CS 211: High Performance Computing Project 1

Zhenyu Yang (862187998)

Fall 2019

1.Register Reuse

1.1 Part 1

Assumptions:

1. 4 double floating-point operations per cycle when operands are in registers.
2. 100 cycles delay to read/write one operand from/to memory.
3. clock frequency is 2 Ghz.

To calculate the time for **dgemm0** and **dgemm1** when n= 1000, we need to count the number of floating-point operations and read/write operations for each algorithm. The results are shown in the following table.

Table 1: operations numbers

	read	write	multiplication	addition	**total circles**	**runtime(s)**
dgemm0	$3n^3$	n^3	n^3	n^3	$400.5n^3$	200.25
dgemm1	$2n^3 + n^2$	n^2	n^3	n^3	$200.5n^3 + 200n^2$	100.35

The result shows that memory-access operations wasted a lot of time.
For **dgemm0**, there is **200s** is wasted in **200.25s**.
For **dgemm1**, there is **100.1s** is wasted in **100.35s**.

Experiments:
To measure the performance of **dgemm0** and **dgemm1**, these two algorithms are tested by different Matrix size n. Using Gflops as metric, which means, we need to count all floating-point operations and to divide the count number by running time for each algorithm. The results are shown in the following tables.

Table 2: **dgemm0** Performance

n	runtime(s)	performance(Gflops)
66	0.00322	0.17857
126	0.01232	0.32474
258	0.10701	0.32097
510	1.21821	0.21778
1026	8.89407	0.24287
2046	83.32261	0.20558

Table 3: **dgemm1** Performance

n	runtime(s)	performance(Gflops)
66	0.00054	1.06480
126	0.00365	1.09610
258	0.03553	0.96670
510	0.50519	0.52515
1026	4.20654	0.51351
2046	38.76792	0.44185

1.2 Part 2

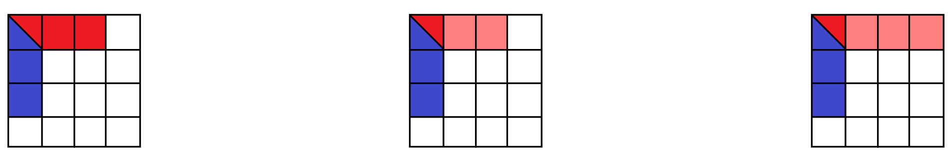
dgemm2 tries to use 12 registers to improve the Matrices Multiplication's performance. The experiments results are shown as follows.

Table 4: **dgemm2** Performance

n	runtime(s)	performance(Gflops)
66	0.00022	2.61360
126	0.00170	2.35338
258	0.01925	1.78426
510	0.20846	1.27268
1026	2.61031	0.82752
2046	30.51656	0.56132

1.3 Part 3

There is 3 algorithms which are designed to show and to validate the idea of **Register Reuse**. First, **dgemm3_3x4** tries to use **16** registers in a **3x4** block matrices multiplication. Then, **dgemm3_3x3_v2** tries to use **13** registers in a **3x3** block matrices multiplication. Last, **dgemm3** tries to use **15** registers in a **3x3** block matrices multiplication. The experiments' results are shown as follows.



(a) **dgemm3** (b) **dgemm3_3x3_v2** (c) **dgemm3_3x4**

Figure 1: Different types of dgemm3

Table 5: **dgemm3**

n	runtime
64	0.00020
128	0.00154
256	0.01252
512	0.16585
1024	2.02078
2048	24.83506
2046	6.46601
2040	11.92055

Table 6: **dgemm3_3x3_v2**

n	runtime
64	0.00015
128	0.00158
256	0.01152
512	0.15851
1024	1.89364
2048	24.59618
2046	6.31309
2040	10.57885

Table 7: **dgemm3_3x4**

n	runtime
64	0.00016
128	0.00140
256	0.01162
512	0.16226
1024	1.81243
2048	23.61749
2046	6.90081
2040	12.66067

1.4 Results

Comparing **dgemm0**, **dgemm1**, **dgemm2**, **dgemm3**, **dgemm3_3x3_v2** and **dgemm3_3x4**, there is a line chart shown performance differences between those six algorithms with different matrix size n.

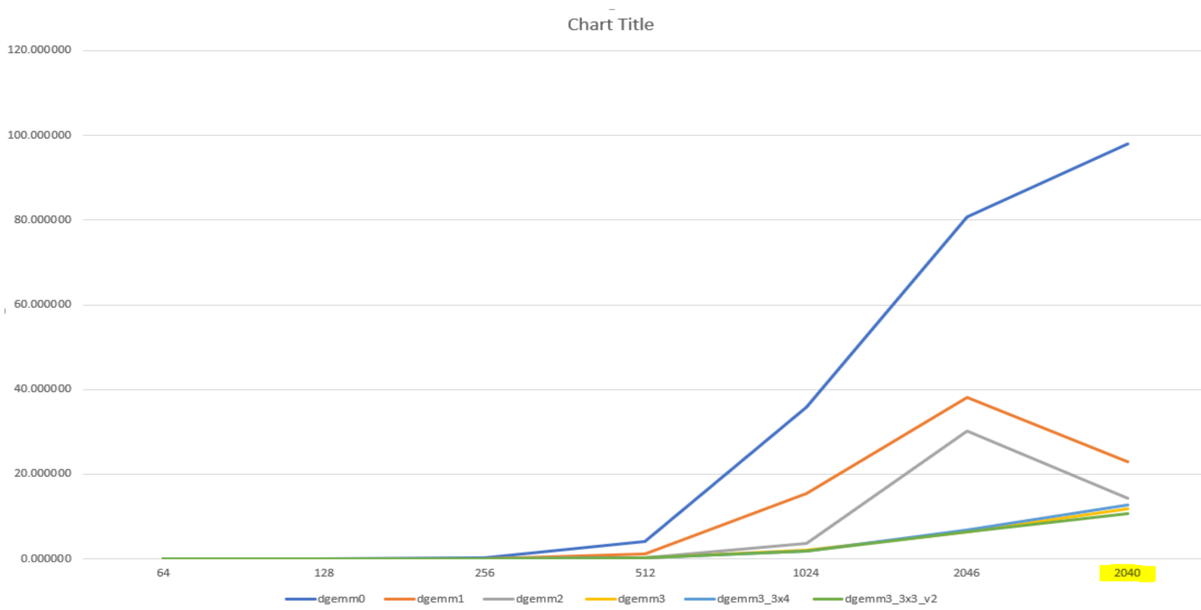


Figure 2: For different n

The results show that **dgemm3_3x3_v2(13 registers)** may have better performance.
Another problem is, although the **2040** blocksize can be divided by **3x3** grid, and it requires **less** computations than **2046**, it still costs **more** than **2046**.

2. Cache Reuse

2.1 Part 1

Assumptions:

1. Cache size is 60 lines and each line can hold 10 doubles.
2. Cache replacement rule is least recently used firs.

First, a line of $\mathbf{C}[i \times n + j]$ is cached before third loop.
Second, a line of $\mathbf{A}[i \times n + j]$ and $\mathbf{B}[i \times n + j]$ is cached in third loop.
Third, during the third loop, there are a lot of $\mathbf{A}[i \times n + j]$ and $\mathbf{B}[i \times n + j]$ are cached. So, the cached line of $\mathbf{C}[i \times n + j]$ is going to be deprecated if the cache is full of $\mathbf{A}[i \times n + j]$ and $\mathbf{B}[i \times n + j]$, means if the row size of matrices beyond cache size, the number of cache misses of matrix **C** will be large.
Last, $\mathbf{C}[i \times n + j]$ are invoked again after third loop. if $\mathbf{C}[i \times n + j]$ elements are existed in cache, there is no need to cache $\mathbf{C}[i \times n + j]$ elements again. But if previously cached $\mathbf{C}[i \times n + j]$ elements has been replaced, caching $\mathbf{C}[i \times n + j]$ elements is necessary.

The cache misses analysis for each algorithm is shown as below.

Table 8: cache misses (n = 10)

	A$[i*n+k]$	B$[k*n+j]$	C$[i*n+j]$	total miss	total read	**misses rate**
IJK	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%
IKJ	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%
JIK	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%
JKI	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%
KIJ	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%
KJI	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	1 (at the first time read this row) 0 total is 10	30	2100	1.42857%

analysis : For n = 10, the matrices size are too small to be deprecated in cache. The cache has 60 lines, but it only need 30 lines to store the whole A, B and C. So, there are only read cache misses during first time read the rows. For n = 10000, a row of matrix need 1000 cache lines to store, but we only have 60 cache lines, which means there is no entire row can be reused in cache and every row data(no matter it is used before or not) will be missed in cache. for every row-wise matrix has 10% read cache misses rate. for every column-wise matrix has 100% read cache misses rate.

Table 9: cache misses (n = 10000)

	A$[i*n+k]$	B$[k*n+j]$	C$[i*n+j]$	total miss	total read	**misses rate**
IJK	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	10^4 total is 10^{12}	1 total is 10^8	$10^{11} + 10^{12} + 10^8$	$2 * 10^{12} + 10^8$	55.0023%
IKJ	1 total is 10^8	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	$10^{11} + 10^{11} + 10^8$	$2 * 10^{12} + 10^8$	10.0050%
JIK	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	10^4 total is 10^{12}	1 total is 10^8	$10^{11} + 10^8 + 10^{12}$	$2 * 10^{12} + 10^8$	55.0023%
JKI	10^4 total is 10^{12}	1 total is 10^8	10^4 total is 10^{12}	$10^{12} + 10^8 + 10^{12}$	$2 * 10^{12} + 10^8$	100.000%
KIJ	1 total is 10^8	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	$10^4 (k\%10 = 0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{11}	$10^{11} + 10^{11} + 10^8$	$2 * 10^{12} + 10^8$	10.0050%
KJI	10^4 total is 10^{12}	1 total is 10^8	10^4 total is 10^{12}	$10^{12} + 10^8 + 10^{12}$	$2 * 10^{12} + 10^8$	100.000%

2.2 Part 2

1. Each block is a 10*10 matrix.
2. Inner three loops uses the same loop order as the three outer loops

The cache misses analysis for each algorithm is shown as below.

Table 10: cache misses for block algorithm (n = 10000)

	A$[i*n+k]$	B$[k*n+j]$	C$[i*n+j]$	total miss	total read	**misses rate**
BIJK	$10^3(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$1(j\%10=0)$ $0(j\%10 \neq 0)$ total is 10^7	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%
BIKJ	$1(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^7	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%
BJIK	$10^3(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$1(j\%10=0)$ $0(j\%10 \neq 0)$ total is 10^7	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%
BJKI	$10^3(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^7	$1(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%
BKIJ	$1(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^7	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%
BKJI	$10^3(k\%10=0)$ $0 \quad (k\%10 \neq 0)$ total is 10^7	$1(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^3(j\%10=0)$ $0 \quad (j\%10 \neq 0)$ total is 10^{10}	$10^{10}+10^{10}+10^7$	$2*10^{12}+10^8$	1.00045%

analysis :
For block algorithms, every block matrix has its own cache reuse. Even total cache size can hold 6 10*10 matrices block, it is too small for n = 10000. So, the previously cached data has been deprecated when next time read same elements. for row-wise matrix, every row has 10% read cache misses rate.for every column-wise matrix also has 10% read cache misses rate because the whole block is cached. The cache misses number of reading a 10*10 block is $10 * \frac{10}{10} = 10$.

2.3 Part 3

The following chart shows the difference by using different n for variant block sizes to measure the performance for block algorithms and non-block algorithms.

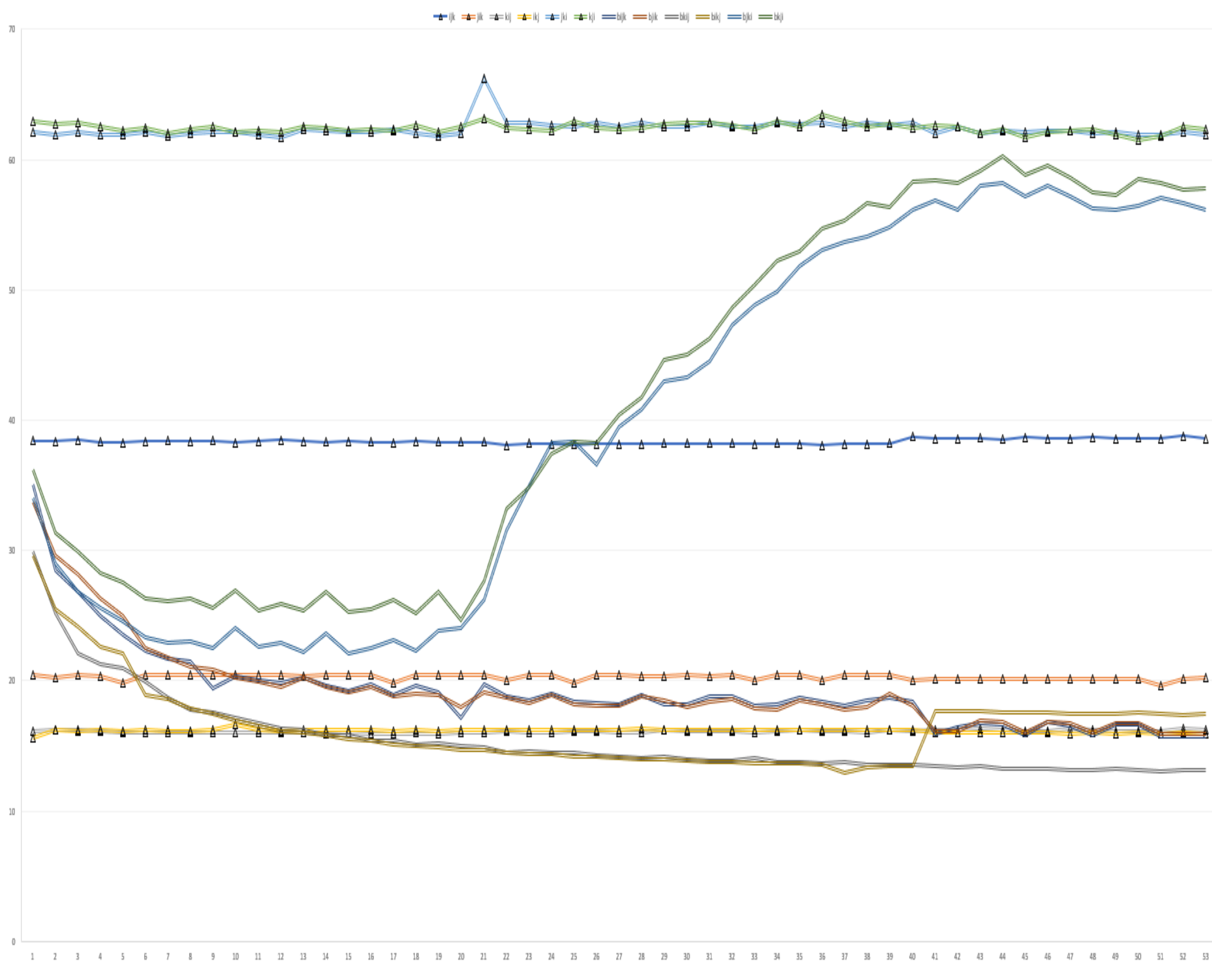


Figure 3: For different n

The results show that **the best block size n is around 40*3 + 9 = 129(the x-axis is the number of for-loop, which start in 9)**, and it also tells that the **bijk, bjik, bkij, bikj** could has nearly performance. And as us previous computations, the results of non-block versions **ijk, ikj, jki,jik, kij,kji** are basically following our expectation.

2.4 Part 4

To combine the cache blocking algorithm and registers blocking, this algorithm uses **dgemm3_3x3_v2(13** registers)in a **129x129** block matrices multiplication. The experiments' results are shown as follows.

```
*********** Cache reuse - Part 4. ***********
O0 : elapsed time is 10.83420 second(s).
O1 : elapsed time is 10.86322 second(s).
O2 : elapsed time is 10.89277 second(s).
O3 : elapsed time is 10.87716 second(s).
```

3. Strassens algorithm

Strassens algorithm is very innovation and very effective if n is integer power of 2, so I only use the $n = 2^i$ to experiment and to compare the performance with previous algorithm. The experiments' results are shown as follows.

Unfortunately, the Strassens algorithm suffered a lot about precision lost during large amount of recursion, which leads that it cannot get exactly computations without any deviation(maybe too large). And its performance also does not better than naive matrices multiplication version. Here is the result.

```
incorrect. -17279.399700 --17240.779700 ,bias = 38.620000
error detected at function strassen, matrix size = 64.
incorrect. 20631.981500 -20670.601500 ,bias = 38.620000
error detected at function strassen, matrix size = 128.
incorrect. 32366.794200 -32405.414200 ,bias = 38.620000
error detected at function strassen, matrix size = 256.
incorrect. -27622.741900 --27633.501900 ,bias = 10.760000
error detected at function strassen, matrix size = 512.
incorrect. -194176.261900 --194226.141900 ,bias = 49.880000
error detected at function strassen, matrix size = 1024.
strassen -   64: elapsed time is  0.00482 second(s).
strassen -  128: elapsed time is  0.03344 second(s).
strassen -  256: elapsed time is  0.22343 second(s).
strassen -  512: elapsed time is  1.44139 second(s).
strassen - 1024: elapsed time is 10.09701 second(s).
```