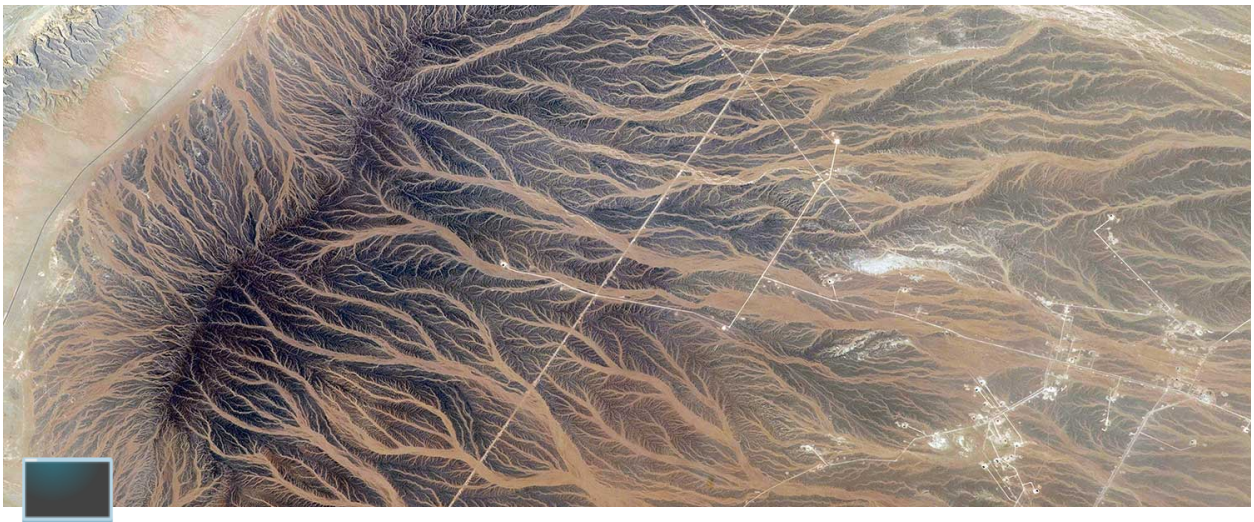


SPP Assignment

Know Your Computer

Part 1

Feature	Specification
CPU	i5-1035G1
Generation	10th
Frequency Range	1-3.6 GHz
Number of Cores	4
Hyper-Threading Availability	8 (2 threads per socket)
SIMD ISA	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2, Intel® AVX-512
Cache size	6MB Smart Cache
Main Memory Bandwidth	58.3 GB/s
L1d cache	192 KiB
L1i cache	128 KiB
L2 cache	2 MiB
L3 cache	6MiB

Part 2

Theoretically

Flops per core = sockets * cycles per second * Flops per cycle

= 1 * 3.6 * 10^9 * 16

= 59.6 GFlops/sec

Flops per processor = sockets * cores * cycles per second * Flops per cycle

= 59.6 * 4 GFlops/sec

= 238.4 Gflops/sec

Assuming FMA instructions the flops per cycle would be 32

Flops per core = 119.2 GFlops/ sec

Flops per processor = 476.8 GFlops/sec

Whetstone benchmark program

Using ICC compiler

for loop count 800000 the results were 80000 MIPS (Million Instruction Per Second)

for loop count 900000 the results were 67500 MIPS (Million Instruction Per Second)

for loop count 1000000 the results were 50000 MIPS (Million Instruction Per Second)

Part 3

1. Main memory size : 8GB
2. max main memory size: 64GB
3. DDR4
4. Max Memory Bandwidth: 58.3 GB/s
5. Steam Benchmark

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           7126.4     0.023677     0.022452     0.025509
Scale:          6496.9     0.025265     0.024627     0.025973
Add:           10748.3     0.023637     0.022329     0.026630
Triad:          9795.9     0.026468     0.024500     0.029467
-------------------------------------------------------------
```

6. Own Benchmark

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
direct add bw: 14.628705
parallel add bw: 26.098587
loop unrolling add time 41.137451
```

Part 4

1. Secondary structure storage size: 512 GB SSD
2. Disk write speed: 977 MB/s
3. Disk read speed : 1.3 GB/s

Cluster

- ADA Peak FLOPs: 70.66 TFLOPS
- Abacus Peak FLOPs: 14 TFLOPS

Blas

▼ Level 1

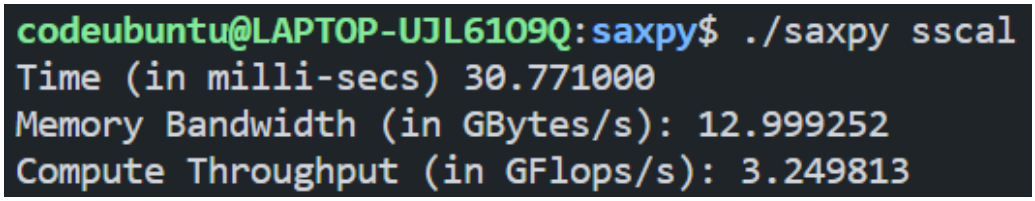
SSCAL

Function: X = alpha X

O.I = (N)/(4*N) = 0.25

Using gcc -O3 = on an average 2.8 GFlops/sec was observed with the maximum ranging to 3.11

Using ICC -O3 = on an average 2.8 GFlops/sec with the maximum up to 3.17



Best execution time

Baseline Time of execution = 244.588000 milliseconds

Best Time of execution = 30.77 milliseconds

Baseline GFlops/sec = 0.4

Best GFlops/sec = 3.24

Speedup = 3.24 / 0.4 = 8.1

Memory Bandwidth achieved = 12.99 GBtyes/s

Techniques Used for optimisation

1. Used Vectorization

   ```
   #pragma omp simd
   ```

   It gives slightly better results.

2. Used Parallelization

   There wasn't any significant increase by using this, probably the compiler with flag -O3 is smart to identify vectorization and parallelization. Also using the compiler with flag -O0 gives 0.5 GFlops/sec on an average thus it shows that the -O3 flag is doing significant optimization

The Program is Memory Bound

DSCAL (double precision)

Function: X = alpha X

O.I = (N)/(8*N) = 0.125

Using gcc -O3 = on an average 1.45 GFlops/sec was observed with the maximum ranging to 1.58

Using ICC -O3 = on an average 1.4 GFlops/sec with the maximum up to 1.53

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy dscal
Time (in milli-secs) 63.143000
Memory Bandwidth (in GBytes/s): 12.669655
Compute Throughput (in GFlops/s): 1.583707
```

Baseline Time of execution = 240.446000 milliseconds

Best Time of execution = 63.14 milliseconds

Baseline GFlops/sec = 0.41

Best GFlops/sec = 1.58

Speedup = 1.58 / 0.41 = 3.85

Memory Bandwidth achieved = 12.66 GBtyes/s

Techniques Used for optimisation

The compiler with flag -O3 was already doing optimization so results with vectorization and parallelization didn't have any change.

The Program is Memory Bound

SAXPY

Function: Y = alpha X + Y

O.I = (2N)/(8*N) = 0.25

Using gcc -O3 = on an average 3.75 GFlops/sec was observed with the maximum ranging to 3.81

Using ICC -O3 = on an average 2.5 GFlops/sec with the maximum up to 2.8

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy saxpy
Time (in milli-secs) 49.493000
Memory Bandwidth (in GBytes/s): 16.163902
Compute Throughput (in GFlops/s): 4.040975
```

best execution time

Baseline Time of execution = 311.996000 milliseconds

Best Time of execution = 51.53 milliseconds

Baseline GFlops/sec = 0.64

Best GFlops/sec = 3.88

Speedup = 3.88 / 0.64 = 6.0625

Memory Bandwidth achieved = 15.523129 GBtyes/s

Techniques Used for optimisation

1. Used Vectorization

   ```
   #pragma omp simd
   ```

   It gives slightly better results.

2. Used Parallelization

   ```
   #pragma omp parallel for
   ```

3. Using both vectorization and parallelization

   ```
   #pragma omp parallel
       {
       #pragma omp for simd
           for (int i = 0; i < N; ++i)
           {
               Y[i*incY] = alpha * X[i*incX] + Y[i*incY];
           }
       }
   ```

   This part gave the best result. Results obtained with gcc -O3 were very close to the results obtained using this code, this implies that gcc with -O3 was optimizing quite well whereas results with the -O0 flag were around 0.6-0.7 GFlops/sec.

The Program is Memory Bound

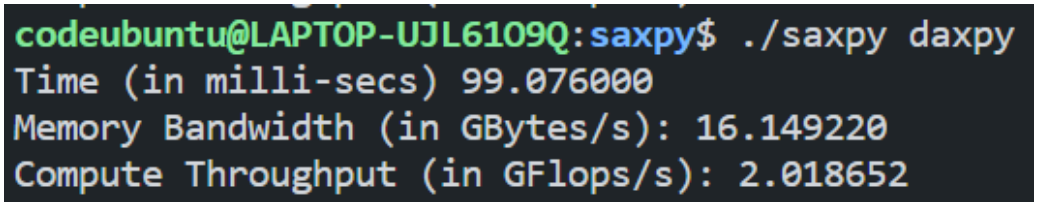
DAXPY (Double Precision)

Function: Y = alpha X + Y

O.I = (2N)/(16*N) = 0.125

Using gcc -O3 = on an average 1.85 GFlops/sec was observed with the maximum ranging to 1.98

Using ICC -O3 = on an average 1.90 GFlops/sec with the maximum up to 1.97



best execution time

Baseline Time of execution =633.045000 milliseconds

Best Time of execution = 99.07 milliseconds

Baseline GFlops/sec = 0.31

Best GFlops/sec = 2.01

Speedup = 2.01 / 0.31 = 6.0625

Memory Bandwidth achieved = 16.149220 GBtyes/s

Techniques Used for optimisation

1. Used Vectorization

   ```
   #pragma omp simd
   ```

   It gives slightly better results.

2. Used Parallelization

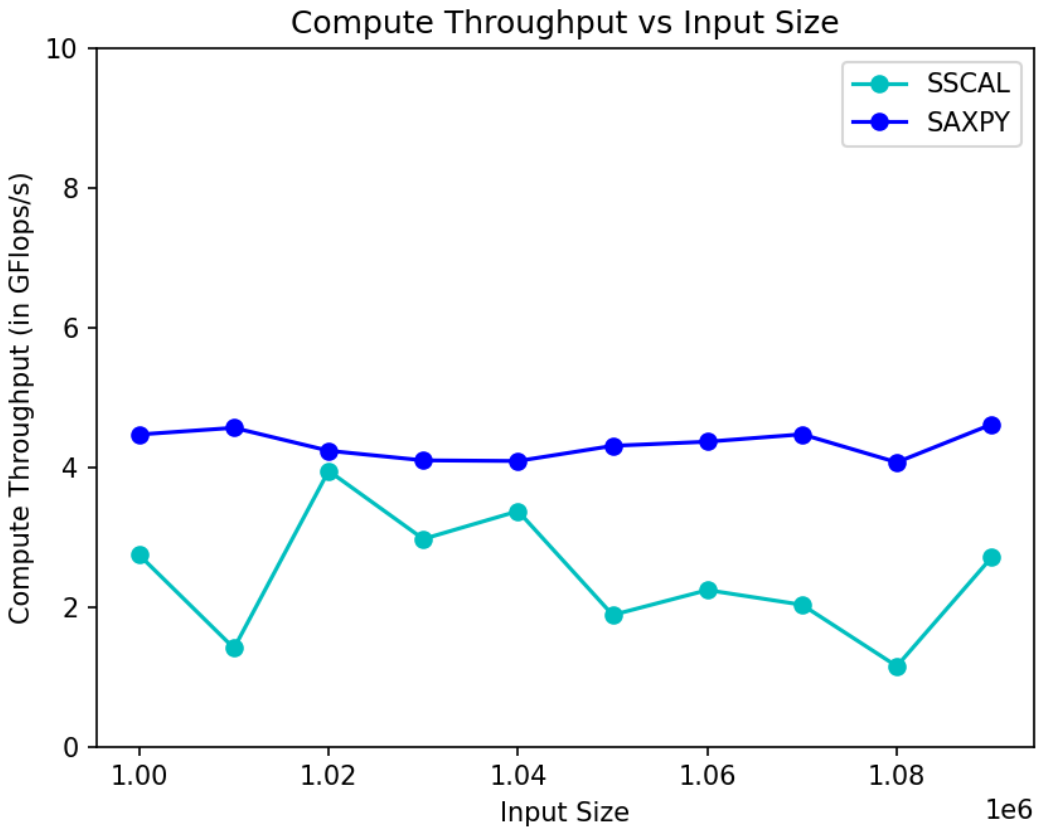
   ```
   #pragma omp parallel for
   ```

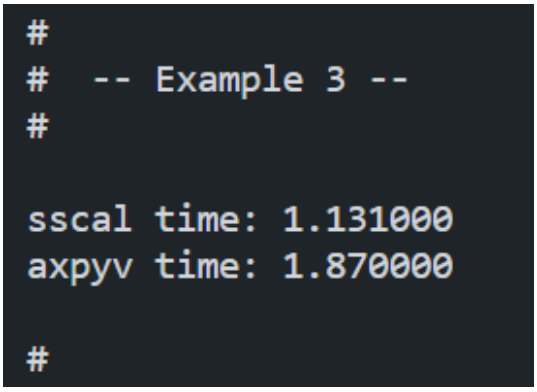
3. Using both vectorization and parallelization

   ```
   #pragma omp parallel
       {
       #pragma omp for simd
           for (int i = 0; i < N; ++i)
   ```

```
        {
            Y[i*incY] = alpha * X[i*incX] + Y[i*incY];
        }
    }
```

This part gave the best result. Results obtained with gcc -O3 were very close to the results obtained using this code, this implies that gcc with -O3 was optimizing quite well whereas results with the -O0 flag were around 0.3-0.4 GFlops/sec.





BLIS LEVEL 1 RESULTS for n=1e6

Observation

Using double precision the compute throughput is half of what that was obtained using the single precision.

The Program is Memory Bound because { O.I * Memory Bandwidth << Theoretically Observed Peak Flops }

▼ Level 2

SGEMY

Function Y = αAX + βY A is in row major form

O.I = 2(n+1)m / 4(mn+n+m)

where A is matrix of size m*n

X is a vector of size n

Y is a vector of size m.

Used sdot for this program, that program is optimised using vectorization

▼ For input n=1e5 and m=1e3

Using gcc -O0 the average GFlops/sec were 0.75

Using gcc -O3 the average was around 1.7

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemy
Time (in milli-secs) 117.301000
Memory Bandwidth (in GBytes/s): 3.413475
Compute Throughput (in GFlops/s): 1.705032
```

Using icc -O3 the average was around 6.4

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemy
Time (in milli-secs) 31.019000
Memory Bandwidth (in GBytes/s): 12.908346
Compute Throughput (in GFlops/s): 6.447726
```

Techniques

1. Using parallelization

   The average flops was around 12

```
#pragma omp parallel for simd
            for (int i = 0; i < M; i++)
            {
                float AX = cblas_sdot(N,A+i*lda,1,X,incX);
                Y[i*incY] = (beta * Y[i*incY]) + (alpha * AX);
            }
```

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemy
Time (in milli-secs) 15.802000
Memory Bandwidth (in GBytes/s): 25.338818
Compute Throughput (in GFlops/s): 12.656752
```

best execution time

2. Using Vectorization

   It was already implemented in cblas_sdot()

Baseline Time of execution =363.931000 milliseconds

Best Time of execution = 15.80 milliseconds

Baseline GFlops/sec = 0.550104

Best GFlops/sec = 12.65

Speedup = 12.65 / 0.55 = 23

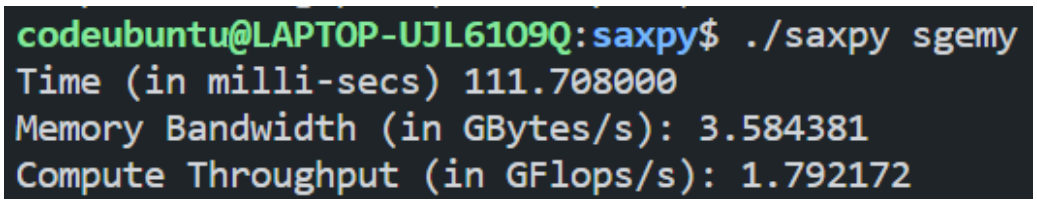
Memory Bandwidth achieved = 25.3388 GBtyes/s

The program is memory bound

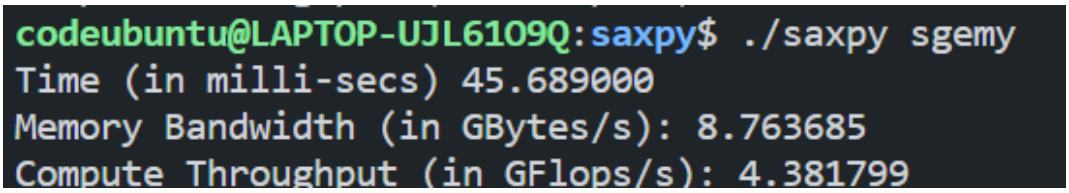
▼ For input n=1e3 and m=1e5

Using gcc -O0 the average GFlops/sec were 0.75

Using gcc -O3 the average was around 1.75



Using icc -O3 the average was around 4.38



Techniques

1. Using parallelization

   The average flops was around 11.5

```
#pragma omp parallel for simd
            for (int i = 0; i < M; i++)
            {
                float AX = cblas_sdot(N,A+i*lda,1,X,incX);
                Y[i*incY] = (beta * Y[i*incY]) + (alpha * AX);
            }
```

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemy
Time (in milli-secs) 16.437000
Memory Bandwidth (in GBytes/s): 24.359921
Compute Throughput (in GFlops/s): 12.179838
```

best execution time

2. Using Vectorization

   It was already implemented in cblas_sdot()

Baseline Time of execution =363.931000 milliseconds

Best Time of execution = 16.47 milliseconds

Baseline GFlops/sec = 0.550104

Best GFlops/sec = 12.17

Speedup = 12.17 / 0.55 = 22.12

Memory Bandwidth achieved = 24.35 GBtyes/s

```
#
#  -- GEMV tweeked--
#

calctime: 56.212000
```

Blis output for level 2

SGEMYT

Y = αA^T X + βY A is in row major form

O.I = 2(m+1)n / 4(mn+n+m)

where A is matrix of size m*n

this implies A transpose will be of size n*m

X is a vector of size m

Y is a vector of size n.

▼ For Input n=1e5 and m=1e3

For checking the code for correctness and debugging used valgrind

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ valgrind ./saxpy sgemyt
==2449== Memcheck, a memory error detector
==2449== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2449== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2449== Command: ./saxpy sgemyt
==2449==
==2449== Warning: set address range perms: large range [0x4e4b040, 0x1cbc3440) (undefined)
Time (in milli-secs) 5190.118000
Memory Bandwidth (in GBytes/s): 0.077147
Compute Throughput (in GFlops/s): 0.038573
59.836460
58.697601
56.926380
57.972511
60.077621
60.629887
58.071831
58.084938
57.106281
57.371365
==2449== Warning: set address range perms: large range [0x4e4b028, 0x1cbc3458) (noaccess)
==2449==
==2449== HEAP SUMMARY:
==2449==     in use at exit: 0 bytes in 0 blocks
==2449==   total heap usage: 4 allocs, 4 frees, 400,405,024 bytes allocated
==2449==
==2449== All heap blocks were freed -- no leaks are possible
==2449==
==2449== For lists of detected and suppressed errors, rerun with: -s
==2449== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Using gcc -O0 the average GFlops/sec were 0.5

Using gcc -O3 the average was around 1.2

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 164.927000
Memory Bandwidth (in GBytes/s): 2.427765
Compute Throughput (in GFlops/s): 1.213870
```

Using icc -O3 the answer was ranging from 1.1 to 1.85 with average nearly 1.3

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 150.640000
Memory Bandwidth (in GBytes/s): 2.658019
Compute Throughput (in GFlops/s): 1.328996
```

Techniques

1. Using vectorization of inner loop

   There was any significant increase in the GFlops/sec because of high stride resulting in cache miss

2. Using Parallelization on the outer loop

   There was increase in the GFlop/sec

   The average GFlops were 5.6 with the highest ranging upto 5.94

```
codeubuntu@LAPTOP-UJL61O9Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 33.671000
Memory Bandwidth (in GBytes/s): 11.891658
Compute Throughput (in GFlops/s): 5.945769
```

3. Using Axpy program

   Using the axpy program we can increase GFlops, because of high cache hit.

```
codeubuntu@LAPTOP-UJL61O9Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 28.102000
Memory Bandwidth (in GBytes/s): 14.248239
Compute Throughput (in GFlops/s): 7.124048
```

Best Execution Time

Baseline Time of execution =568.333000 milliseconds

Best Time of execution = 14.24 milliseconds

Baseline GFlops/sec = 0.352258

Best GFlops/sec = 7.12

Speedup = 7.12 / 0.35 = 20.34

Memory Bandwidth achieved = 14.24 GBtyes/s

The program is memory bound

▼ For input n=1e3 and m=1e5

Using gcc -O0 the average GFlops/sec were 0.17

Using gcc -O3 the average was around 0.21

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 958.252000
Memory Bandwidth (in GBytes/s): 0.417848
Compute Throughput (in GFlops/s): 0.208715
```

Using icc -O3 the answer was nearly the same as gcc -O3 , i.e. around 0.176

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 1132.045000
Memory Bandwidth (in GBytes/s): 0.353700
Compute Throughput (in GFlops/s): 0.176673
```

Techniques

1. Using vectorization of inner loop

   There was any significant increase in the GFlops/sec because of high stride resulting in cache miss

2. Using Parallelization on the outer loop

```
#pragma omp parallel for simd
            for (int j = 0; j < N; j++)
            {
                float AX = 0;
                #pragma omp simd
                for (int i = 0; i < M; i++)
                {
                    AX += A[i*lda+j] * X[i*incX];
                }
                Y[j*incY] = (beta * Y[j]) + (alpha * AX);
            }
```

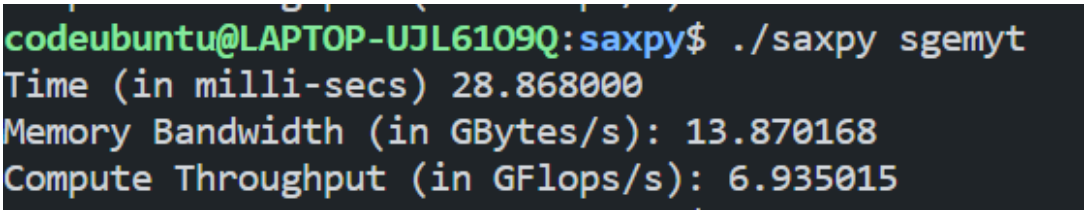
```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyt
Time (in milli-secs) 154.788000
Memory Bandwidth (in GBytes/s): 2.586790
Compute Throughput (in GFlops/s): 1.292103
```

There was increase in the GFlop/sec

The average GFlops were 1.2 with the range 0.8 to 1.7

3. Using Axpy

   Using the axpy program we can increase GFlops, because of high cache hit.



Baseline Time of execution =1262.482000 milliseconds

Best Time of execution = 28.83 milliseconds

Baseline GFlops/sec = 0.15

Best GFlops/sec = 6.93

Speedup = 6.93 / 0.15 = 46.2

Memory Bandwidth achieved = 13.87 GBtyes/s

The program is memory bound

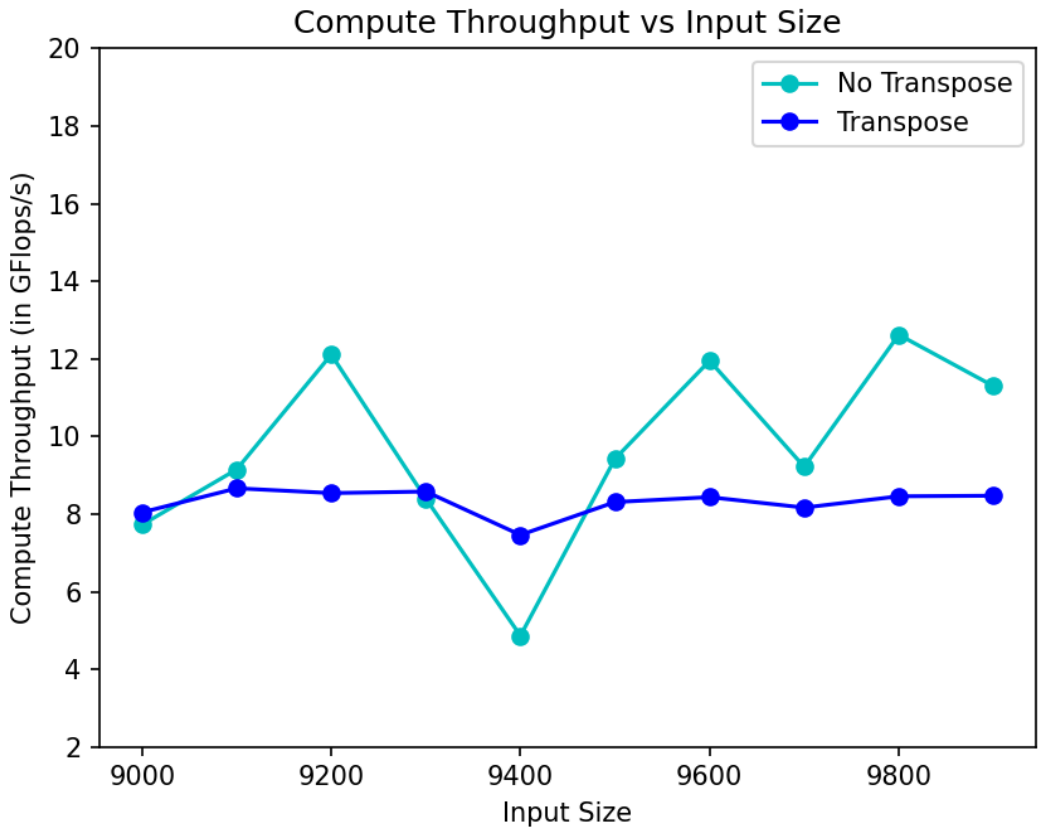
▼ Data Prefetching

Data Prefetching using ***__builtin_prefetch***

Data prefetch, or cache management, instructions allow a compiler or an assembly language programmer to minimize cache-miss latency by moving data into a cache before it it accessed. Data prefetch instructions are generally treated as hints; they affect the performance but not the functionality of software in which they are used.

Most computers do this task automatically. Data prefetching can be done by two ways hardware and software.

Since the compliers do this automatically there wasn't any increase in the GFlops. In one or two cases it did show a better result but on an average the results were comparable to the the earlier optimised one.



Input size is number of rows, for the purpose of plotting a graph the matrix is considered to be a square matrix

Observations For Row Major

- SAXPY V/S DOT
  - To compute problems of level 2 we can use code written for level 1.
  - There are 2 code that can do ours job i.e SAXPY and Dot Product
  - Using saxpy is better when we want to compute A transpose and Dot product is better when we want to compute for normal matrix A
  - For using saxpy we first need to use sscal to get Beta Y then use saxpy
  - Advantage of using saxpy in A transpose is that the stride access for dot product is lda number of coulmns, this means that cache miss is very high

whereas in saxpy stride access is 1 so cache hit will be high

- In case of normal A, dot product outperforms saxpy beacuse now stride access for dot product is 1 whereas it is lda for saxpy
- Moreover we would be able to parallelize the for loop (in which the dot product is called) but it isn't possible for saxpy

- RACE CONDITION
  - It occurs when 2 or more threads are trying to write at the same location, this is happening when we try to parallelize the for loop (in which saxpy is called) because they all try to write at the same location, this will decrease the accuracy and might as well increase the time.
- M V/s N

  To generalise it the code favours those values where it can access more values together

Column Major

C means that the matrix A is stored is column major fashion

Derivation to find the matrix

Let the matrix A be of the order m*n.

Matrix-Vector multiplication is defined to be

$y_j = \sum_{i=0}^{N-1} a_{ji} x_i$

Since the Matrix A is stored in column major form then:

a_{ij} = A[i*m + j]

SGEMYC

O.I = 2(n+1)m / 4(mn+n+m)

where A is matrix of size m*n

X is a vector of size n

Y is a vector of size m.

▼ For Input n=1e5 and m=1e3

Using gcc -O0 the average GFlops/sec were 0.14

Using gcc -O3 the average was around 0.20

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 961.406000
Memory Bandwidth (in GBytes/s): 0.416478
Compute Throughput (in GFlops/s): 0.208031
```

Using icc -O3 the answer was ranging from 0.17

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 1163.036000
Memory Bandwidth (in GBytes/s): 0.344275
Compute Throughput (in GFlops/s): 0.171965
```

Techniques

1. Using vectorization of inner loop

   There was any significant increase in the GFlops/sec because of high stride resulting in cache miss

   ```
   codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
   Time (in milli-secs) 1150.067000
   Memory Bandwidth (in GBytes/s): 0.348157
   Compute Throughput (in GFlops/s): 0.173905
   ```

2. Using Parallelization on the outer loop

   There was increase in the GFlop/sec

   The average GFlops were 1.5 with the highest ranging upto 1.69 on time

   ```
   codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
   Time (in milli-secs) 118.060000
   Memory Bandwidth (in GBytes/s): 3.391530
   Compute Throughput (in GFlops/s): 1.694071
   ```

3. Using Axpy

   Using the axpy program we can increase GFlops, because of high cache hit.

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 23.046000
Memory Bandwidth (in GBytes/s): 17.374121
Compute Throughput (in GFlops/s): 8.678382
```

Best Execution Time

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 30.680000
Memory Bandwidth (in GBytes/s): 13.050978
Compute Throughput (in GFlops/s): 6.525424
```

with parallelization decreases

Baseline Time of execution =1487.592000 milliseconds

Best Time of execution = 23.04 milliseconds

Baseline GFlops/sec = 0.13

Best GFlops/sec = 8.67

Speedup = 8.67 / 0.13 = 66

Memory Bandwidth achieved = 13.05 GBtyes/s

The program is memory bound

▼ For input n=1e3 and m=1e5

Using gcc -O0 the average GFlops/sec were 0.51

Using gcc -O3 the average was around 1.1

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 151.747000
Memory Bandwidth (in GBytes/s): 2.638629
Compute Throughput (in GFlops/s): 1.319301
```

Using icc -O3 the answer was nearly the same as gcc -O3 , i.e. around 1.05

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 168.050000
Memory Bandwidth (in GBytes/s): 2.382648
Compute Throughput (in GFlops/s): 1.191312
```

Techniques

1. Using vectorization of inner loop

   There was any significant increase in the GFlops/sec because of high stride resulting in cache miss

2. Using Parallelization on the outer loop

```
#pragma omp parallel for simd
            for(int j=0;j<M;j++)
            {
                float AX = 0;
                #pragma omp simd
                for(int i=0;i<N;i++)
                {
                    // __builtin_prefetch(A+(i+1)*lda+j);
                    AX += A[i*lda+j] * X[i*incX];
                }
                // AX += A[(N-1)*lda+j] * X[(N-1)*incX];
                Y[j*incY] = (beta * Y[j*incY]) + (alpha * AX);
            }
```

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 37.464000
Memory Bandwidth (in GBytes/s): 10.687700
Compute Throughput (in GFlops/s): 5.343797
```

   There was increase in the GFlop/sec

   The average GFlops were 1.2 with the range 0.8 to 1.7

3. Using Axpy

   Using the axpy program we can increase GFlops, because of high cache hit.

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyc
Time (in milli-secs) 27.468000
Memory Bandwidth (in GBytes/s): 14.577108
Compute Throughput (in GFlops/s): 7.288481
```

Baseline Time of execution = 384.585000 milliseconds

Best Time of execution = 27.46 milliseconds

Baseline GFlops/sec = 0.52

Best GFlops/sec = 7.28

Speedup = 7.28 / 0.52 = 14

Memory Bandwidth achieved = 14.57 GBtyes/s

The program is memory bound

SGEMYCT

Y = αA^T X + βY A is in column major form

O.I = 2(m+1)n / 4(mn+n+m)

where A is matrix of size m*n

this implies A transpose will be of size n*m

X is a vector of size m

Y is a vector of size n.

Used sdot for this program, that program is optimised using vectorization

▼ For input n=1e5 and m=1e3

Using gcc -O0 the average GFlops/sec were 0.75

Using gcc -O3 the average was around 1.7

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyct
Time (in milli-secs) 109.965000
Memory Bandwidth (in GBytes/s): 3.641195
Compute Throughput (in GFlops/s): 1.820579
```

Using icc -O3 the average was around 6.4

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyct
Time (in milli-secs) 30.442000
Memory Bandwidth (in GBytes/s): 13.153012
Compute Throughput (in GFlops/s): 6.576440
```

Techniques

1. Using parallelizatio

   The average flops was around 11.5

   ```
   #pragma omp parallel for simd
               for (int i=0;i<N;i++)
               {
                   float AX = cblas_sdot(M,A+i*lda,1,X,incX);
                   Y[i*incY] = (beta * Y[i*incY]) + (alpha * AX);
               }
   ```

   ```
   codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemyct
   Time (in milli-secs) 16.977000
   Memory Bandwidth (in GBytes/s): 23.585085
   Compute Throughput (in GFlops/s): 11.792425
   ```

2. Using Vectorization

   It was already implemented in cblas_sdot()

Baseline Time of execution =269.247000 milliseconds

Best Time of execution = 16.97 milliseconds

Baseline GFlops/sec = 0.743555

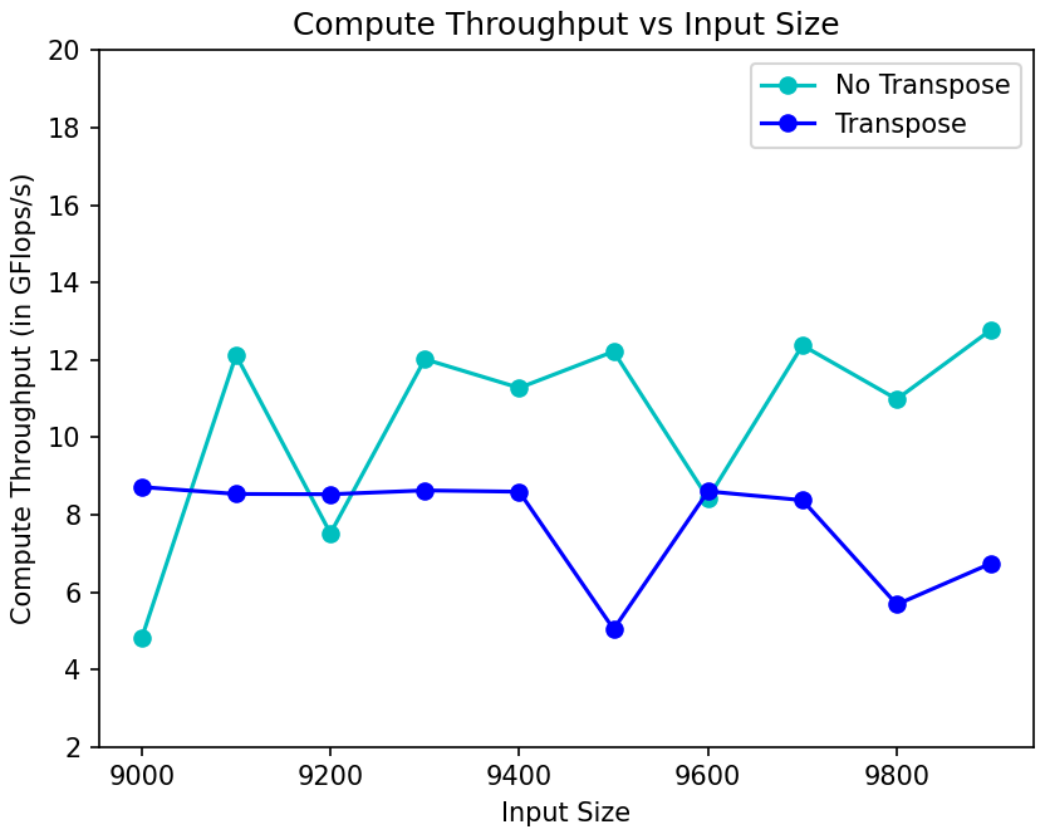
Best GFlops/sec = 11.79

Speedup = 11.79 / 0.74 = 15.93

Memory Bandwidth achieved = 23.585 GBtyes/s

The program is memory bound

The results in this part is exactly similar to the results for row major where matrix A is not a transpose matrix



Input size is number of rows, for the purpose of plotting a graph the matrix is considered to be a square matrix

Observations

Since the elements are stored to opposite way in column major the results obtained are completely opposite.

A generalised result can be:

Row major no transpose = Coulmn Major Transpose

and,

Row major Transpose = Coulmn Major No Transpose

The Program is Memory Bound because { O.I * Memory Bandwidth << Theoretically Observed Peak Flops }

▼ Level 3

Operational Intensity = $\frac{2n^2(n+1)}{12n^2}$ **=** $\frac{n+1}{6}$

Reasoning:

For a general case let us assume that all the 3 matrices are of form n*n, i.e they are square matrices. Then the number of operations would be equal to n times the number of operations in Blas level 2.

Execution Time = Number of operations / GFlops/sec obtained

Code	Meaning
sgemm	Row major
sgemmbt	Row Major with B Transpose
sgemmat	Row Major with A transpose
sgemmatbt	Row Major with A and B transpose
sgemmc	Column Major
sgemmcbt	Column Major with B transpose
sgemmcat	Column Major with A transpose
sgemmcatbt	Column Major with A and B transpose

Code	GCC -O0	GCC -O3	ICC -O3
sgemm	0.65	9.1	2.7
sgemmbt	0.75	1.9	2.6
sgemmat	0.66	9.2	2.7
sgemmatbt	0.73	1.7	2.8

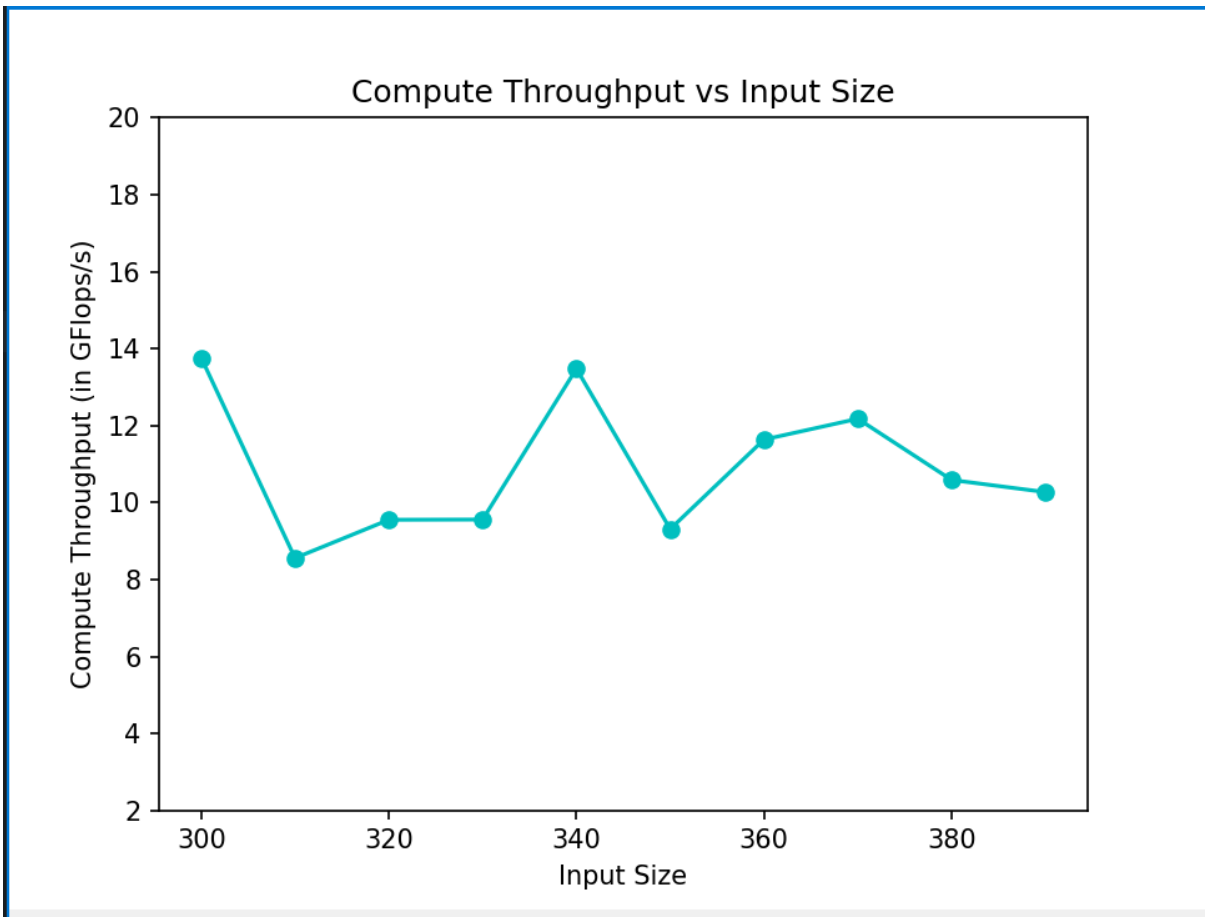
Code	GCC -O0	GCC -O3	ICC -O3
sgemmc	0.70	1.7	2.6
sgemmcbt	0.62	9.3	2.5
sgemmcat	0.73	1.8	2.9
sgemmcatbt	0.65	9	2.8

Code	Baseline Execution Time	Best Execution Time	Memory Bandwidth	Speed Up
sgemm	181	4.51	0.28	40.13303769
sgemmbt	158	7.22	0.19	21.88365651
sgemmat	182	5.67	0.26	32.09876543
sgemmatbt	165	4.21	0.27	39.19239905
sgemmc	214	6.46	0.17	33.12693498
sgemmcbt	173	6.65	0.18	26.01503759
sgemmcat	159	3.76	0.30	42.28723404
sgemmcatbt	176	6.18	0.21	28.4789644

SGEMM

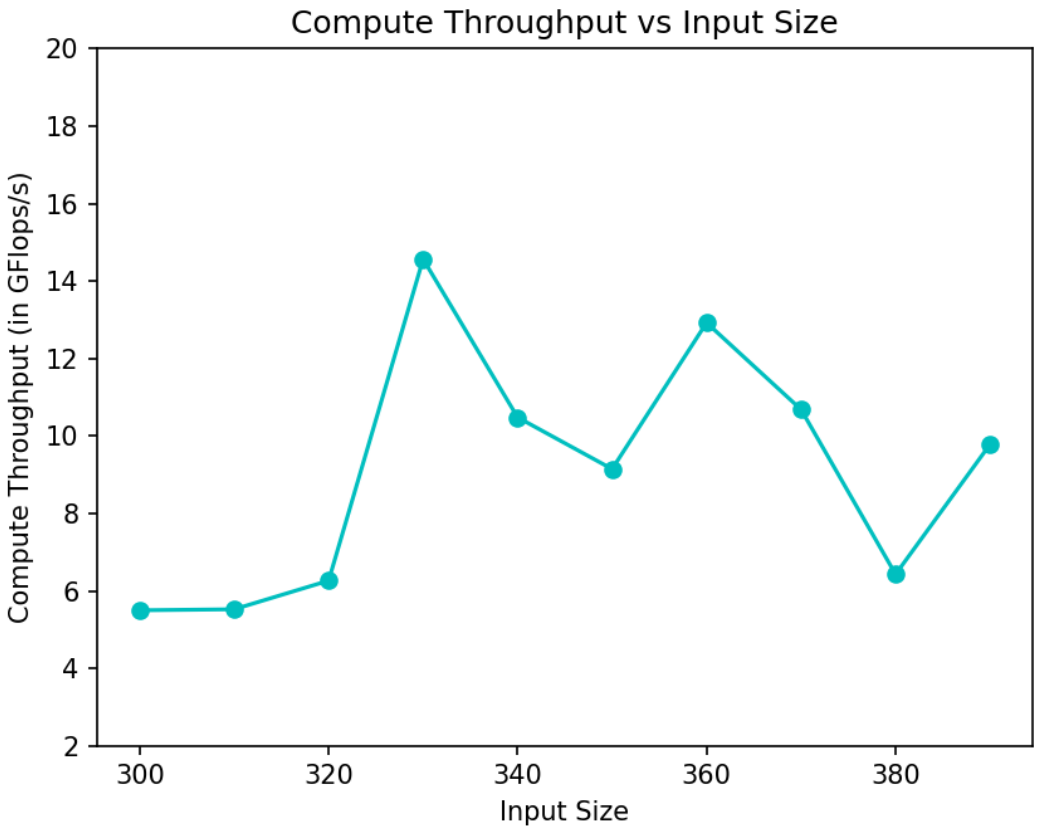
The best execution time was reached by using gcc -O3

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemm
Time (in milli-secs) 6.681000
Memory Bandwidth (in GBytes/s): 0.161652
Compute Throughput (in GFlops/s): 8.109564
Time (in milli-secs) 6.896000
Memory Bandwidth (in GBytes/s): 0.167227
Compute Throughput (in GFlops/s): 8.667952
Time (in milli-secs) 5.542000
Memory Bandwidth (in GBytes/s): 0.221725
Compute Throughput (in GFlops/s): 11.862288
Time (in milli-secs) 8.742000
Memory Bandwidth (in GBytes/s): 0.149485
Compute Throughput (in GFlops/s): 8.246603
Time (in milli-secs) 7.877000
Memory Bandwidth (in GBytes/s): 0.176108
Compute Throughput (in GFlops/s): 10.008785
Time (in milli-secs) 7.224000
Memory Bandwidth (in GBytes/s): 0.203488
Compute Throughput (in GFlops/s): 11.904070
Time (in milli-secs) 9.397000
Memory Bandwidth (in GBytes/s): 0.165500
Compute Throughput (in GFlops/s): 9.957561
Time (in milli-secs) 12.434000
Memory Bandwidth (in GBytes/s): 0.132122
Compute Throughput (in GFlops/s): 8.169519
Time (in milli-secs) 11.214000
Memory Bandwidth (in GBytes/s): 0.154521
Compute Throughput (in GFlops/s): 9.812092
Time (in milli-secs) 12.232000
Memory Bandwidth (in GBytes/s): 0.149215
Compute Throughput (in GFlops/s): 9.723855
```

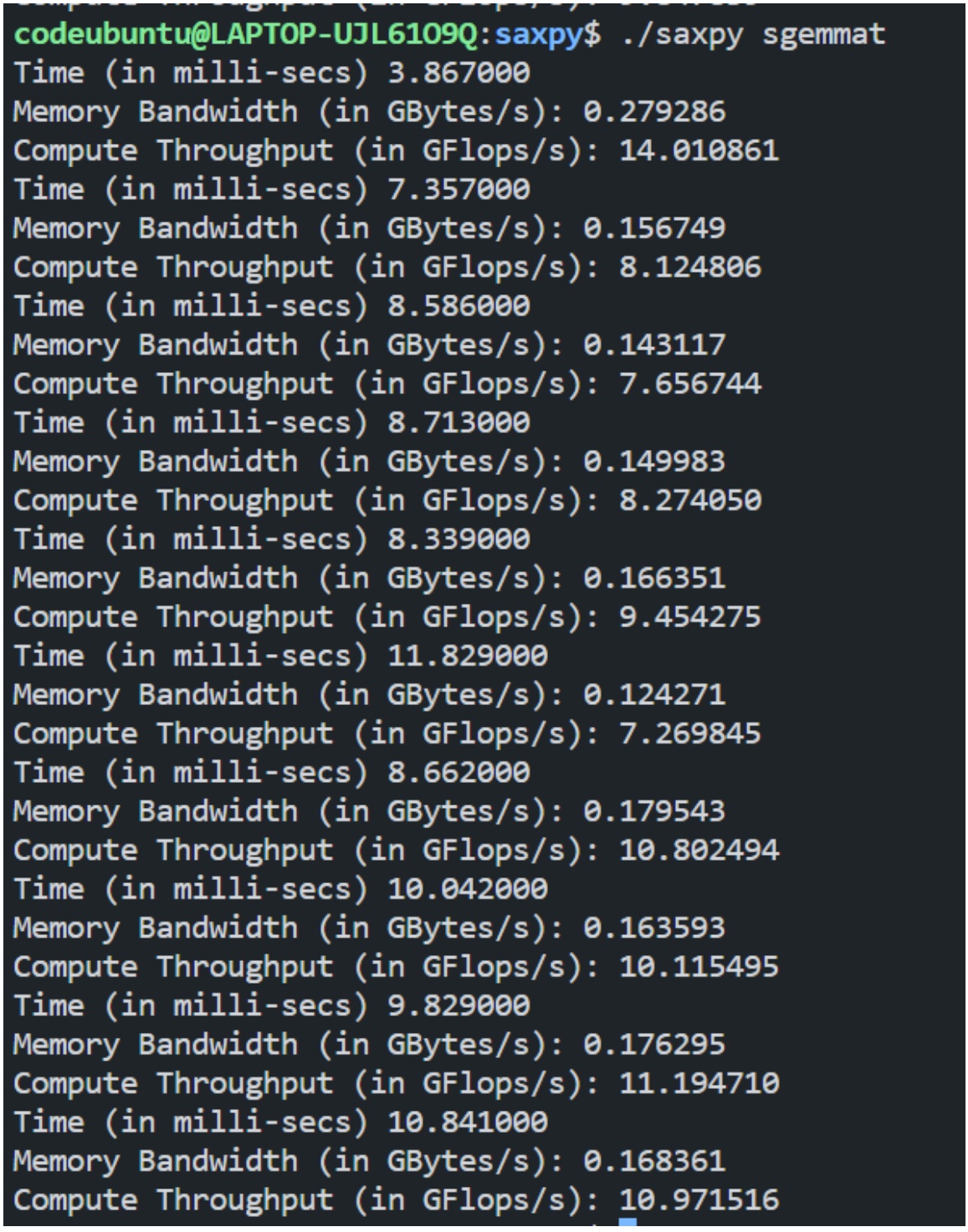

Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380
Input Size

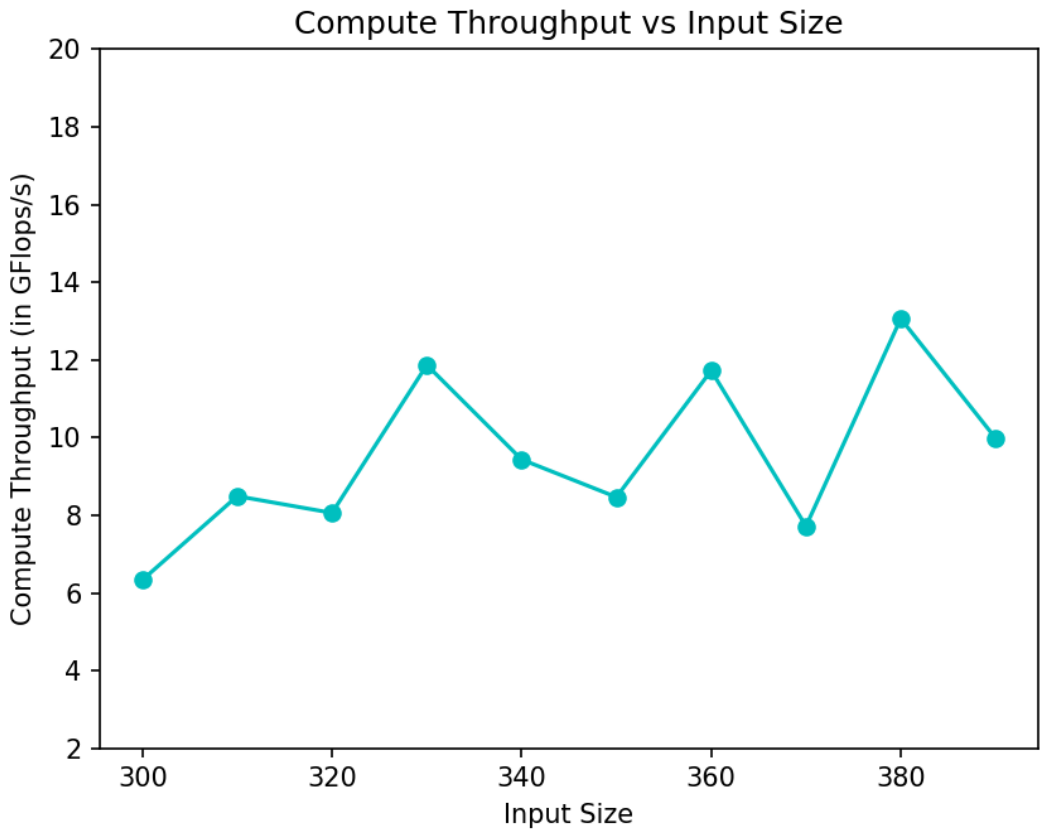
SGEMMBT

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmbt
Time (in milli-secs) 15.503000
Memory Bandwidth (in GBytes/s): 0.069664
Compute Throughput (in GFlops/s): 3.494807
Time (in milli-secs) 3.964000
Memory Bandwidth (in GBytes/s): 0.290918
Compute Throughput (in GFlops/s): 15.079263
Time (in milli-secs) 15.505000
Memory Bandwidth (in GBytes/s): 0.079252
Compute Throughput (in GFlops/s): 4.239974
Time (in milli-secs) 13.702000
Memory Bandwidth (in GBytes/s): 0.095373
Compute Throughput (in GFlops/s): 5.261407
Time (in milli-secs) 17.284000
Memory Bandwidth (in GBytes/s): 0.080259
Compute Throughput (in GFlops/s): 4.561398
Time (in milli-secs) 8.877000
Memory Bandwidth (in GBytes/s): 0.165596
Compute Throughput (in GFlops/s): 9.687394
Time (in milli-secs) 16.479000
Memory Bandwidth (in GBytes/s): 0.094375
Compute Throughput (in GFlops/s): 5.678209
Time (in milli-secs) 12.445000
Memory Bandwidth (in GBytes/s): 0.132005
Compute Throughput (in GFlops/s): 8.162298
Time (in milli-secs) 15.128000
Memory Bandwidth (in GBytes/s): 0.114543
Compute Throughput (in GFlops/s): 7.273453
Time (in milli-secs) 17.178000
Memory Bandwidth (in GBytes/s): 0.106252
Compute Throughput (in GFlops/s): 6.924101
```


Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
Input Size
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380

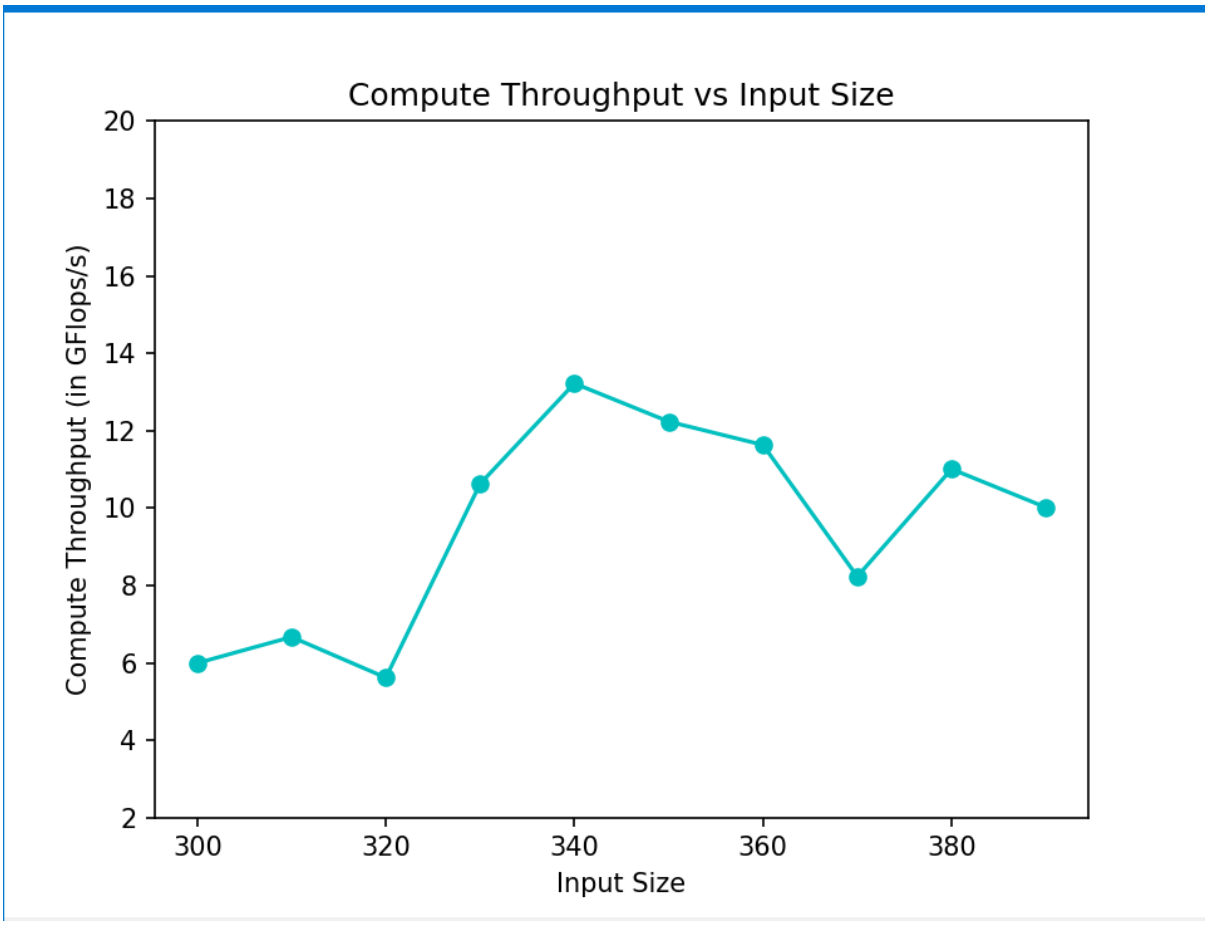
SGEMMAT


codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmat
Time (in milli-secs) 3.867000
Memory Bandwidth (in GBytes/s): 0.279286
Compute Throughput (in GFlops/s): 14.010861
Time (in milli-secs) 7.357000
Memory Bandwidth (in GBytes/s): 0.156749
Compute Throughput (in GFlops/s): 8.124806
Time (in milli-secs) 8.586000
Memory Bandwidth (in GBytes/s): 0.143117
Compute Throughput (in GFlops/s): 7.656744
Time (in milli-secs) 8.713000
Memory Bandwidth (in GBytes/s): 0.149983
Compute Throughput (in GFlops/s): 8.274050
Time (in milli-secs) 8.339000
Memory Bandwidth (in GBytes/s): 0.166351
Compute Throughput (in GFlops/s): 9.454275
Time (in milli-secs) 11.829000
Memory Bandwidth (in GBytes/s): 0.124271
Compute Throughput (in GFlops/s): 7.269845
Time (in milli-secs) 8.662000
Memory Bandwidth (in GBytes/s): 0.179543
Compute Throughput (in GFlops/s): 10.802494
Time (in milli-secs) 10.042000
Memory Bandwidth (in GBytes/s): 0.163593
Compute Throughput (in GFlops/s): 10.115495
Time (in milli-secs) 9.829000
Memory Bandwidth (in GBytes/s): 0.176295
Compute Throughput (in GFlops/s): 11.194710
Time (in milli-secs) 10.841000
Memory Bandwidth (in GBytes/s): 0.168361
Compute Throughput (in GFlops/s): 10.971516


Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
Input Size
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380

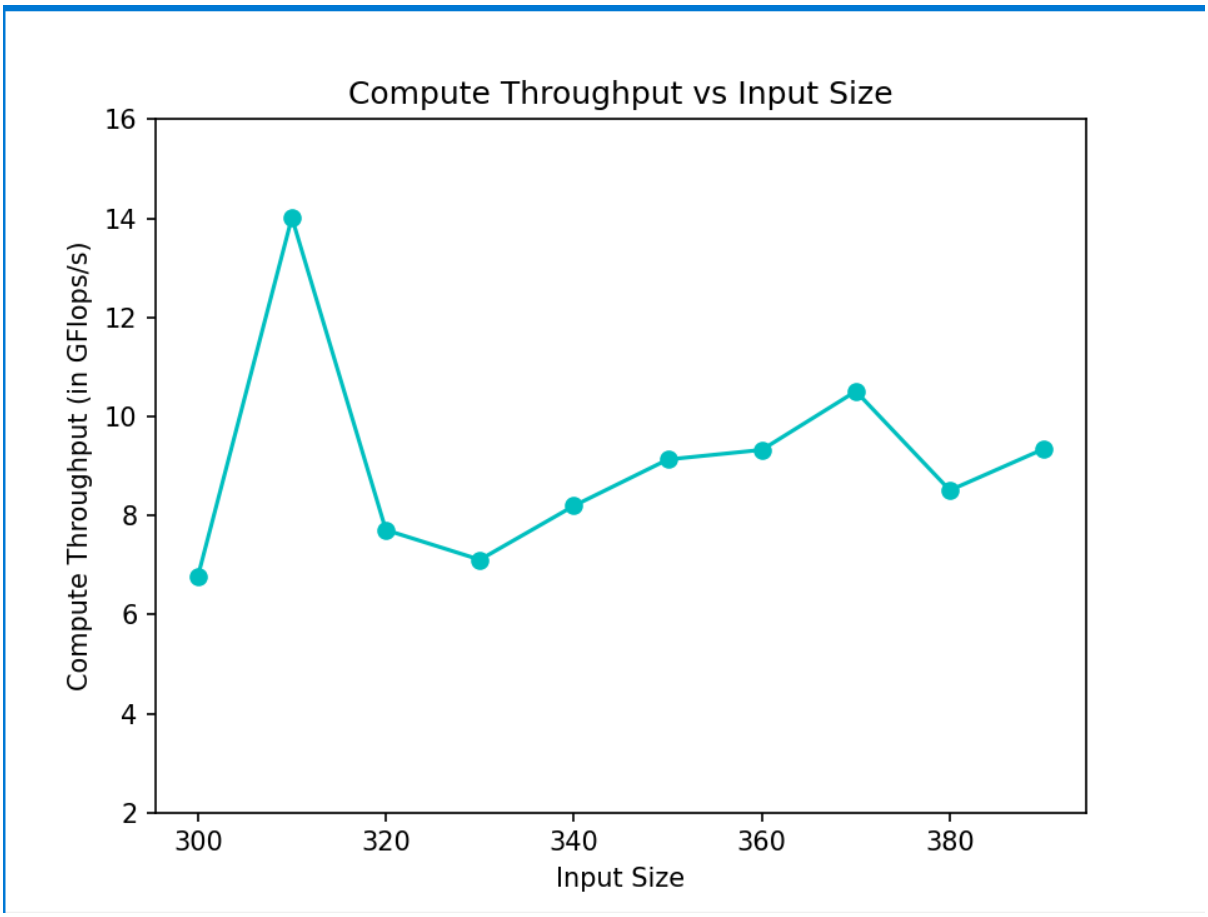
SGEMMATBT

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmatbt
Time (in milli-secs) 7.640000
Memory Bandwidth (in GBytes/s): 0.141361
Compute Throughput (in GFlops/s): 7.091623
Time (in milli-secs) 5.701000
Memory Bandwidth (in GBytes/s): 0.202280
Compute Throughput (in GFlops/s): 10.484862
Time (in milli-secs) 7.668000
Memory Bandwidth (in GBytes/s): 0.160250
Compute Throughput (in GFlops/s): 8.573396
Time (in milli-secs) 9.044000
Memory Bandwidth (in GBytes/s): 0.144494
Compute Throughput (in GFlops/s): 7.971230
Time (in milli-secs) 8.485000
Memory Bandwidth (in GBytes/s): 0.163489
Compute Throughput (in GFlops/s): 9.291597
Time (in milli-secs) 7.619000
Memory Bandwidth (in GBytes/s): 0.192939
Compute Throughput (in GFlops/s): 11.286914
Time (in milli-secs) 12.216000
Memory Bandwidth (in GBytes/s): 0.127308
Compute Throughput (in GFlops/s): 7.659725
Time (in milli-secs) 10.772000
Memory Bandwidth (in GBytes/s): 0.152507
Compute Throughput (in GFlops/s): 9.429985
Time (in milli-secs) 15.755000
Memory Bandwidth (in GBytes/s): 0.109984
Compute Throughput (in GFlops/s): 6.983992
Time (in milli-secs) 14.836000
Memory Bandwidth (in GBytes/s): 0.123025
Compute Throughput (in GFlops/s): 8.017134
```

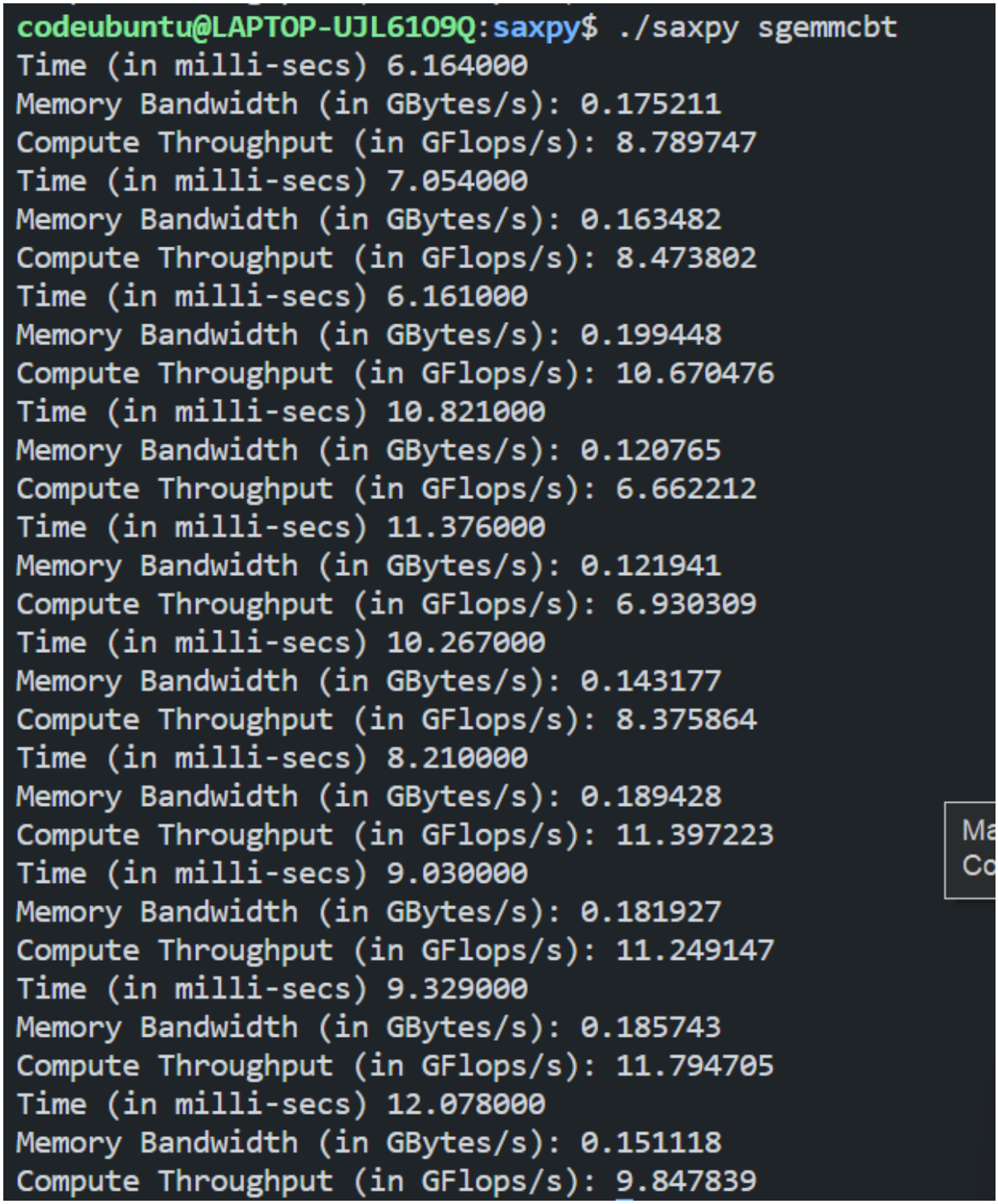

Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380
Input Size

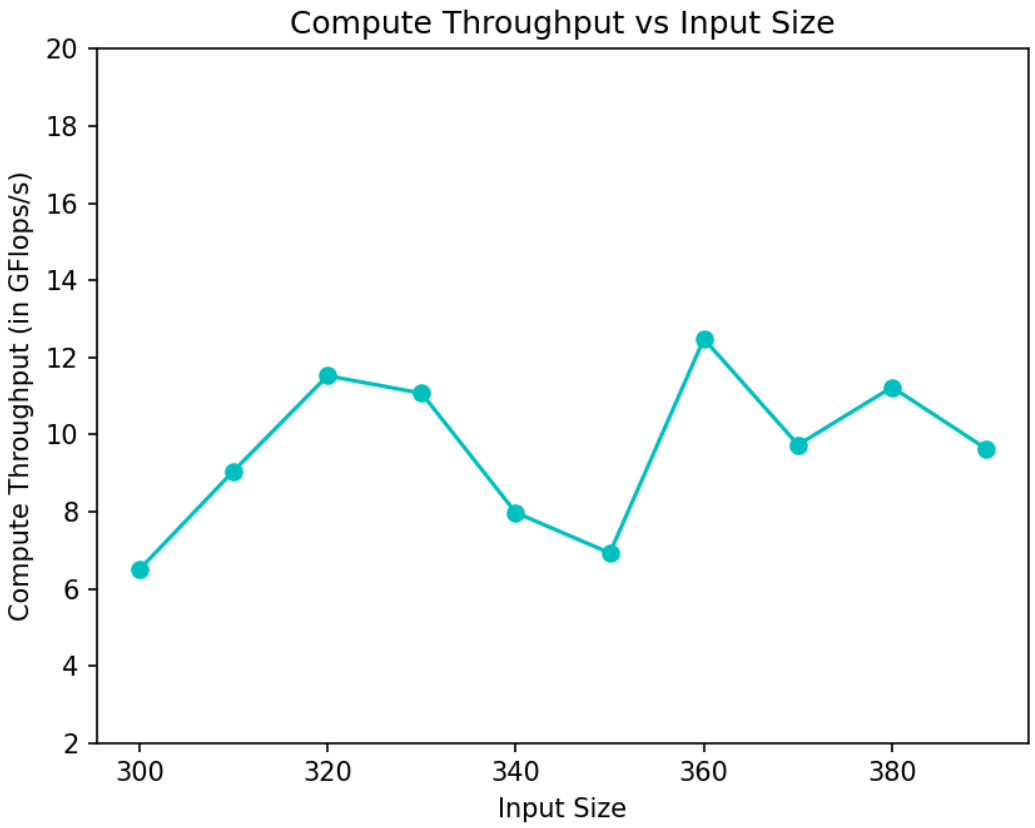
SGEMMC

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmc
Time (in milli-secs) 5.833000
Memory Bandwidth (in GBytes/s): 0.185153
Compute Throughput (in GFlops/s): 9.288531
Time (in milli-secs) 11.133000
Memory Bandwidth (in GBytes/s): 0.103584
Compute Throughput (in GFlops/s): 5.369101
Time (in milli-secs) 9.441000
Memory Bandwidth (in GBytes/s): 0.130156
Compute Throughput (in GFlops/s): 6.963330
Time (in milli-secs) 12.475000
Memory Bandwidth (in GBytes/s): 0.104754
Compute Throughput (in GFlops/s): 5.778902
Time (in milli-secs) 12.724000
Memory Bandwidth (in GBytes/s): 0.109022
Compute Throughput (in GFlops/s): 6.196102
Time (in milli-secs) 16.647000
Memory Bandwidth (in GBytes/s): 0.088304
Compute Throughput (in GFlops/s): 5.165796
Time (in milli-secs) 13.146000
Memory Bandwidth (in GBytes/s): 0.118302
Compute Throughput (in GFlops/s): 7.117846
Time (in milli-secs) 11.587000
Memory Bandwidth (in GBytes/s): 0.141780
Compute Throughput (in GFlops/s): 8.766704
Time (in milli-secs) 11.521000
Memory Bandwidth (in GBytes/s): 0.150404
Compute Throughput (in GFlops/s): 9.550629
Time (in milli-secs) 11.738000
Memory Bandwidth (in GBytes/s): 0.155495
Compute Throughput (in GFlops/s): 10.133089
```


Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
Input Size
16
14
12
10
8
6
4
2
300
320
340
360
380

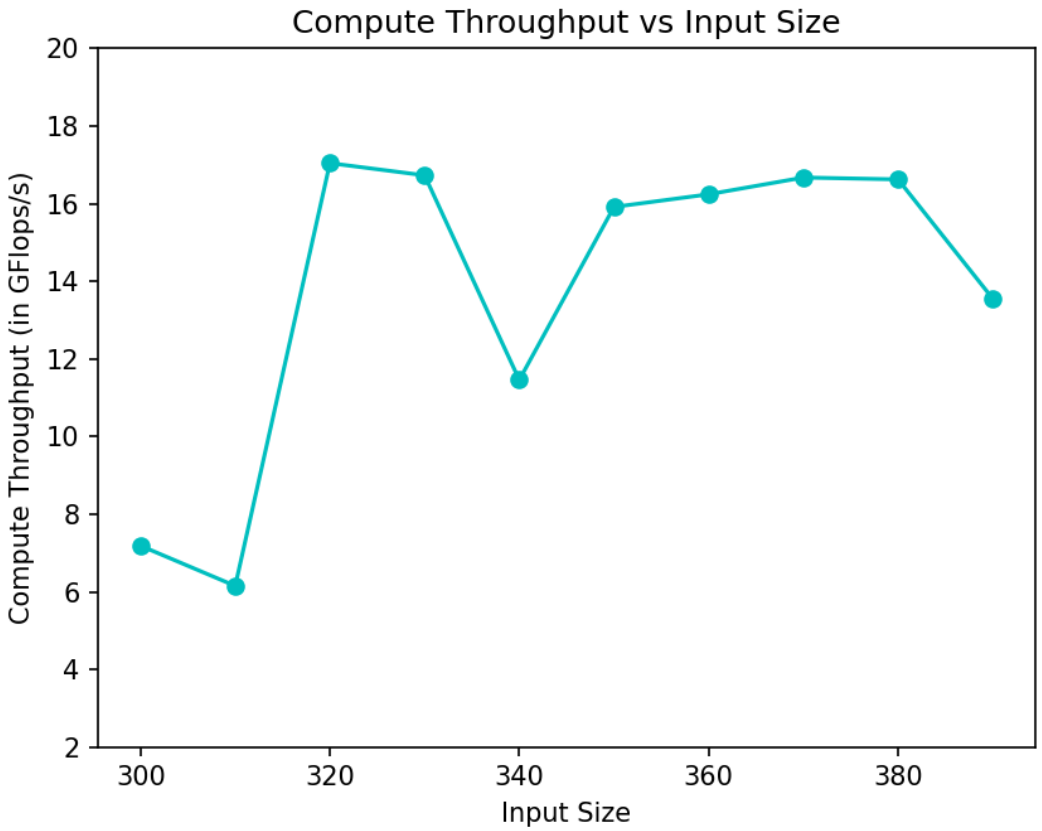
SGEMMCBT


codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmcbt
Time (in milli-secs) 6.164000
Memory Bandwidth (in GBytes/s): 0.175211
Compute Throughput (in GFlops/s): 8.789747
Time (in milli-secs) 7.054000
Memory Bandwidth (in GBytes/s): 0.163482
Compute Throughput (in GFlops/s): 8.473802
Time (in milli-secs) 6.161000
Memory Bandwidth (in GBytes/s): 0.199448
Compute Throughput (in GFlops/s): 10.670476
Time (in milli-secs) 10.821000
Memory Bandwidth (in GBytes/s): 0.120765
Compute Throughput (in GFlops/s): 6.662212
Time (in milli-secs) 11.376000
Memory Bandwidth (in GBytes/s): 0.121941
Compute Throughput (in GFlops/s): 6.930309
Time (in milli-secs) 10.267000
Memory Bandwidth (in GBytes/s): 0.143177
Compute Throughput (in GFlops/s): 8.375864
Time (in milli-secs) 8.210000
Memory Bandwidth (in GBytes/s): 0.189428
Compute Throughput (in GFlops/s): 11.397223
Time (in milli-secs) 9.030000
Memory Bandwidth (in GBytes/s): 0.181927
Compute Throughput (in GFlops/s): 11.249147
Time (in milli-secs) 9.329000
Memory Bandwidth (in GBytes/s): 0.185743
Compute Throughput (in GFlops/s): 11.794705
Time (in milli-secs) 12.078000
Memory Bandwidth (in GBytes/s): 0.151118
Compute Throughput (in GFlops/s): 9.847839


Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
Input Size
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380

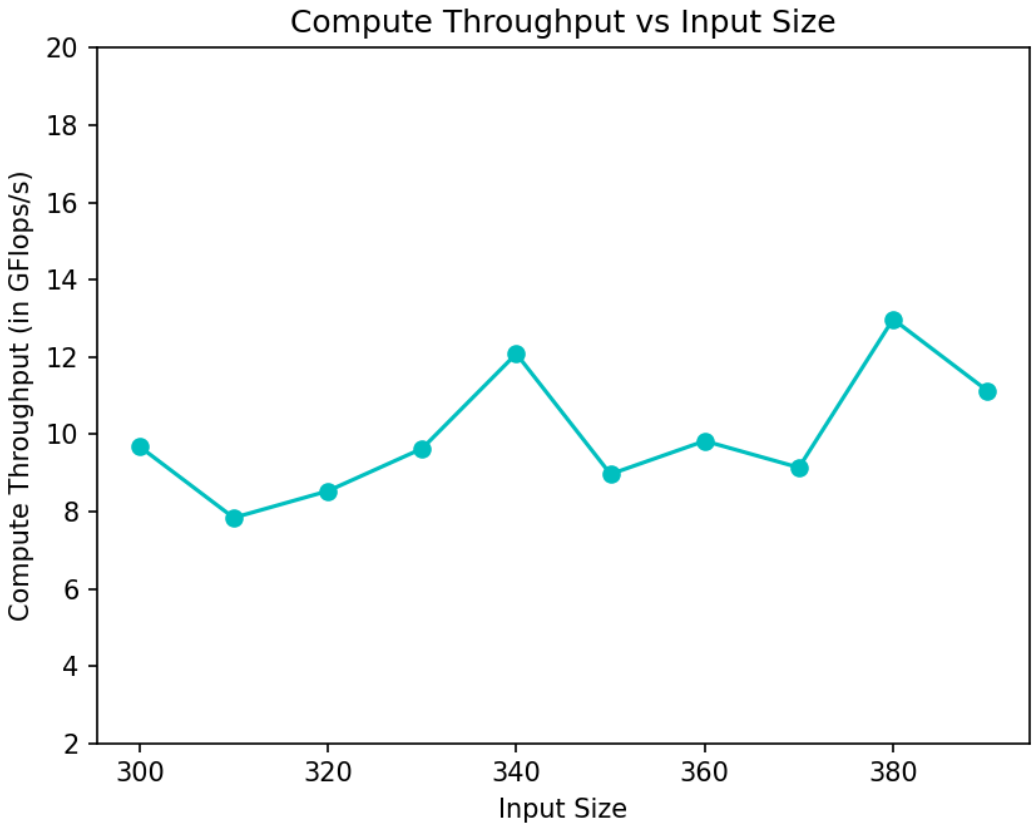
SGEMMCAT

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmcat
Time (in milli-secs) 6.186000
Memory Bandwidth (in GBytes/s): 0.174588
Compute Throughput (in GFlops/s): 8.758487
Time (in milli-secs) 7.632000
Memory Bandwidth (in GBytes/s): 0.151101
Compute Throughput (in GFlops/s): 7.832049
Time (in milli-secs) 10.714000
Memory Bandwidth (in GBytes/s): 0.114691
Compute Throughput (in GFlops/s): 6.135972
Time (in milli-secs) 22.365000
Memory Bandwidth (in GBytes/s): 0.058431
Compute Throughput (in GFlops/s): 3.223421
Time (in milli-secs) 5.048000
Memory Bandwidth (in GBytes/s): 0.274802
Compute Throughput (in GFlops/s): 15.617908
Time (in milli-secs) 8.705000
Memory Bandwidth (in GBytes/s): 0.168868
Compute Throughput (in GFlops/s): 9.878805
Time (in milli-secs) 9.252000
Memory Bandwidth (in GBytes/s): 0.168093
Compute Throughput (in GFlops/s): 10.113619
Time (in milli-secs) 7.768000
Memory Bandwidth (in GBytes/s): 0.211483
Compute Throughput (in GFlops/s): 13.076699
Time (in milli-secs) 6.602000
Memory Bandwidth (in GBytes/s): 0.262466
Compute Throughput (in GFlops/s): 16.666586
Time (in milli-secs) 10.290000
Memory Bandwidth (in GBytes/s): 0.177376
Compute Throughput (in GFlops/s): 11.559009
```


Compute Throughput vs Input Size
Compute Throughput (in GFlops/s)
Input Size
20
18
16
14
12
10
8
6
4
2
300
320
340
360
380

SGEMMCATBT

```
codeubuntu@LAPTOP-UJL6109Q:saxpy$ ./saxpy sgemmcatbt
Time (in milli-secs) 6.114000
Memory Bandwidth (in GBytes/s): 0.176644
Compute Throughput (in GFlops/s): 8.861629
Time (in milli-secs) 6.983000
Memory Bandwidth (in GBytes/s): 0.165144
Compute Throughput (in GFlops/s): 8.559960
Time (in milli-secs) 5.885000
Memory Bandwidth (in GBytes/s): 0.208802
Compute Throughput (in GFlops/s): 11.170909
Time (in milli-secs) 9.202000
Memory Bandwidth (in GBytes/s): 0.142013
Compute Throughput (in GFlops/s): 7.834362
Time (in milli-secs) 6.649000
Memory Bandwidth (in GBytes/s): 0.208633
Compute Throughput (in GFlops/s): 11.857302
Time (in milli-secs) 10.449000
Memory Bandwidth (in GBytes/s): 0.140683
Compute Throughput (in GFlops/s): 8.229974
Time (in milli-secs) 8.475000
Memory Bandwidth (in GBytes/s): 0.183504
Compute Throughput (in GFlops/s): 11.040850
Time (in milli-secs) 12.619000
Memory Bandwidth (in GBytes/s): 0.130185
Compute Throughput (in GFlops/s): 8.049750
Time (in milli-secs) 11.212000
Memory Bandwidth (in GBytes/s): 0.154549
Compute Throughput (in GFlops/s): 9.813842
Time (in milli-secs) 13.083000
Memory Bandwidth (in GBytes/s): 0.139509
Compute Throughput (in GFlops/s): 9.091355
```




```
#
#  -- Example 1 --
#

calctime: 8.375000
```

BLIS OUTPUT OF sgemm

Observation

- Row Major A B
  - Best results were obtained from using gcc -O3
  - Naive Approach
  - The naive approach is to use call level 2 where we pass matrix A and coulmn of matrix B then we would recieve coulmn of matrix C. This approach is not

good beacuse stride of coulmn matrix B would be lda and stride of coulmn matrix obtained would be ldc.

- If AB = C then, B'A' = C'

  Using this we can call level 2 code where we pass B' and row of matrix A, the output obtained will be the row of matrix C. This results in stride 1 access of A and stride 1 access of C. Now we can call

```
for(int i=0;i<M;i++)
{
    cblas_sgemv(CblasRowMajor,CblasTrans,N,K,alpha,B,ldb,A+i*lda,1,beta,C+i*ldc,1);
}
```

  We can't parallelize this function because it will lead to race condition. Also the internal function is already obtimised while doing level 2.

- Row Major A B'
  - Best results were obtained from using icc -O3 with parallelization
  - Using the same approach we were able to reduce the stride access to 1 for both C and A.
  - We can also parallelize this code that is why icc was giving best results as icc is good in parallelization

```
#pragma omp for simd
for(int i=0;i<M;i++)
{
   cblas_sgemv(CblasRowMajor,CblasNoTrans,K,N,alpha,B,N,A+i*lda,1,beta,C+i*ldc,1);
}
```

- Row Major A' B
  - Best results were obtained from using gcc -O3
  - Using the same approach we were able to reduce the stride access for output C to 1 but we were not able to reduce stride access for A.
  - Parallelization was not possible beacuse of race condition

```
for(int i=0;i<M;i++)
{
```

```
  cblas_sgemv(CblasRowMajor,CblasTrans,N,K,alpha,B,ldb,A+i,lda,beta,C+i*ldc,1);
}
```

- Row Major A’ B’
  - Best results were obtained from using icc -O3 with parallelization
  - Using the same approach we were able to reduce the stride access for output C to 1 but we were not able to reduce stride access for A
  - We can also parallelize this code that is why icc was giving best results as icc is good in parallelization

```
#pragma omp parallel for simd
for(int i=0;i<M;i++)
{
   cblas_sgemv(CblasRowMajor,CblasNoTrans,K,N,alpha,B,N,A+i,lda,beta,C+i*ldc,1);
}
```

- Column Major
  - AB

    It is similar to row major A’B’
  - AB’

    It is similar to row major A’B
  - A’B

    It is similar to row major AB’
  - A’B’

    It is similar to row major AB

The Program is Memory Bound because { O.I * Memory Bandwidth < Theoretically Observed Peak Flops }

Stencil Program

▼ Analysis

Logic

Using the two outer loops we can reach out to the destination cell of the output image, now we need to find the product of the grid with the stencil, this is done in the two inner loops. To deal with zero padding the width allocated is w + k -1 and the height allocated is h + k -1.

```
for (int i = 0; i < h; i++)
    {
        for (int j = 0; j < w; j++)
        {
            Y[i*w +j] = 0.0;
            for (int m = 0; m < k; m++)
            {
                for (int n = 0; n < k; n++)
                {
                    Y[i*w +j] += X[(i + m) * (actual_w) + j + n] * S[m * k + n];
                }
            }
        }
    }
```

h = height of the image

w = width of the image

k = size of stencil

Number of Operation Performed = 2*h*w*k*k

Reasoning

In the inner most loop we are performing 2 operations and that loop is called h*w*k*k times

Memory Fetched = 4*(h*w)*(2*k*k)

Reasoning

In the inner most loop we are fetching 2 floating number(X[(i + m) * (actual_w) + j + n] and S[m * k + n]), this implies we are fetching 8 bytes of memory, that inner most loop is called h*w*k*k times therefore Memory fetched is equal to 8*h*w*k*k

Operational Intensity = Number of operations / Memory Fetched = 1/4

Using gcc -O0

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
h
3
Time (in milli-secs) 66.235000
Memory Bandwidth (in GBytes/s): 2.254083
Compute Throughput (in GFlops/s): 0.563521
4
Time (in milli-secs) 107.958000
Memory Bandwidth (in GBytes/s): 2.458556
Compute Throughput (in GFlops/s): 0.614639
5
Time (in milli-secs) 165.132000
Memory Bandwidth (in GBytes/s): 2.511445
Compute Throughput (in GFlops/s): 0.627861
6
Time (in milli-secs) 230.719000
Memory Bandwidth (in GBytes/s): 2.588416
Compute Throughput (in GFlops/s): 0.647104
7
Time (in milli-secs) 308.159000
Memory Bandwidth (in GBytes/s): 2.637766
Compute Throughput (in GFlops/s): 0.659441
8
Time (in milli-secs) 398.560000
Memory Bandwidth (in GBytes/s): 2.663798
Compute Throughput (in GFlops/s): 0.665949
9
Time (in milli-secs) 500.305000
Memory Bandwidth (in GBytes/s): 2.685747
Compute Throughput (in GFlops/s): 0.671437
```

FOR HD

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
u
3
Time (in milli-secs) 262.259000
Memory Bandwidth (in GBytes/s): 2.277126
Compute Throughput (in GFlops/s): 0.569282
4
Time (in milli-secs) 445.123000
Memory Bandwidth (in GBytes/s): 2.385146
Compute Throughput (in GFlops/s): 0.596286
5
Time (in milli-secs) 668.466000
Memory Bandwidth (in GBytes/s): 2.481622
Compute Throughput (in GFlops/s): 0.620406
6
Time (in milli-secs) 945.354000
Memory Bandwidth (in GBytes/s): 2.526871
Compute Throughput (in GFlops/s): 0.631718
7
Time (in milli-secs) 1265.977000
Memory Bandwidth (in GBytes/s): 2.568297
Compute Throughput (in GFlops/s): 0.642074
8
Time (in milli-secs) 1650.864000
Memory Bandwidth (in GBytes/s): 2.572430
Compute Throughput (in GFlops/s): 0.643108
9
Time (in milli-secs) 2072.733000
Memory Bandwidth (in GBytes/s): 2.593084
Compute Throughput (in GFlops/s): 0.648271
```

FOR UHD

Using gcc -O3

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
h
3
Time (in milli-secs) 22.463000
Memory Bandwidth (in GBytes/s): 6.646450
Compute Throughput (in GFlops/s): 1.661612
4
Time (in milli-secs) 22.819000
Memory Bandwidth (in GBytes/s): 11.631570
Compute Throughput (in GFlops/s): 2.907892
5
Time (in milli-secs) 36.671000
Memory Bandwidth (in GBytes/s): 11.309209
Compute Throughput (in GFlops/s): 2.827302
6
Time (in milli-secs) 58.081000
Memory Bandwidth (in GBytes/s): 10.282137
Compute Throughput (in GFlops/s): 2.570534
7
Time (in milli-secs) 76.370000
Memory Bandwidth (in GBytes/s): 10.643593
Compute Throughput (in GFlops/s): 2.660898
8
Time (in milli-secs) 87.141000
Memory Bandwidth (in GBytes/s): 12.183510
Compute Throughput (in GFlops/s): 3.045877
9
Time (in milli-secs) 117.869000
Memory Bandwidth (in GBytes/s): 11.399883
Compute Throughput (in GFlops/s): 2.849971
```

FOR HD

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
u
3
Time (in milli-secs) 71.227000
Memory Bandwidth (in GBytes/s): 8.384416
Compute Throughput (in GFlops/s): 2.096104
4
Time (in milli-secs) 92.944000
Memory Bandwidth (in GBytes/s): 11.422827
Compute Throughput (in GFlops/s): 2.855707
5
Time (in milli-secs) 129.717000
Memory Bandwidth (in GBytes/s): 12.788455
Compute Throughput (in GFlops/s): 3.197114
6
Time (in milli-secs) 198.734000
Memory Bandwidth (in GBytes/s): 12.020023
Compute Throughput (in GFlops/s): 3.005006
7
Time (in milli-secs) 312.367000
Memory Bandwidth (in GBytes/s): 10.408925
Compute Throughput (in GFlops/s): 2.602231
8
Time (in milli-secs) 348.928000
Memory Bandwidth (in GBytes/s): 12.170800
Compute Throughput (in GFlops/s): 3.042700
9
Time (in milli-secs) 467.962000
Memory Bandwidth (in GBytes/s): 11.485486
Compute Throughput (in GFlops/s): 2.871372
```

FOR UHD

Using icc -O3

```
codeubuntu@LAPTOP-UJL6109Q:spp$ icc -O3 stencil.c
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
h
3 1.213459
4 1.408247
5 2.018456
6 2.183279
7 2.202825
8 3.237313
9 2.533568
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
u
3 1.628997
4 1.821181
5 1.927370
6 1.549724
7 1.302058
8 1.576854
9 1.578761
```

The output denotes stencil size and GFlops

Baseline Time of execution is dependent upon the stencil size, that is coming in the case without using any optimisation flags

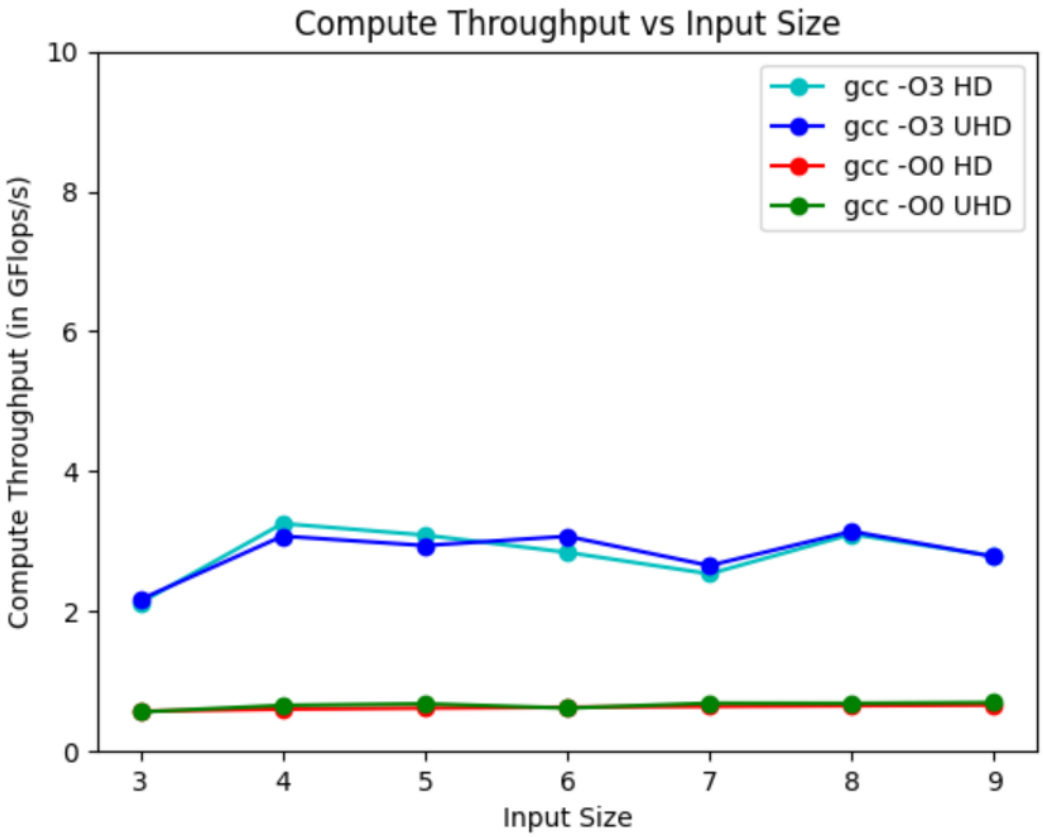
Baseline Flops = 0.56

Best Execution Time is also dependent upon the stencil size so it can’t be used as a parameter

Best Flops achieved = 3.19

Speedup = 3.19 / 0.56 = 5.69

Main Memory Bandwidth = 12.78



Techniques

1. Using parallelization

   Parallelisation is not possible in this code since there is a race condition when trying to write at Y[i*w + j].

```
codeubuntu@LAPTOP-UJL61O9Q:spp$ icc -g -O3 -axCORE-AVX2  -qopenmp stencil.c
stencil.c(64): remark #15009: main has been targeted for automatic cpu dispatch
stencil.c(31): remark #15009: RandomMatrixImage has been targeted for automatic cpu dispatch
codeubuntu@LAPTOP-UJL61O9Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
h
3 0.015231
4 0.025163
5 0.037341
6 0.050321
7 0.065408
8 0.080353
9 0.092720
```

Flops decreased

2. Using Vectorization

   On a average there was a slight increase in the GFlops/sec, this means the complier was optimizing it itself and in some cases the pragma forced it to do it

```
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
h
3 1.938045
4 2.223327
5 2.666529
6 2.561010
7 2.510939
8 4.580011
9 4.017691
codeubuntu@LAPTOP-UJL6109Q:spp$ ./a.out
Enter the image type (HD or UHD):
h FOR HD
u FOR UHD
u
3 1.962552
4 2.322690
5 2.793198
6 2.610445
7 2.589226
8 4.599217
9 4.009587
```

.

The Program is Memory Bound because { O.I * Memory Bandwidth << Theoretically Observed Peak Flops }