Parallel Computing System

Exercise 3

October 30 2019

s1250250

Phea Sinchhean

Computer Specifications

I used Oracle VirtualBox on my PC.

CPU	Intel(r) Core(tm) i5-6500 CPU @ 3.20GHz × 4
Memory	8GB
Base System	Oracle Solaris 11.4 64-bit
GCC version	9.1.0

Source Code Outline

This code will take matrix sizes and loop order as arguments.
Float and Double precision both are of the same values and will be calculated in the same program.
Matrix initialization of random numbers between 0 and 1.

Input and Output

E.g. 1
Input

```
-----------------------------------------------------------------------------
>gcc -o mmO0 mxm.c -O0
>./mmO0 -o ijk -d “8 16 32 64 128 256 512 1024 2048”
-----------------------------------------------------------------------------
```

This will calculate matrix multiplication with:

- o : loop order in this case ijk (There are currently only ijk, jki, and kij.)
- d : {8, 16, 32, 64, 128, 256, 512, 1024, 2048} as matrix sizes

*The default value for :

- o : ijk
- d : {8, 16}

Output

```
-----------------------------------------------------------------------------
8 3.8147e-06 3.62396e-06
16 3.78132e-05 3.78609e-05
32 0.000146008 0.000147009
64 0.00116062 0.00120807
128 0.0200626 0.0229237
256 0.114716 0.132679
512 1.08174 1.09514
1024 12.7827 16.3217
2048 199.529 208.256
-----------------------------------------------------------------------------
```

Output format :

- 1st row : size of matrix
- 2nd row : elapsed time of matrix multiplication for float
- 3nd row : elapsed time of matrix multiplication for double

E.g 2
Input

```
-----------------------------------------------------------------------------
>gcc -o mmO0 mxm.c -O0
>./mmO0 8 16 32 64 128 256 512 1024 2048
-----------------------------------------------------------------------------
```

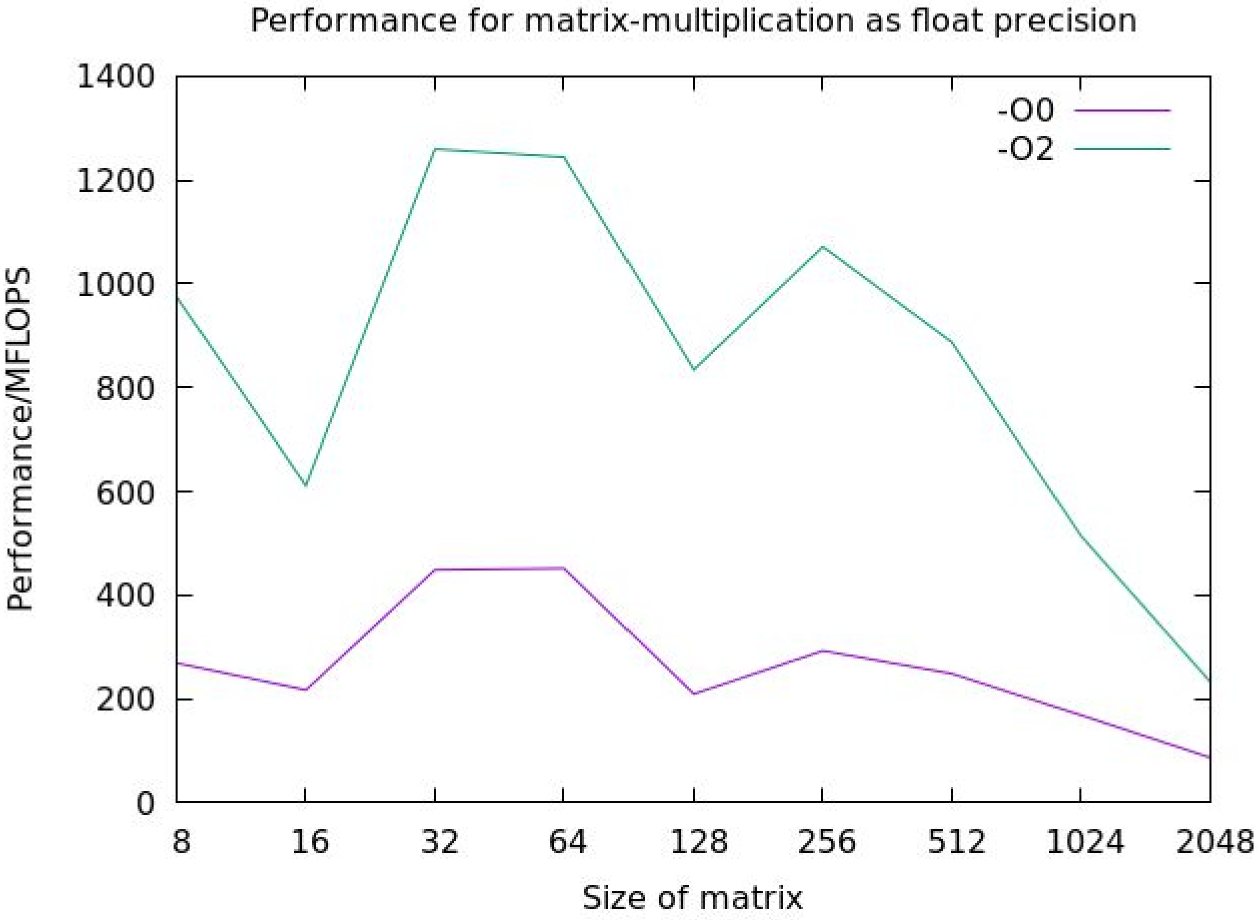
This will give the same result as E.g 1 using default loop option ijk.

***NOTE** : These are the only two input ways that are supported.
Mixing them together may cause error or unwanted results.

Result

Float precision : Comparing optimization option -O0 and -O2 (using ijk)

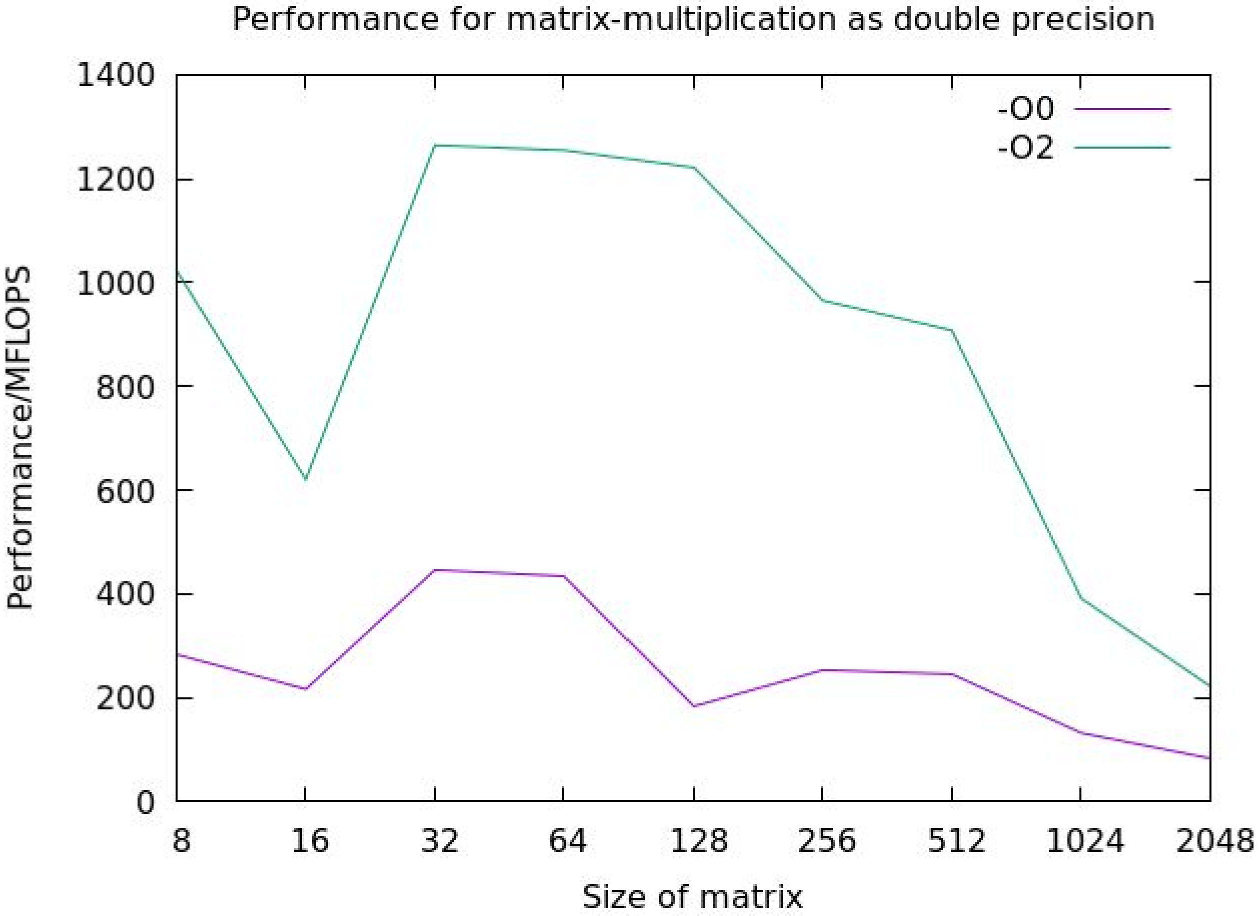
Size	**ElapsedTime of -O0 (s)**	**ElapsedTime of -O2 (s)**
8	3.81E-06	1.05E-06
16	3.78E-05	1.34E-05
32	0.000146008	5.20E-05
64	0.00116062	0.000421143
128	0.0200626	0.00502639
256	0.114716	0.0313088
512	1.08174	0.302542
1024	12.7827	4.17999
2048	199.529	73.9729



Optimization level 2 (-O2) has better performance when compared to no optimization (-O0)

Double precision : Comparing optimization option -O0 and -O2 (using ijk)

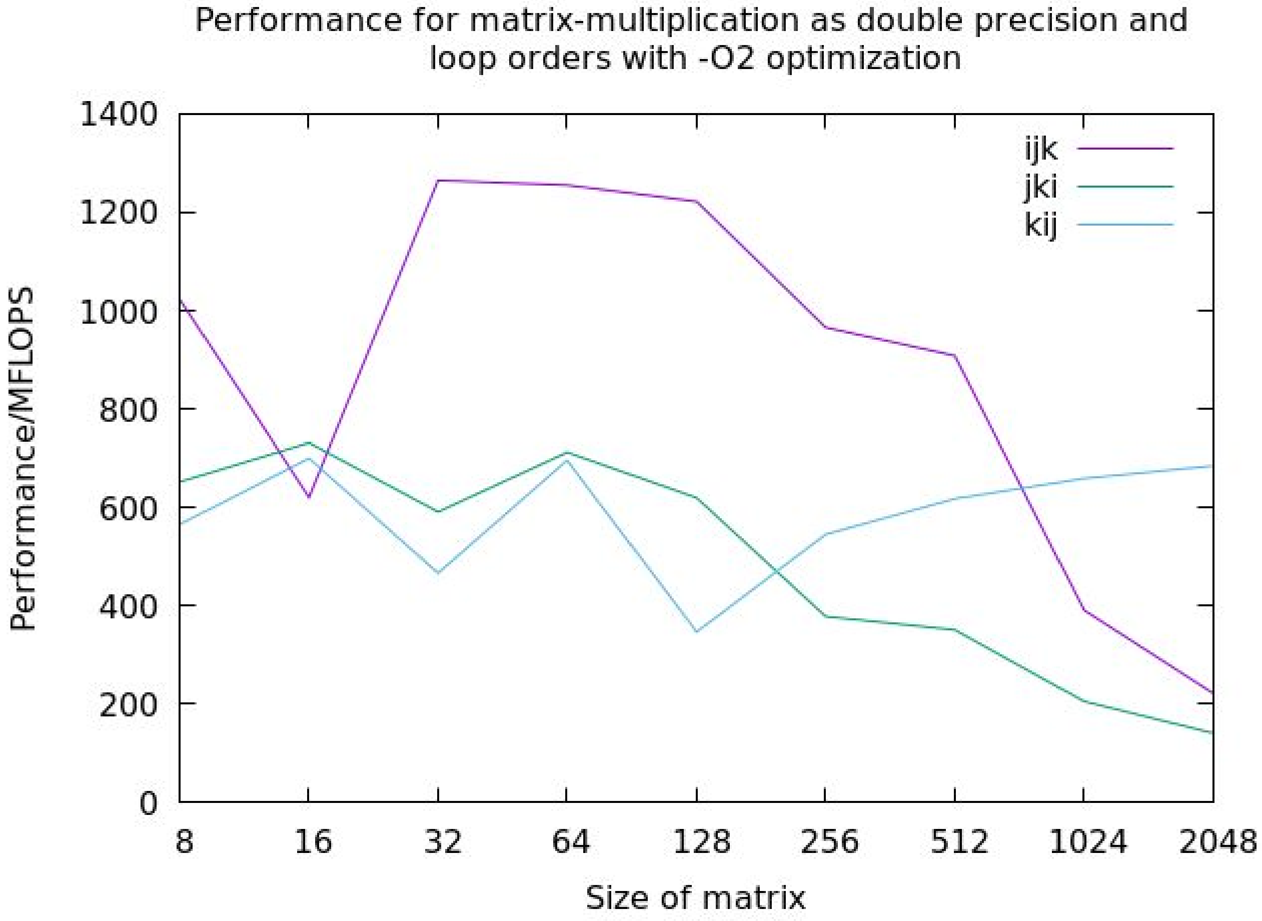
Size	ElapsedTime of -O0 (s)	ElapsedTime of -O2 (s)
8	3.62E-06	1.00E-06
16	3.79E-05	1.32E-05
32	0.000147009	5.18E-05
64	0.00120807	0.000417614
128	0.0229237	0.00343184
256	0.132679	0.0347508
512	1.09514	0.295556
1024	16.3217	5.4952
2048	208.256	77.5877



Optimization level 2 (-O2) has better performance when compared to no optimization (-O0)

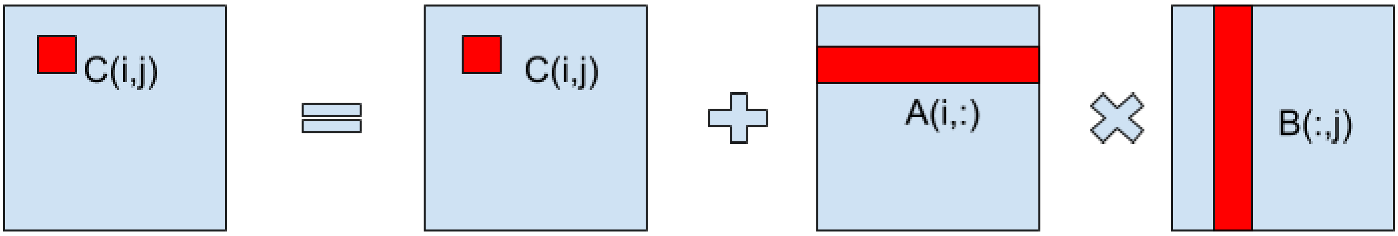
Double precision : Comparing loop orders using same optimization -O2

Size	ijk (s)	jki (s)	kij (s)
8	1.00E-06	1.57E-06	1.81E-06
16	1.32E-05	1.12E-05	1.17E-05
32	5.18E-05	0.000110817	0.000140429
64	0.000417614	0.00073595	0.000753021
128	0.00343184	0.00676618	0.0120928
256	0.0347508	0.0887176	0.0615006
512	0.295556	0.764767	0.434388
1024	5.4952	10.4691	3.25728
2048	77.5877	122.221	25.0791

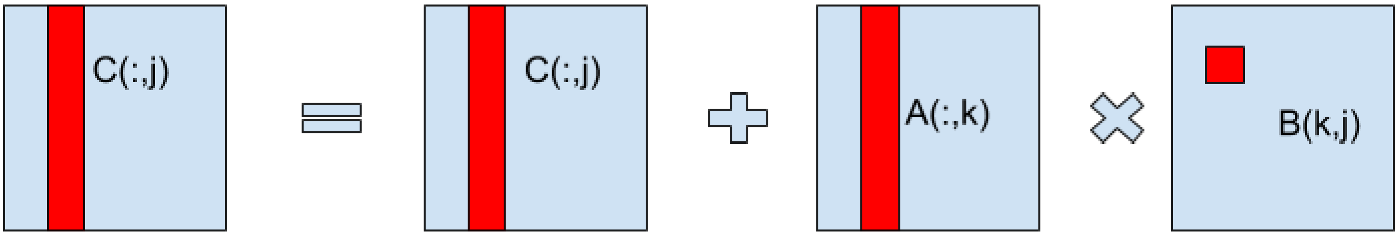


Discussion :

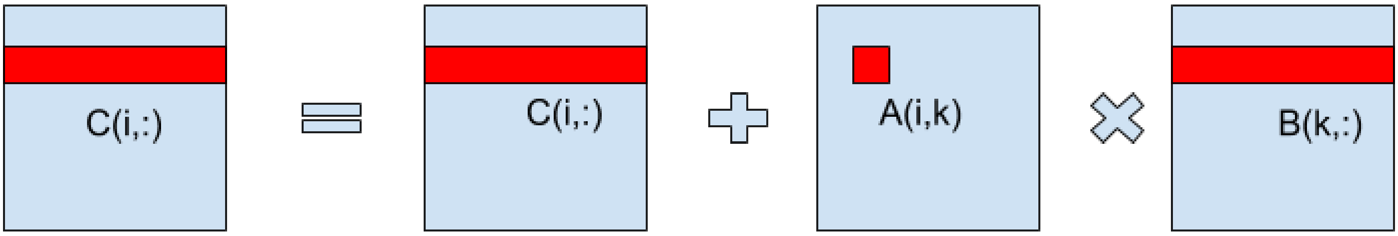
- ijk :
    - A : continuous access
    - B : non-continuous access
    - C : fixed access



- jki :
    - A : non-continuous access
    - B : fixed access
    - C : non-continuous access



- kij :
    - A : fixed access
    - B : continuous access
    - C : continuous access



(We consider when matrix size is big, more than 1000s, for performance evaluation.)

Conclusion:

- ijk : this loop order is faster than jki but slower than kij because it is accessing only one A spatial locality.
- jki : the slowest among the three because it does not use spatial locality at all.
- kij : the fastest loop order because it is accessing 2 spatial localities of B and C.