Sheet2
01. Dec. 2019 Abraham Gutierrez && Huan Chen

Exercise 1:

(1) Running stream.c with g++ compiler ($ g++ stream.c && ./a.out)

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 52484 microseconds.
   (= 52484 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           3225.1     0.050991     0.049611     0.052321
Scale:          3395.6     0.049334     0.047119     0.049904
Add:            4415.3     0.054580     0.054356     0.056243
Triad:          4273.6     0.056270     0.056159     0.056626
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

with gcc compiler ($ gcc stream.c && ./a.out)

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
```

```
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 52304 microseconds.
   (= 52304 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           3494.0     0.047388     0.045793     0.047937
Scale:          3066.2     0.052201     0.052182     0.052234
Add:            4579.3     0.056111     0.052410     0.058745
Triad:          4401.3     0.055742     0.054529     0.056303
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

(2) Running flops.c ($ gcc flops.c && ./a.out) , ps: need to uncomment Timer options (#define UNIX) in the code, not working with g++ compiler

```
 FLOPS C Program (Double Precision), V2.0 18 Dec 1992

  Module     Error        RunTime      MFLOPS
                            (usec)
    1     4.0146e-13      0.0045   3097.4335
    2    -1.4166e-13      0.0011   6379.5285
    3     4.7184e-14      0.0078   2169.8339
    4    -1.2557e-13      0.0060   2512.2398
    5    -1.3800e-13      0.0114   2551.9740
    6     3.2380e-13      0.0113   2559.6152
    7    -8.4583e-11      0.0061   1958.6440
    8     3.4867e-13      0.0118   2553.1240

  Iterations      =  512000000
  NullTime (usec) =     0.0003
  MFLOPS(1)       =  3903.6108
  MFLOPS(2)       =  2319.7016
  MFLOPS(3)       =  2478.9494
  MFLOPS(4)       =  2467.0869
```

Exercise 2.

Define: sizeof(double) = d

	Memory allocated (Bytes)	Floating point operations (FLOPS)	Read/Write from/to memory operations (/)
A, Inner Product	2*N*d	2*N	4*N
B, Matrix Vector	(M*N+N+M)*d	2*M*N	4*N*M+M
C, Matrix Matrix	(M*L+L*N+M*N)*d	2*M*L*N	4*M*L*N+M*N
D, Polynomial function	(p+1)+2*N	3*(p+1)*N	7*N*(p+1)

Exercise 3

See code

Exercise 4

To check computer performance ($ lscpu):

```
------------------------------------------------------------
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           48 bits physical, 48 bits virtual
CPU(s):                  64
On-line CPU(s) list:     0-63
Thread(s) per core:      2
Core(s) per socket:      32
Socket(s):               1
NUMA node(s):            4
Vendor ID:               AuthenticAMD
CPU family:              23
Model:                   1
Model name:              AMD EPYC 7551P 32-Core Processor
Stepping:                2
Frequency boost:         enabled
CPU MHz:                 1725.588
CPU max MHz:             2000.0000
CPU min MHz:             1200.0000
BogoMIPS:                4000.00
Virtualization:          AMD-V
L1d cache:               1 MiB
L1i cache:               2 MiB
L2 cache:                16 MiB
L3 cache:                64 MiB
------------------------------------------------------------
```

As far Parameters selection, the total memory allocated is around 1o times of L3 cache, that is 640MiB. Then increase the number of loops to make running time more than 10 seconds under -Ofast compiler.

A: N = 4*10^7, NLOOPS = 200.
B: M = 4*10^4, N = 2*10^3, NLOOPS = 150.
C: M = 1000, L = 1200, N = 1000, NLOOPS = 5.
D: p +1 = 2*10^5 , N = 5000, NLOOPS = 10.

1. with g++ compiler, -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	29.7115	0.596046	0.501529	8.02446
B. Matrix Vector	44.3464	0.596359	0.504026	8.06543
C. Matrix Matrix	24.8376	0.025332	0.449958	7.20083
D. Polynomial function	36.7284	0.00156462	0.760711	14.1999

2. with g++ compiler, -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	4e-06	0.596046	9313230	1.49012e+08
B. Matrix Vector	13.1059	0.596359	1.70547	27.2909
C. Matrix Matrix	13.6277	0.025332	0.820082	13.124
D. Polynomial function	13.9386	0.00156462	2.00448	37.417

3. with g++ compiler, -Ofast

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	4e-06	0.596046	9313230	1.49012e+08
B. Matrix Vector	13.1048	0.596359	1.70562	27.2933
C. Matrix Matrix	13.9073	0.025332	0.803600	12.8603
D. Polynomial function	14.0033	0.00156462	1.99522	37.2442

4. with clang++ compiler, -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	20.4929	0.596046	0.290855	4.65368
B. Matrix Vector	76.9549	0.596359	0.290453	4.64782
C. Matrix Matrix	62.1692	0.025332	0.179765	2.87685
D. Polynomial function	50.2429	0.00156462	0.556093	10.3804

5. with clang++ compiler, -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	5e-06	0.596046	7450580	1.19209e+08
B. Matrix Vector	13.052	0.596359	1.71252	27.4037
C. Matrix Matrix	12.9185	0.025332	0.865105	13.8446
D. Polynomial function	13.3644	0.00156462	2.0906	39.0246

5. with clang++ compiler, -Ofast

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	4e-06	0.596046	9313230	1.49012e+08
B. Matrix Vector	13.0455	0.596359	1.71337	27.4174
C. Matrix Matrix	13.3283	0.025332	0.838508	13.4189
D. Polynomial function	13.3559	0.00156462	2.09193	39.0494

The inner product performance has strong deviations under -O3 compiler, we think case2 is reasonable:

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
g++, -O3, case1	4e-06	0.596046	9313230	1.49012e+08
g++, -O3, case2	9.14284	0.596046	1.62982	26.0771

Exercise 5

	Memory allocated (Bytes)	Floating point operations (FLOPS)	Read/Write from/to memory operations (/)
(a) Inner Product	2*N*d	2*N	4*N
(a) L2 Norm	2*N*d	2*N	3*N
(b) Kanhan Summation	2*N*d	4*N	12*N
(c) Non-transposed	(M*L+L*N+M*N)*d	2*M*L*N	4*M*L*N+M*N
(d) Transposed	(M*L+2*L*N+M*N)*d	2*M*L*N	4*M*L*N+M*N+2*L*N

ps: We guess for computation a[i]*a[i], the compiler only read memory once.

(a) L2 Norm performance

Using the same parameters with (A), that is A: N = 4*10^7, NLOOPS = 500. Only test under -O1 compiler.

With compiler g++ , -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Inner Product	29.7115	0.596046	0.501529	8.02446
L2 Norm	29.5949	0.298023	0.503505	6.04206

With compiler clang++ , -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Inner Product	51.2392	0.596046	0.290855	4.65368
L2 Norm	51.0187	0.290823	0.292072	3.50487

According to observation, no big difference, is this correct? Is there a way to call a funtion in C++ library directly?

(b) Kahan summation performance

N = 4*10^7, NLOOPS = 50, test under -O3 compiler.

With compiler

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
g++ , -O1	13.3809	0.596046	0.556805	13.3633
g++ , -O3	13.3754	0.596046	0.557035	13.3689
g++ , -Ofast	0.795784	0.596046	9.36257	224.702
clang++ , -O1	18.9254	0.596046	0.393681	9.44834
clang++ , -O3	13.6021	0.596046	0.547752	13.146
clang++ , -Ofast	0.766399	0.596046	9.72154	233.317

Also compared Kahan Summation with normal summation (setting data type as ‘float’):
Normal summation result is:1.64473
Kahan summation result is:1.64493
KahanSum - NormalSum = 0.000208735
pi^2/6 - NormalSum = 0.000208744
pi^2/6 - KahanSum = 8.65883e-09

What we noticed is that under -Ofast compiler, the calculation speed is faster, but calculation result of Kahan summation is not that accurate anymore compared with other compilers:
Normal summation result is:1.64473
Kahan summation result is:1.64482
The difference: KahanSum - NormalSum = 9.84669e-05
The difference: pi^2/6 - NormalSum = 0.000208744
The difference: pi^2/6 - KahanSum = 0.000110277

(c) Row-wise and column-wise comparison

Compare A*B and A*BT under -O1 compiler.
Non-transposed: M = 1000, L = 1200, N = 1000, NLOOPS = 5.
Transposed: M = 1000, L = 1200, N = 1000, NLOOPS = 5.

With compiler g++ , -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Non-Transposed	24.8376	0.025332	0.449958	7.20083
Transposed	22.1983	0.0342727	0.503457	8.06101

With compiler g++ , -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Non-Transposed	13.6277	0.025332	0.820082	13.124
Transposed	6.76604	0.0342727	1.65176	26.4469

With compiler clang++ , -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Non-Transposed	62.1692	0.025332	0.179765	2.87685
Transposed	38.7052	0.0342727	0.288743	4.62317

With compiler clang++ , -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
Non-Transposed	12.9185	0.025332	0.865105	13.8446
Transposed	6.66337	0.0342727	1.67721	26.8544

Under -O3, and clang++ compiler, the transposed case is obviously faster, but since there are more memory allocated and more memory read/write operations, we have no clue is this comparison fair or not.

Exercise 6

For comparison, use the same dimensions with Exercise 3, that is:
A: N = 4*10^7, NLOOPS = 200.
B: M = 4*10^4, N = 2*10^3, NLOOPS = 150.
C: M = 1000, L = 1200, N = 1000, NLOOPS = 5.
With compiler g++, -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	8.4938	0.596046	1.75436	28.0697
B. Matrix Vector	9.15332	0.596359	2.44193	39.0757
C. Matrix Matrix	3.47743	0.025332	3.21383	51.432

With compiler g++, -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	8.47243	0.596046	1.75878	28.1405
B. Matrix Vector	9.11957	0.596359	2.45096	39.2203
C. Matrix Matrix	3.50099	0.025332	3.1922	51.0858

With compiler clang++, -O1

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	8.49942	0.596046	1.7532	28.0512
B. Matrix Vector	9.10317	0.596359	2.45538	39.291
C. Matrix Matrix	3.48613	0.025332	3.20581	51.3037

With compiler clang++, -O3

	Total running time: sec	Allocated memory: GiB	Double precision performance: GFLOPS	Memory bandwidth: GiB/s
A. Inner Product	8.47739	0.596046	1.75775	28.124
B. Matrix Vector	9.11071	0.596359	2.45335	39.2585
C. Matrix Matrix	3.47966	0.025332	3.21178	51.3991

It’s obviously faster compared with out own created function, but advanced compiler, e.g. -O3, doesn’t speed up the codes.

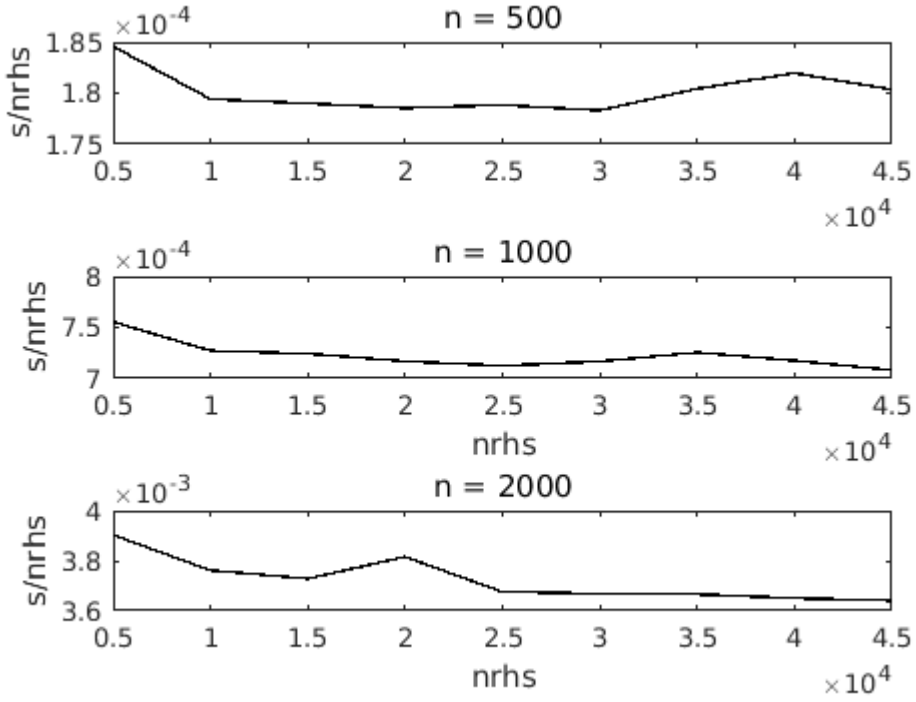
Exercise 7

######## When n is 500 ######
sec./nrhs = 0.00018457 nrhs = 5000
sec./nrhs = 0.000179416 nrhs = 10000
sec./nrhs = 0.000178995 nrhs = 15000
sec./nrhs = 0.000178534 nrhs = 20000
sec./nrhs = 0.000178833 nrhs = 25000
sec./nrhs = 0.000178304 nrhs = 30000
sec./nrhs = 0.000180433 nrhs = 35000
sec./nrhs = 0.000181959 nrhs = 40000
sec./nrhs = 0.000180355 nrhs = 45000

######## When n is 1000 ######
sec./nrhs = 0.000755382 nrhs = 5000
sec./nrhs = 0.000727206 nrhs = 10000

sec./nrhs = 0.000724233 nrhs = 15000
sec./nrhs = 0.000716836 nrhs = 20000
sec./nrhs = 0.000712453 nrhs = 25000
sec./nrhs = 0.000716719 nrhs = 30000
sec./nrhs = 0.000725516 nrhs = 35000
sec./nrhs = 0.000717464 nrhs = 40000
sec./nrhs = 0.000708236 nrhs = 45000

######## When n is 2000 ######
sec./nrhs = 0.00390285 nrhs = 5000
sec./nrhs = 0.00376145 nrhs = 10000
sec./nrhs = 0.00372805 nrhs = 15000
sec./nrhs = 0.0038153 nrhs = 20000
sec./nrhs = 0.0036754 nrhs = 25000
sec./nrhs = 0.00366862 nrhs = 30000
sec./nrhs = 0.00366573 nrhs = 35000
sec./nrhs = 0.0036502 nrhs = 40000
sec./nrhs = 0.00364051 nrhs = 45000



Exercise 8

Make run, get the following result:

```
Intervalls: 100 x 100
Start Jacobi solver for 10201 d.o.f.s
aver. Jacobi rate :  0.997922  (1000 iter)
final error: 0.124971 (rel)   0.000194029 (abs)
JacobiSolve: timing in sec. : 0.080958
ASCI file  square_100.txt  opened
17361  2  34320  3

 Start Jacobi solver for 17361 d.o.f.s
aver. Jacobi rate :  0.998401  (1000 iter)
final error: 0.201744 (rel)   0.000265133 (abs)
JacobiSolve: timing in sec. : 0.189638
```