University of California, Riverside
Department of Computer Science & Engineering

Title: **High-Performance Computing – Project 1**

Student Name:
Mahbod Afarin

Student ID:
862186340

Fall 2020

1- Register Reuse

Part 1.

Wasted time for dgemm0:

In the innermost loop, we have 4 access to the memory. There are 3 loads from loading $a[i \times n + k]$, $b[k \times n + j]$, and $c[i \times n + j]$. In addition of 3 loads, we have one store for $c[i \times n + j]$. Each access to the memory will takes 100 cycles and also, we have 3 loops with $n = 1000$. Therefore, the total cycles to access the memory will be $4 \times 100 \times 10^9$. The clock frequency for the computer is $2Ghz$; Therefore, the wasted time will be:

$$Number\ of\ the\ cycles = (4 \times 100 \times 10^9) = 4 \times 10^{11}\ cycles$$
$$Wasted\ Time = (4 \times 100 \times 10^9) \times \frac{1}{2 \times 10^9} = 200\ Sec$$

Total time for dgemm0:

We have 2 floating-point instruction and the computer can complete 4 double floating-point instruction per cycle. Therefore:

$$Total\ Time = 200 + \left(\frac{2}{4}\right) \times 10^9 \times \frac{1}{2 \times 10^9} = 200.25\ Sec$$

Wasted time for dgemm1:

We have load in the innermost loop from $a[i \times n + k]$ and $b[k \times n + j]$ and also 1 load and 1 store in the second innermost loop. Therefore, the number of the cycles will be $(2 \times 100 \times 10^3) + (2 \times 100)] \times 10^6$ and the wasted time will be:

$$Number\ of\ the\ cycles = [(2 \times 100 \times 10^3) + (2 \times 100)] \times 10^6 = 2.002 \times 10^{11}\ cycles$$
$$Wasted\ Time = [(2 \times 100 \times 10^3) + (2 \times 100)] \times 10^6 \times \frac{1}{2 \times 10^9} = 100.1\ Sec$$

Total time for dgemm1:

We have load in the innermost loop from $a[i \times n + k]$ and $b[k \times n + j]$ and also 1 load and 1 store in the second innermost loop. Also, there will be 2 cycles in the last two innermost loops for arithmetic operation. Therefore:

$$Total\ Time = 100.1 + \left(\frac{2}{4}\right) \times 10^9 \times \frac{1}{2 \times 10^9} = 100.35\ Sec$$

I implemented the two algorithms in the frame work and check the correctness of two algorithm. I have shown the results of the implementation in the below tables. For calculating the giga floating-point operation per seconds, we have 2 floating-point operation in the innermost loop and we should multiply it in the iteration count and then we should divide it by the time and finally divide it by the 1000000000. Therefore, the performance for the dgemm0 and dgmm1 will calculate through this equation.

$$Performance = \frac{2 \times n^3}{t} \times \frac{1}{1000000000}$$

We can clearly see from the tables that dgemm2 has a better performance compare to the dgemm0 because of the register reuse technique. x

Table 1: Performance and execution time for dgemm0

N	Execution Time (Seconds)	Performance (GFLOPS)
66	0.00179	0.321224581
126	0.01230	0.325264390
258	0.10685	0.317713055
510	1.20540	0.2200945744
1026	8.90081	0.242684784
2046	81.69465	0.209678194

Table 2: Performance and execution time for dgemm1

N	Execution Time (Seconds)	Performance (GFLOPS)
66	0.00085	0.6764611765
126	0.00363	1.1021355372
258	0.03521	0.9754905993
510	0.49131	0.539989009
1026	3.97940	0.5428183023
2046	38.18139	0.448637063

Part 2.

In the table 3, we can see the result of the implementation dgemm2. As we can see, with 12 registers (4 registers for A, 4 registers for B, and 4 registers for C) we have a better performance compared to the previous versions.

Table 3: Performance and execution time for dgemm2

N	Execution Time (Seconds)	Performance (GFLOPS)
66	0.00023	2.4999652174
126	0.00195	2.0516676923
258	0.01930	1.7796385492
510	0.22108	1.2000271395
1026	2.41264	0.8953226142
2046	30.21038	0.5686743822

Part 3.

In the table 4, we can see the result of the implementation dgemm2. As we can see, with 16 registers we have a better performance compared to the previous versions.

Table 4: Performance and execution time for dgemm3

N	Execution Time (Seconds)	Performance (GFLOPS)
66	0.00027	2.1296
126	0.00141	2.837412766
258	0.01210	2.8385970248
510	0.10674	2.4854974705
1026	1.02954	2.098112897
2046	7.20772	2.3765610584

In the figure1, we can see the execution time of 4 difference scenario. As it shown in the figure, dgemmo have the biggest average execution time and the dgemm3 has the smallest average execution time.

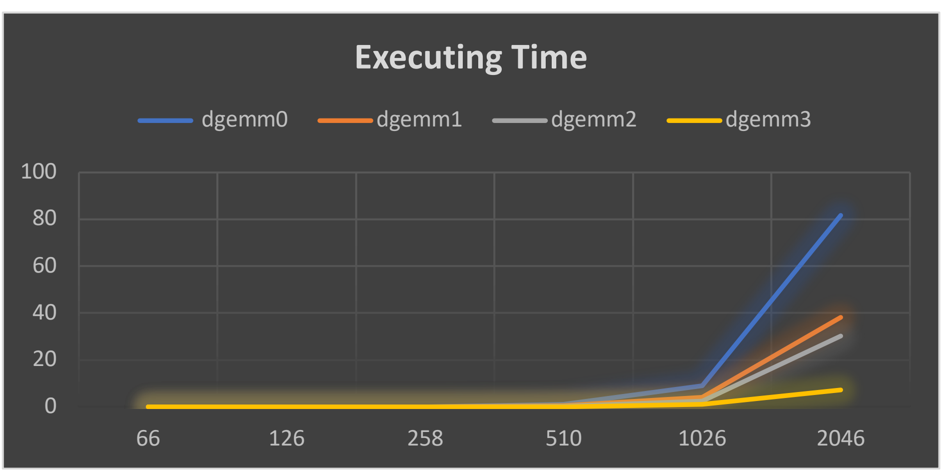

Figure 1: Executing time for 4 different scenarios

2- Cache Reuse

Part 1.

A- Matrix 10X10:

1- For ijk & jik: The cache line size is 10. We will have one cache miss for a_{11} at first and 10 cache misses for b_{i1} which i will starts from 1 to 10. When we go to the second row, we have another cache miss for a_{21}, so we have 1 miss per row for every a_{i1}element. In overall there will be 10 misses for $a_{i1}$and 10 misses for b_{i1}. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{10 + 10}{2 \times 10^3} = 0.01 = 1\%$$

2- For jki & kji: we will have 10 cache misses for b_{i1} and 10 cache misses for c_{i1} in the innermost loop. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{10 + 10}{2 \times 10^3} = 0.01 = 1\%$$

3- For kij & ikj: we will have 10 cache misses for a_{i1} and 10 cache misses for c_{i1}. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{10 + 10}{2 \times 10^3} = 0.01 = 1\%$$

A- Matrix 10000X10000:

1- For ijk & jik: The cache size is 60 and the size of the matrix is 10000X10000, so we have miss in read of the all elements of matrix B which is 10^{12}. We have miss for matrix A in every 10 columns of A in $a_{i(10k+1)}$ and there will be 10^{11} misses for A. Therefore:

$$Miss\ Rate = \frac{10^{12} + 10^{11}}{2 \times 10^{12}} = 0.55 = 55\%$$

2- For jki & kji: B and C have cache misses in the element of $b_{i(10k+1)}$ and $c_{i(10k+1)}$ respectively. We will read every element n times; therefore, we will have 10^{11} cache misses for both B and C. Thus:

$$Miss\ Rate = \frac{10^{11} + 10^{11}}{2 \times 10^{12}} = 0.1 = 10\%$$

3- For kij & ikj: The cache size is much smaller than the n, so every cache reference will be faced with miss. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{10^{12} + 10^{12}}{2 \times 10^{12}} = 1 = 100\%$$

Table 5: Comparing the miss rate for the algorithms for 10X10 and 10000X10000

Algorithms	% of Miss Rate for 10	% of Miss Rate for 10000
ijk	1%	55%
jik	1%	55%
kij	1%	10%
ikj	1%	10%
jki	1%	100%
kji	1%	100%

Part 2.

1- For ijk & jik: Number of misses per element for $a_{i(10k+1)}$ which i is from 1 to 10000 is 10 and for $b_{i(10k+1)}$ which i is from 1 to 10000 is 10. The 10000×10000 matrix is consisting of the 10×10 blocks because the size of our block is 10. Thus, we have 1000 blocks for our matrix. The total number of reads will be 2×10^{12} and the number of the misses for each block is 20 for each algorithm. The total number of misses per element is 20000 and the number of misses is 2×10^{10}. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{2 \times 10^{10}}{2 \times 10^{12}} = 0.01 = 1\%$$

2- For jki & kji: Number of misses per element for $b_{i(10k+1)}$ which i is from 1 to 10000 is 10 and for $c_{i(10k+1)}$ which i is from 1 to 10000 is 10. The 10000×10000 matrix is consisting of the 10×10 blocks because the size of our block is 10. Thus, we have 1000 blocks for our matrix. The total number of reads will be 2×10^{12} and the number of the misses for each block is 20 for each algorithm. The total number of misses per element is 20000 and the number of misses is 2×10^{10}. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{2 \times 10^{10}}{2 \times 10^{12}} = 0.01 = 1\%$$

3- For kij & ikj: Number of misses per element for $a_{i(10k+1)}$ which i is from 1 to 10000 is 10 and for $c_{i(10k+1)}$ which i is from 1 to 10000 is 10. The 10000×10000 matrix is consisting of the 10×10 blocks because the size of our block is 10. Thus, we have 1000 blocks for our matrix. The total number of reads will be 2×10^{12} and the number of the misses for each block is 20 for each algorithm. The total number of misses per element is 20000 and the number of misses is 2×10^{10}. Therefore, the miss rate will be:

$$Miss\ Rate = \frac{2 \times 10^{10}}{2 \times 10^{12}} = 0.01 = 1\%$$

Table 6: Number of misses and the percent of misses for blocking technique

Algorithms	Number of Misses	% of Miss Rate
ijk	2×10^{10}	1%
jik	2×10^{10}	1%
kij	2×10^{10}	1%
ikj	2×10^{10}	1%
jki	2×10^{10}	1%
kji	2×10^{10}	1%

Part 3.

In the below table we can see the execution time for different algorithms. As we can see in the table, with the blocking techniques we can get a better performance compared to the previous algorithm. The block size is 10 and the matrix size is 2000.

Table 7: Execution time for different algorithms

Algorithms	Execution Times (Seconds)
ijk	21.76009
jik	18.43270
kij	15.50405
ikj	15.59776
jki	48.59914
kji	50.99701
Blocking ijk	15.36631
Blocking jik	15.63186
Blocking kij	15.65540
Blocking ikj	16.77092
Blocking jki	15.22280
Blocking kji	14.72226

In the below table, we can see the results of the different blocking size of the 6 algorithms with the matrix size of 2048. As we can see in the below table, in overall, the best performance is for block size 64.

Table 8: Execution time for different block size for N = 2048

Algorithm	BS = 8	BS = 16	BS = 32	BS = 64	BS = 128	BS = 256
Bijk	28.03076	40.80488	49.05894	31.45562	28.03076	136.23169
Bjik	29.69498	44.84190	57.59646	33.70540	29.69498	141.20657
Bkij	15.49741	22.28900	27.94057	19.51716	15.49741	21.65463
Bikj	16.08998	26.56210	30.19341	20.93875	16.08998	24.33307
Bjki	245.10074	207.25071	188.73841	246.66377	245.10074	415.10629
Bkji	244.75277	220.18554	193.42411	236.24918	244.75277	429.00440

Part 4.

In the below table we can see the results for combining the both blocking cache reuse and register reuse for different optimization flag. We set the block size to 64 with register block 2 and n = 2048. The best performance is for O1.

Table 9: Results for 4 different optimization flags

-O0	-O1	-O2	-O3
5.15981	5.14425	5.14943	5.14572