the total number of registers used is 15, which is less than the required 16. This solution satisfies the problem requirements.

In summary, the following table shows the test results for the dgemm3 code.

Execution Time

	N=64	N=128	N=256	N=512	N=1024	N=2048
Dgemm3	0.000189s	0.001367s	0.011827s	0.125445s	0.830550s	7.283626s

Performance

	N=64	N=128	N=256	N=512	N=1024	N=2048
Dgemm3	2.774 Gflops	3.068 Gflops	2.837 Gflops	2.140G flops	2.586 Gflops	2.359 Gflops

Based on the previously measured execution times, the average performance for **dgemm3** can be derived as follows:

The average result of performance is nearly 2.627 **Gflops**.

- Comparison:

  Comparing the performance of dgemm2 (12 registers) and dgemm3 (15 registers) in matrix multiplication, the results show that dgemm3 holds a clear advantage for larger matrices. By utilizing more registers, dgemm3 processes 3×3 matrix blocks at a time, reducing memory access and improving cache hit rates and computational efficiency, even though register reuse still involves some memory access. In contrast, dgemm2 suffers a significant performance drop when handling larger matrices due to the limited number of registers. Overall, the number of registers and their reuse have a substantial impact on performance, especially in large-scale matrix multiplication, where more registers and optimized block processing strategies can greatly enhance computational efficiency.

4. Q4

- When the matrix size is 10*10:

Loop Order	A		B		C		Overall Miss Rate
	#of read	#of miss	#of read	#of miss	#of read	#of miss	
ijk	1000	10	1000	10	100	10	0.0142
ikj	100	10	1000	10	1000	10	0.0142
jik	1000	10	1000	10	100	10	0.0142
jki	1000	10	100	10	1000	10	0.0142
kij	100	10	1000	10	1000	10	0.0142
kji	1000	10	100	10	1000	10	0.0142

Assuming a cache with 60 lines, each capable of holding 10 double values, is

utilized in your program. Therefore, when the matrix size is 10x10, all matrices can fit into the cache. As a result, the cache miss count for each matrix is 10, leading to a total miss rate of 0.0142.

- When the matrix size is 10000*10000:

Loop Order	A		B		C		Overall Miss Rate
	#of read	#of miss	#of read	#of miss	#of read	#of miss	
ijk	10^{12}	10^{11}	10^{12}	10^{12}	10^{8}	10^{7}	0.55
ikj	10^{8}	10^{7}	10^{12}	10^{11}	10^{12}	10^{11}	0.1
jik	10^{12}	10^{11}	10^{12}	10^{12}	10^{8}	10^{7}	0.55
jki	10^{12}	10^{12}	10^{8}	10^{7}	10^{12}	10^{12}	1
kij	10^{8}	10^{7}	10^{12}	10^{11}	10^{12}	10^{11}	0.1
kji	10^{12}	10^{12}	10^{8}	10^{7}	10^{12}	10^{12}	1

For Ijk Loop,

5. Q5

- When the matrix size is 10000*10000(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	10^{12}	10^{10}	10^{12}	10^{10}	10^{8}	10^{7}	0.01
Ikj-ikj	10^{8}	10^{7}	10^{12}	10^{10}	10^{12}	10^{10}	0.01
Jik-jik	10^{12}	10^{10}	10^{12}	10^{10}	10^{8}	10^{7}	0.01
Jki-jki	10^{12}	10^{10}	10^{8}	10^{7}	10^{12}	10^{10}	0.01
Kij-kij	10^{8}	10^{7}	10^{12}	10^{10}	10^{12}	10^{10}	0.01
Kji-kji	10^{12}	10^{10}	10^{8}	10^{7}	10^{12}	10^{10}	0.01

Reasons:

In the case of using blocked GEMM algorithms with single cache reuse and a block size of 10x10, the number of cache misses for a single matrix within each block can be calculated as $B^2/10$, where B is the side length of the block. Since the matrix is divided into multiple blocks, the total number of blocks is$(n/B)^2$, where n is the size of the matrix and b is the block size. Therefore, for example, for matrix AAA or matrix BBB in an "ijk" loop order, the cache misses can be computed as $(B^2/10)*(n/B)*(n/B)^2$. Additionally, For matrix C in the IJK loop, it will only be read n^2 times. However, since blocks are used and the matrix is read or stored row by row, the cache misses for a single block are B/10. Therefore, the total cache misses for matrix C are (B/10)×(n/B)^3×B^2.

- For the kji-kji loop order, cache msses will occur 1000times for both A[0][0] and C[68][90]. A[17][21], C[2000][1297] and B[101][134] will not go through cache misses. For B[100][130], it will be only once cache miss.

6. Q6

- The followings are some result of performance and comparsion between the block versions and non-blocked versions of the algorithm.

- Block Version block-size=10*10

	Performance/Gflops	Execution time/Seconds
Ijk-ijk	0.645	26.643715s
Jik-jik	0.719	23.908847s
Jki-jki	0.521	33.004403s
Ikj-ikj	0.585	29.380575s
Kij-kij	0.617	27.835772s
Kji-kji	0.664	25.863047s

- Non-Block Version

	Performance/Gflops	Execution time/Seconds
Ijk-ijk	0.128	134.291661s
Jik-jik	0.135	127.307848s
Jki-jki	0.091	189.575215s
Ikj-ikj	0.667	25.389531s
Kij-kij	0.674	25.483718s
Kji-kji	0.227	75.53786s

Reasons:

Based on the data results, it can be concluded that the main difference between the Block and non-Block versions of the algorithm lies in the fact that the Block version optimizes cache misses to the greatest extent. In contrast, the non-block version exacerbates cache misses due to different loop strategies in the algorithm. Therefore, using cache blocking can minimize cache misses and is not affected by the loop strategy.

- The result of different block sizes to calculate the 2048*2048 matrix .

Block Sizes:	Performance/Gflops	Execution time/Seconds
2*2	0.168	101.958735s
5*5	0.456	37.680256s
8*8	0.586	29.302666s
15*15	0.683	25.161532s
20*20	0.746	23.040995s
50*50	0.770	22.303512s

Reasons:

From the results, it can be concluded that as the block size increases, the performance of matrix multiplication gradually improves, with GFLOP/s increasing from 0.168 for the 2×2 block to 0.770 for the 50×50 block. Smaller block sizes result in lower cache utilization, leading to more cache misses and thus reducing performance. As the block size increases, cache hit rates improve, and performance is optimized. Therefore, using an appropriate block size (such as 20×20 or 50×50) can effectively reduce cache misses and enhance the overall performance of matrix multiplication.

7. Q7

- Combine cache blocking and register blocking to implement a matrix multiplication of size 2048*2048
  Cache block size:30*30
  register block size:3*3

The result of execution time

	Execution time/seconds
O0	6.958895s
O1	3.335733s
O2	3.203894s
O3	3.281398s

In register optimization, as the number of registers increases and the size of the register blocks expands, the algorithm can significantly reduce the frequency of cache accesses, thus improving computational efficiency. Effective use of registers allows more data to be stored within the processor, reducing dependency on the slower memory hierarchy. However, since the number of registers is limited, excessively large register blocks may lead to register overflow or diminishing returns in efficiency.

For cache blocking optimization, increasing the size of the cache blocks effectively reduces cache misses, thereby lowering memory access latency and shortening clock cycles. Larger cache blocks allow more data to reside in the cache at once, minimizing the need for frequent memory loads. However, as the cache block size continues to grow, the performance gains diminish. This occurs because when the cache block size approaches the capacity of the hardware cache, further enlarging the block provides minimal benefit.

GitHub repository: https://github.com/AlphonseSun1223/CS211-hw-assignment.git