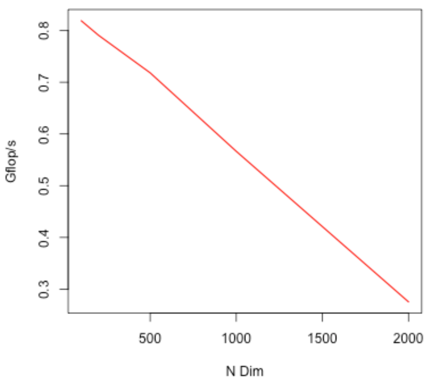
HW3 ECS 230

Heqiao Ruan
email:hruan@ucdavis.edu

November 13, 2018

1 Part 1

For the matrix multiplication, C=AB where both A,B,C are all n by n matrix we write it in the detail: $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$. For randomly initialization of A and B, we initialize it as $A_{ij} = \frac{j}{i+n*j+2}$,$B_{ij} = \frac{i}{j+n*i+1}$.**For robustness, the performance measurement for matrix multiplication is replicated for 9 times for each trial for 100,200,500,1000 and 4 times for size 2000 and take the average of these runs**. For command line, we
Theoretically for matrix multiplication with n by n matrix the total number of the floating point operations is about $2n^3$ which it include the three nested loops and a add operation followed by each multiplication. Let's take 100 by 100 matrix for example, it costs 2000000 operations. Then for the floating point operations per second, we can see that, it takes.
For the Gflops per second, we can see that for 1000 by 1000 matrix, it costs approximately 3.53 seconds which the Gflops here is calculated by $\frac{2*1000^3}{3.53*1e9}$ and is approximately 0.56Gflop/s. The plot of Gflops versus dimension n and table here is shown as below:



n(dim)	clock cycles	gflop/s
100	8776536	0.8188
200	73289624	0.7908473
500	1250310920	0.7181731
1000	12682434440	0.5664039
2000	208683857321	0.2753790

For estimating the peak performance,as the instruction shows, it may depend on a number of factors including SMID registers, fused add-multiply, hyperthreading and even 'turbo boost'. So the typical number of flop every clock cycle ordinarily between 1 and 5 then we can see that the ordinary routine is. The CPU is Inter(R) Core(TM) i7-4790 with 3.60 GHz. The theoretical peak performance may be calculated by the equation, it depends on SIMD registers, fused add-multiply, hyperthreading and turbo boost and the peak Gflops may be the multiplication of these ones. Alternatively Gflops can be calculated by the equation (CPU speed in GHz)*(number of CPU cores)*(CPU instruction per cycle)*(number of CPUs per node)*(SIMD register width:here=1). **(reference:https://stackoverflow.com/questions /6289745/how-to-compute-the-theoretical-peak-performance-of-cpu)** Then we plug in the information and after calculation the theoretical peak is about 57.6Gflops.So we can see that our algorithm can only reach 2% to 3% of the theoretical peak performance in terms of Gflops. Note that due to fused multiply-add FMA, the time complexity may come from $2n^3$ to n^3 where the number of flops required in $sum = sum + a * b$ from 2 to 1 and here the hyperthreading is 1.

2 Part 2

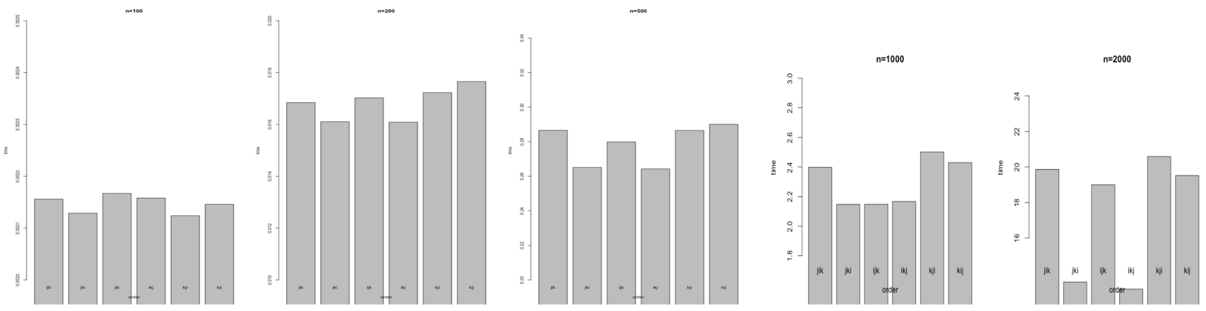
For the implementation of the matrix multiplication, the pipeline is shown as below:

```
for(i=0;i<n_row;i++){
    for(j=0;j<n_row;j++){
        for(k=0;k<n_row;k++){
          sum=sum+matA[i+n_row*k]*matB[k+n_row*j];
      }
      matAB[i+n_row*j]=sum;
      sum=0.0;
    }
  }
```

Here we try 6 possible iterations where in part 1 we use 'ijk', so the 6 possible loop includes jik,jki,ijk,kji,ikj,kij.
Measurements for the naive matrix multiplication program were replicated 10 times for each combination of loop ordering and dimension $n \in 100, 200, 500, 1000, 2000$. For $n = 5000$, only five measurements were made for order 'ijk' and three for all other orders, because these alone took more than eight hours to complete.
Then the performance(total time costed) of 6 possible orders are shown as below:
First is o0 with no optimization:

n(dim)	jik	jki	ijk	kji	ikj	kij
100	0.002156	0.002129	0.002167	0.002158	0.002124	0.002146
200	0.016843	0.016105	0.017024	0.016089	0.017228	0.017653
500	0.286596	0.265140	0.279814	0.264321	0.286510	0.290066
1000	2.39754	2.14734	2.41766	2.16678	2.50141	2.42986
2000	19.86158	13.51869	18.99686	13.12096	20.59163	19.51586

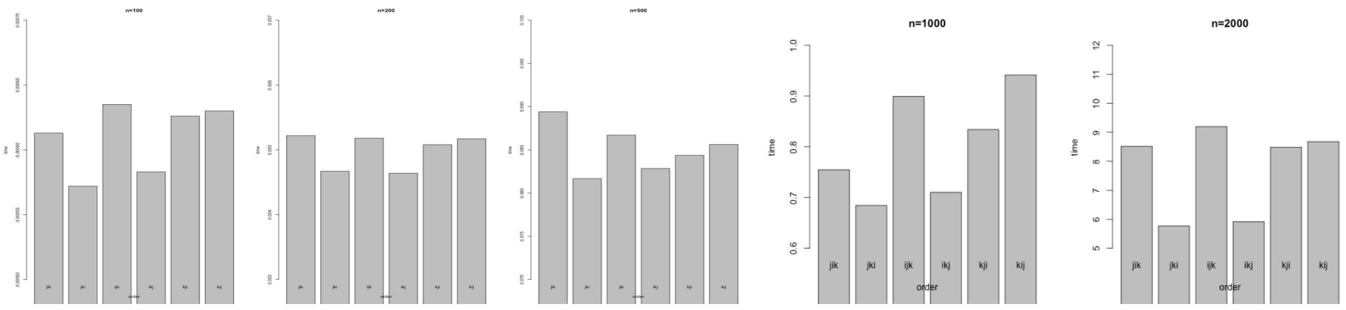
Plots are shown as below:



Then is 02 optimization option:

n(dim)	jik	jki	ijk	kji	ikj	kij
100	0.000613	0.000572	0.000635	0.000583	0.000626	0.000630
200	0.00522	0.00467	0.00518	0.00464	0.00508	0.00517
500	0.08941	0.08168	0.08671	0.08285	0.08437	0.08562
1000	0.75472	0.684433	0.89914	0.71140	0.83417	0.94148
2000	8.51514	5.77148	9.19671	5.91482	8.48174	8.67721

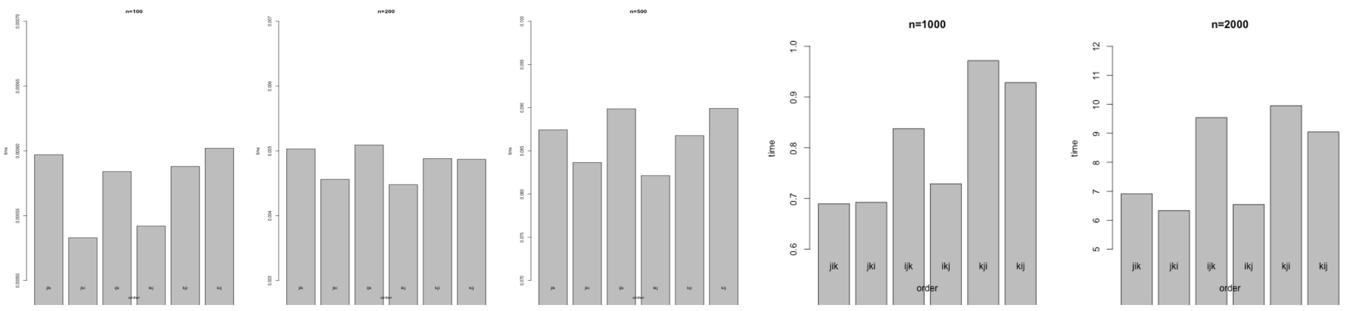
Then the plots are shown as below:



Then is o3 optimization option:

n(dim)	jik	jki	ijk	kji	ikj	kij
100	0.000597	0.000525	0.000584	0.000542	0.000588	0.000602
200	0.00503	0.00456	0.00509	0.00448	0.00488	0.00487
500	0.08743	0.08365	0.08986	0.08214	0.08677	0.08991
1000	0.68955	0.69243	0.83756	0.72893	0.97174	0.92856
2000	6.91476	6.33176	9.54183	6.54114	9.95165	9.04816

Then for o3 optimization choice, the plots are shown as below:



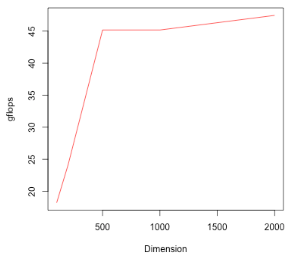
From the table and plot we can see that the order jki and kji are of the optimal ordering. Maybe it because when looping across jki or kji, both of them has row

index i which may potentially help accelerating the program.
Then we draw the plot of the comparison performance:(for visualization we choose not to draw time vs dim or when n=100,200, difference is very very small!)
What's more we can see that comparing with -o, -o2 and -o3 significantly reduce the time complexity for calculating the same equation and we observe that -o3 optimization option is only slightly better than -o2 optimization option.
The nominal peak performance here we reach is calculated by: $\frac{number of floating point iterations}{time*1e+9} = \frac{2*100^3}{0.000525s} = 3.81 Gflops$. So at most we can reach 6.56% of the peak performance theoretically.

3 Part 3

Here we use the dgemm function to evaluate the performance without parallelization:

n(dim)	time	clock cycles
100	0.0001147	414111
200	0.0006704	2410414
500	0.0055564	19959379
1000	0.0442874	159068770
2000	0.3371885	1211079017



We can see that for 1000 by 1000 matrix the Gflop/s is 45.18 and accelerate for about 80 times comparing to the naive implementation and we can see that the dgemm function performs even better for ultra-large matrix and the gflops for the dgemm are increasing for small sized matrix and then saturate after some point.
The time here is determined by $\frac{Diff_{TSC}}{cpu_{speed}}$ as we know the CPU is 3.6GHz so the time=$\frac{clock_c ircle}{cpu_s peed}$ which is approximately follow the above table.
Here the dgemm function works even better for large dataset even without parallelization and it maybe because of the nature of blas library.
What's more, comparing with the naive implementation, the dgemm function has a Gflops of approximately: $\frac{2*1000^3}{0.0442874} = 45.16 Gflop/s$ which has an improvement of almost 80 times. Note that it is also even larger than the theoretically peak performance which can be explained by the fact of the **turbo boost**.