

SPP Assignment

Ayush Agrawal

Know Your Computer (KYC)

1. CPU model, generation, frequency range, number of cores, hyper threading availability, SIMD ISA, cache size, main memory bandwidth, etc.

Ans:

- CPU Model Name: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
 - Generation: 10th Generation
 - Frequency Range: 400 - 4200 Mhz
 - Number of Cores: 4
 - HyperThreading: 2 hyperthreads per core
 - Cache size: L1 - 128kB, L2- 1 MB, L3- 6MB
 - SIMD ISA: Intel SSE4.1, SSE4.2, AVX2
 - Main Memory Bandwidth: 45.8 GB/s (Maximum)
2. Whetstone Benchmark Result:
Loops: 1000000, Iterations: 1, Duration: 18 sec.
C Converted Double Precision Whetstones: 5555.6 MIPS
 3. Stream Benchmark Result:
Main Memory Size: 8192MB
It is DDR4.
Maximum Main Memory Bandwidth: 45.8 GB/s

```

-----
STREAM version $Revision: 5.10 $
-----
This system uses 8 bytes per array element.
-----
Array size = 100000000 (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.
-----
Number of Threads requested = 4
Number of Threads counted = 4
-----
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 14401 microseconds.
(= 14401 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:         12799.0    0.012700    0.012501    0.013387
Scale:        12737.8    0.012778    0.012561    0.013463
Add:          13576.4    0.017821    0.017678    0.018192
Triad:        13625.4    0.017985    0.017614    0.020262
-----
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-----

```

4. Secondary Storage Size: 512 GB

It is SDD.

Know Your Cluster

Peak FLOPS for Ada Cluster: 70.66 TFLOPS (CPU) + 4588 TFLOPS (FP32 GPU).

Peak FLOPS for Abacus Cluster: 14 TFLOPS

BLAS Level 1

(xSCAL) Scalar multiplication of a vector.

$$X = \alpha X$$

SINGLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed = N multiplications.

Memory accesses = N memory access * 4 Bytes each = 4N bytes

Operational Intensity = $N/4N = 1/4 = 0.25$

Execution Times:

N = 10 million

Using gcc -O3 :


- a. Execution Time: 5.98 ms
- b. GFLOPS: 1.65 GFLOPS/s
- c. Memory Bandwidth: 6.68 GB/s

Using icc -O3 :

- a. Execution Time: 7.25 ms
- b. GFLOPS: 1.34 GFLOPS/s
- c. Memory Bandwidth: 5.51 GB/s

Baseline & Best Execution Time:

Baseline execution time is 27.56 ms (without using -O3 option).



On Vectorizing using SSE instruction set and using -O3, Execution time drops to 6.80 ms (using icc compiler).

On multithreading, there is no effect on the runtime and it slightly increases.

Using gcc compiler and -O3 option, the execution time remains almost the same even after vectorization and multithreading.

Speedup:

In icc compiler: $27.56/6.80 = 4.05X$ speedup.

In gcc compiler: $27.56/7.25 = 3.80X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.37, Optimized GFLOPS/s = 1.48

Using gcc: Baseline GLOPS/s = 0.36, Optimized GLOPS/s = 1.65

Optimization Strategies:

Use the -O3 flag for compilation,

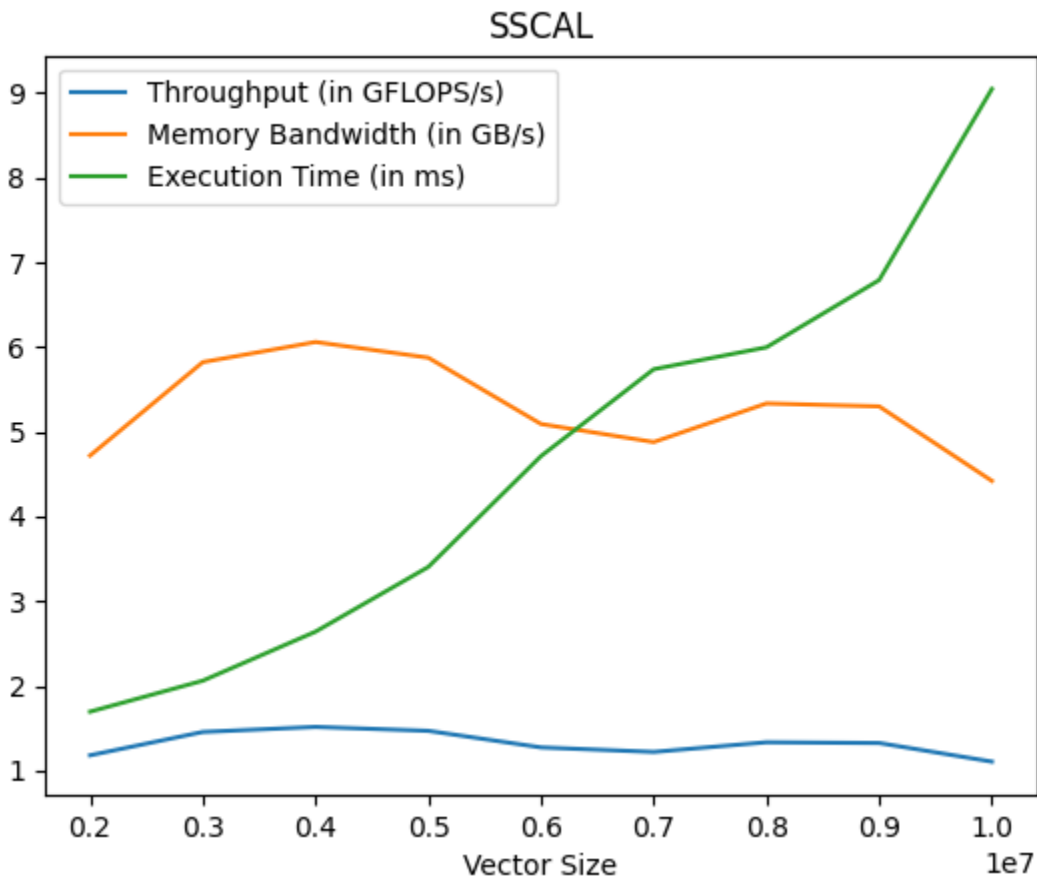
Vectorize the for loop using `#pragma omp simd`.

Memory Bandwidth Achieved:

Using icc compiler: 5.60 GB/s

Using gcc compiler: 6.68 GB/s

The problem is clearly memory bound as the main memory is not able to supply the data at the rate at which it can perform floating point operations. This is the reason why vectorization or multi-threading does not have any significant impact on the performance as memory bandwidth is the primary issue which places an upper bound on the performance of the program.



DOUBLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed = N multiplications.

Memory accesses = N memory access * 8 Bytes each = $8N$ bytes

Operational Intensity = $N/8N = 1/8 = 0.125$

Execution Times:

$N = 10$ million

Using gcc -O3 :

- d. Execution Time: 11.51 ms
- e. GFLOPS: 0.86 GFLOPS/s
- f. Memory Bandwidth: 6.94 GB/s

Using icc -O3 :

- d. Execution Time: 12.25 ms
- e. GFLOPS: 0.81 GFLOPS/s
- f. Memory Bandwidth: 6.52 GB/s

Baseline & Best Execution Time:

Baseline execution time is 27.77 ms (without using -O3 option).

On Vectorizing using SSE instruction set and using -O3, Execution time drops to 12.15 ms (using icc compiler) and to 11.51 ms (using gcc compiler).

On multithreading, there is no effect on the runtime and it slightly increases.

Speedup:

In icc compiler: $27.77/12.15 = 2.28X$ speedup.

In gcc compiler: $27.56/11.51 = 2.39X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.37, Optimized GFLOPS/s = 0.86

Using gcc: Baseline GLOPS/s = 0.35, Optimized GLOPS/s = 0.87

Optimization Strategies:

Use the -O3 flag for compilation,

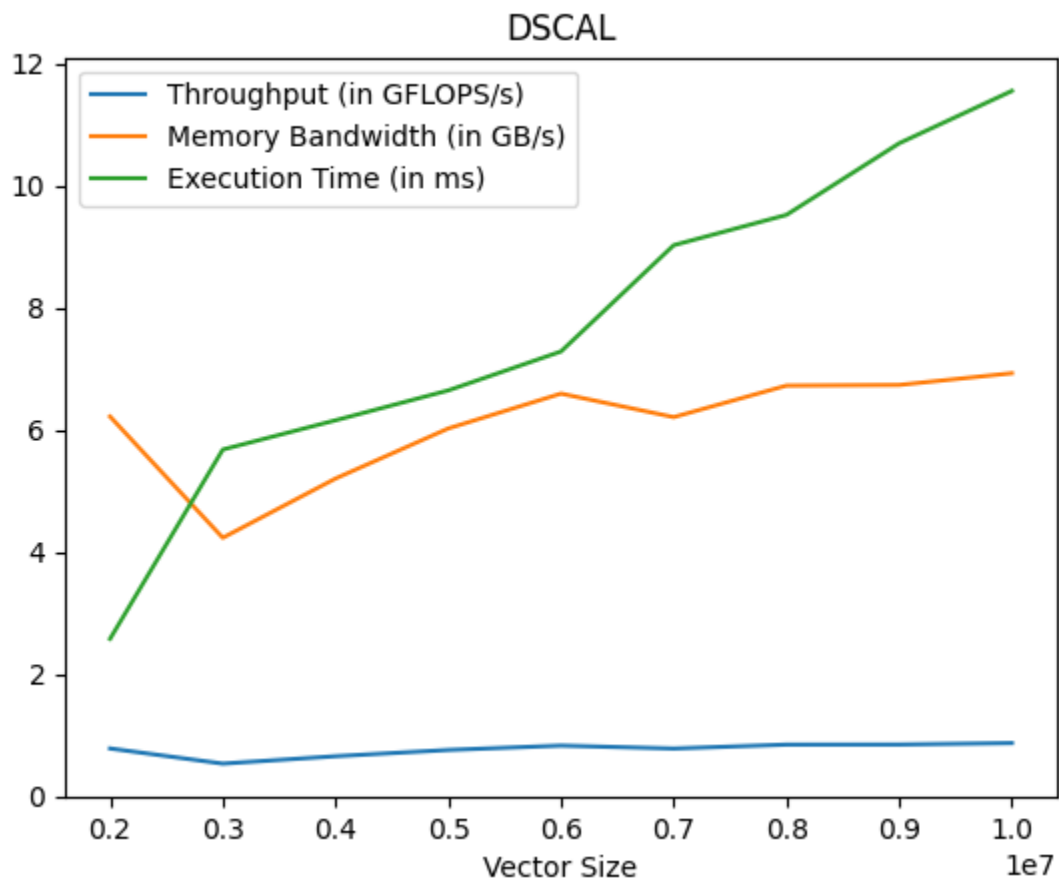
Vectorize the for loop using `#pragma omp simd.`

Memory Bandwidth Achieved:

Using icc compiler: 6.94 GB/s

Using gcc compiler: 6.92 GB/s

The problem is memory bound as we are not able to achieve peak computation flops.



BLAS Level 1

(xAXPY) Scalar Multiplication of a vector followed by addition with another vector.

$$Y = \alpha X + Y$$

SINGLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed = N multiplications + N additions = 2N

Memory accesses = 2N memory access (for each array X and Y) * 4 Bytes each = 8N bytes

Operational Intensity = $2N/8N = 1/4 = 0.25$

Execution Times:

Using gcc -O3 :


- g. Execution Time: 9.01 ms
- h. GFLOPS: 2.25 GFLOPS/s
- i. Memory Bandwidth: 9.02 GB/s

Using icc -O3 :

- g. Execution Time: 10.01 ms
- h. GFLOPS: 2.02 GFLOPS/s
- i. Memory Bandwidth: 8.10 GB/s

Baseline & Best Execution Time:

Baseline execution time is 29.21 ms. (without using -O3 option).



On Vectorizing using SSE instruction set and using -O3, Execution time drops to 9.81 ms (using icc compiler). On multithreading, the execution time remains the same.

We observe the same pattern for gcc compiler.

Hence, best execution time is 9 ms (using gcc).

Speedup:

In icc compiler: $29.21/9.81 = 2.97X$ speedup.

In gcc compiler: $29.21/9.01 = 3.24X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.67, Optimized GFLOPS/s = 2.02

Using gcc: Baseline GLOPS/s = 0.67, Optimized GLOPS/s = 2.29

Optimization Strategies:

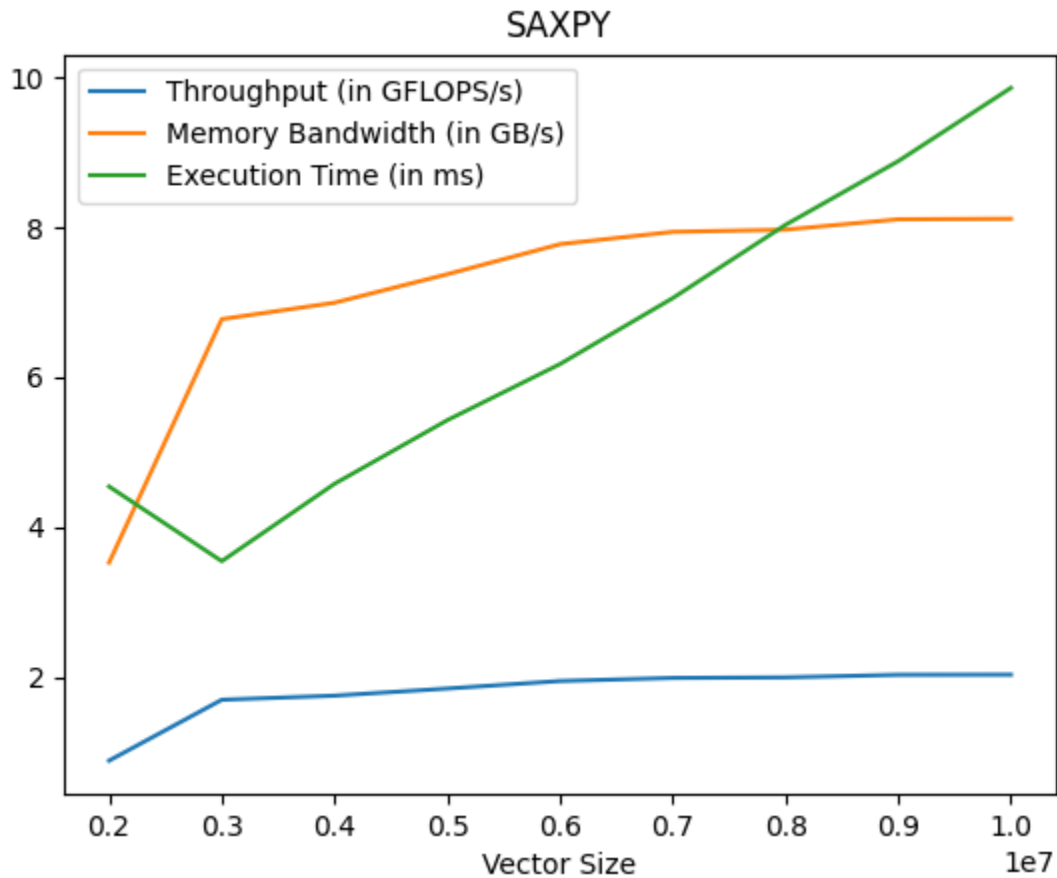
Use the -O3 flag for compilation, Vectorize the for loop using #pragma omp simd.

Memory Bandwidth Achieved:

Using icc compiler: 8.09 GB/s

Using gcc compiler: 9.18 GB/s

The problem is clearly memory bound as the main memory is not able to supply the data at the rate at which it can perform floating point operations. This is the reason why vectorization or multi-threading does not have any significant impact on the performance as memory bandwidth is the primary issue which places an upper bound on the performance of the program.



DOUBLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed = N multiplications + N additions = $2N$

Memory accesses = $2N$ memory access (for each array X and Y) * 8 Bytes each = $16N$ bytes

Operational Intensity = $2N/16N = 1/8 = 0.125$

Execution Times:

Using gcc -O3 :

- j. Execution Time: 17.52 ms
- k. GFLOPS: 1.14 GFLOPS/s
- l. Memory Bandwidth: 9.12 GB/s

Using icc -O3 :

- j. Execution Time: 17.37 ms
- k. GFLOPS: 1.15 GFLOPS/s
- l. Memory Bandwidth: 9.21 GB/s

Baseline & Best Execution Time:

Baseline execution time is 30.25 ms. (without using -O3 option).

On Vectorizing using SSE instruction set and using -O3, Execution time drops to 17.34 ms (using icc compiler). On multithreading, the execution time remains the same.

We observe the same pattern for gcc compiler.

Hence, best execution time is 17.34 ms (using gcc).

Speedup:

In icc compiler: $30.25/17.34 = 1.74X$ speedup.

In gcc compiler: $30.25/17.34 = 1.75X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.64, Optimized GFLOPS/s = 1.15

Using gcc: Baseline GLOPS/s = 0.65, Optimized GLOPS/s = 1.13

Optimization Strategies:

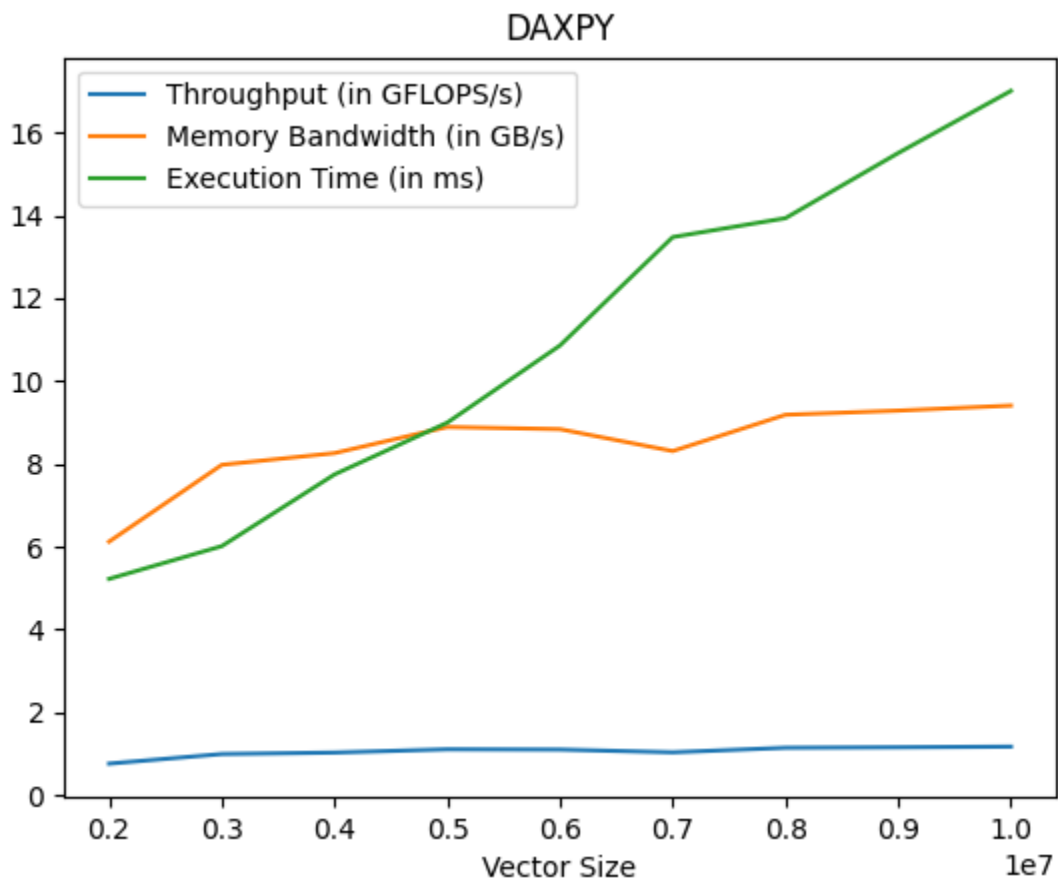
Use the -O3 flag for compilation, Vectorize the for loop using #pragma omp simd.

Memory Bandwidth Achieved:

Using icc compiler: 8.56 GB/s

Using gcc compiler: 9.22 GB/s

The program is memory bound.



BLAS Level 2

(xGEMV) Matrix times a vector.

$$Y = \alpha AX + \beta Y \text{ or } Y = \alpha A^T X + \beta Y$$

Dot Product Formulation for computing AX:

```
for (int i = 0; i < M; ++i)
{
    Y[i] = alpha * cblas_sdot(N, A + i * lda, 1, X, incX) +
    beta * Y[i];
}
```

We consider the matrix of dimensions M X N stored in a row major fashion. Here, we compute the dot product of every row of Matrix A with the vector X. This has a very good memory locality when matrices are stored in row major fashion as it offers stride 1 memory reference pattern leading to low cache misses. However, if we wish to compute $A^T X$, then performance decreases steeply due to stride N reference pattern.

SINGLE PRECISION ANALYSIS:

Operational Intensity:

Number of floating point operations performed =

Assuming the dimensions of matrix as M x N, vector X as N x 1, and vector Y as M x 1,

We require N multiplications and N - 1 additions to generate each element of AX. This is done M times. Then M multiplications for multiplying alpha with AX and M multiplications for multiplying beta with Y. Then we need M additions to get the final result.

Therefore, total FLOPS = $M(2N - 1) + M + M + M = 2MN + 2M$

Memory accesses = MN memory access for matrix, N memory accesses for X and M for Y.

Therefore, total Memory Access = $(MN + N + M) * 4$ Bytes

Operational Intensity = $2(MN + 2)/4(MN + N + M)$

When $M = N$,

Operational Intensity = $\frac{1}{2} = 0.5$

Execution Times:

$N = 20000$

Using gcc -O3 :

- m. Execution Time: 497 ms
- n. GFLOPS: 1.60 GFLOPS/s
- o. Memory Bandwidth: 3.21 GB/s


Using icc -O3 :

- m. Execution Time: 210 ms
- n. GFLOPS: 3.80 GFLOPS/s
- o. Memory Bandwidth: 7.62 GB/s

Baseline & Best Execution Time:

Baseline execution time is 1123 ms. (without using -O3 option).

On using -O3 option, Execution time drops to 210 ms using icc and to 500 ms using gcc.



On Vectorizing using SSE instruction set, it further drops to 132 ms using icc and to 250 ms using gcc.

On Multithreading using 4 threads, it further drops to 120 ms using icc and to 131 ms using gcc.

Therefore, the best execution time is 120 ms using icc and 131 ms using gcc.

Speedup:

In icc compiler: $1123/120 = 9.35X$ speedup.

In gcc compiler: $1123/131 = 8.57$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.71, Optimized GFLOPS/s = 6.63

Using gcc: Baseline GLOPS/s = 0.71, Optimized GLOPS/s = 6.07

Optimization Strategies:

Use the -O3 flag for compilation, Vectorize the for loop using `#pragma omp simd`, Parallelize using 4 threads by `#pragma omp parallel for simd`.

The best time is given by all these optimizations combined.

Memory Bandwidth Achieved:

Using icc compiler: 12.83 GB/s

Using gcc compiler: 12.15 GB/s

The program is memory bound as in my local machine, the peak achievable flops is 53.58 GFLOPS/s but we are able to achieve only 6.63 GFLOPS/s as the best due to the limitations on the memory bandwidth.

Observations:

On using the dot product formulation (using `cblas_sdot`) for $Y = \alpha A^T X + \beta Y$, we notice that the performance decreases steeply. **It takes 5030 ms, and GFLOPS/s goes down to 0.15.** Even after optimizations using -O3 and vectorization and multithreading, the time taken is around 4200 ms. This is because of the poor memory access pattern of the `cblas_sdot` routine. It makes stride N memory reference pattern which leads to cache misses and then repopulating caches which is a very expensive and time consuming operation. Therefore, we use `cblas_saxpy` instead of the dot product routine to overcome this locality of reference problem. It uses the idea that the product AX can be represented as the linear combination of columns of matrix A. It makes stride 1 reference pattern and hence its cache locality is very good. On using this, we are able to achieve approximately 6.53 GFLOPS/s which is a very good improvement over the earlier 0.15 GLOPS/s and is almost equal to the dot product formulation for the earlier problem.

SAXPY Formulation for computing $A^T X$:

```
for (int i = 0; i < M; ++i)
{
    cblas_saxpy(N, X[i], A + i * M, 1, temp, 1);
}
```

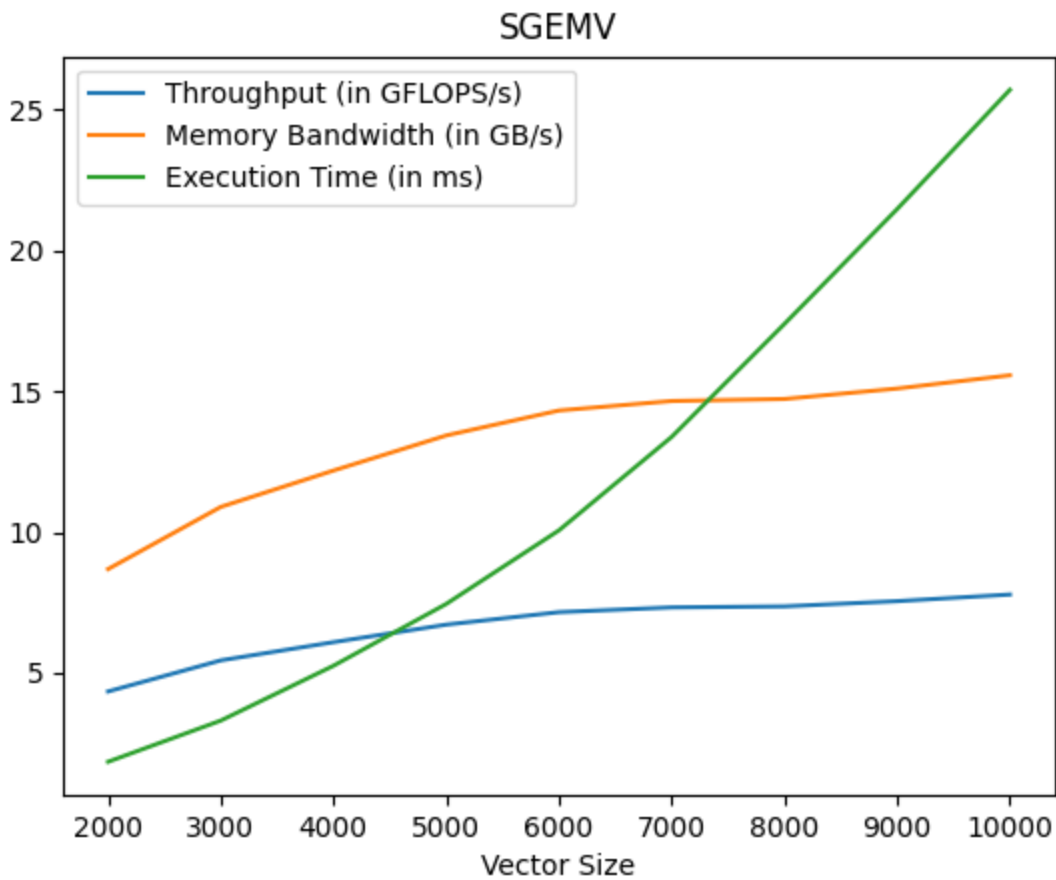
Here, we accumulate the linear combination of rows of A and elements of X $A_i x_I$ where A_i is ith row in the original matrix (which would be the column in the transpose matrix). This gives us a stride 1 reference pattern which improves the performance significantly. This is stored in a temporary vector. We then compute Y using the following loop:

```
for (int i = 0; i < N; i++)
{
    Y[i] = alpha * temp[i] + beta * Y[i];
}
```

```
}
```

Similarly, when the matrix is stored in a column major fashion, we use the `cblas_saxpy` subroutine to avoid making stride N memory reference and thus maintain a good cache locality.

| | Row Major | Column Major |
|--------------|--------------------------|--------------------------|
| No Transpose | <code>cblas_dot</code> | <code>cblas_saxpy</code> |
| Transpose | <code>cblas_saxpy</code> | <code>cblas_dot</code> |



DOUBLE PRECISION ANALYSIS:

Operational Intensity:

Number of floating point operations performed =

Assuming the dimensions of matrix as $M \times N$, vector X as $N \times 1$, and vector Y as $M \times 1$,

We require N multiplications and $N - 1$ additions to generate each element of AX . This is done M times. Then M multiplications for multiplying alpha with AX and M multiplications for multiplying beta with Y . Then we need M additions to get the final result.

Therefore, total FLOPS = $M(2N - 1) + M + M + M = 2MN + 2M$

Memory accesses = MN memory access for matrix, N memory accesses for X and M for Y .

Therefore, total Memory Access = $(MN + N + M) * 8$ Bytes

Operational Intensity = $2(MN + 2)/8(MN + N + M)$

When $M = N$,

Operational Intensity = $\frac{1}{4} = 0.25$

Execution Times:

$N = 20000$

Using gcc -O3 :

- p. Execution Time: 426 ms
- q. GFLOPS: 0.47 GFLOPS/s
- r. Memory Bandwidth: 1.87 GB/s



Using icc -O3 :

- p. Execution Time: 202 ms
- q. GFLOPS: 1.97 GFLOPS/s
- r. Memory Bandwidth: 7.91 GB/s

Baseline & Best Execution Time:

Baseline execution time is 1048 ms. (without using -O3 option).

On using -O3 option, Execution time drops to 202 ms using icc and to 426 ms using gcc.

On Vectorizing using SSE instruction set, it further drops to 124 ms using icc and to 242 ms using gcc.

On Multithreading using 4 threads, it further drops to 120 ms using icc and to 131 ms using gcc.

Therefore, the best execution time is 120 ms using icc and 131 ms using gcc.

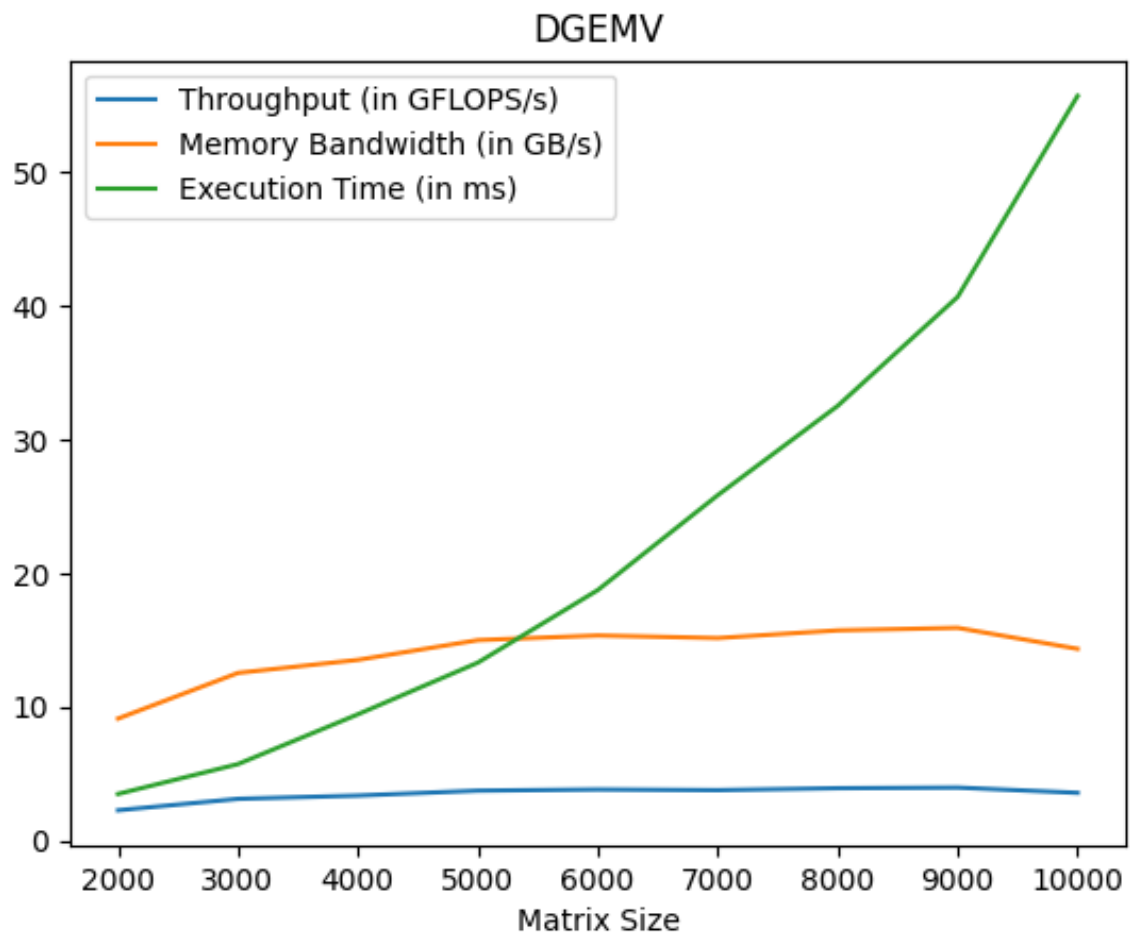
Optimization Strategies:

Use the -O3 flag for compilation, Vectorize the for loop using `#pragma omp simd`, Parallelize using 4 threads by `#pragma omp parallel for simd`. The best time is given by all these optimizations combined.

Memory Bandwidth Achieved:

Using icc compiler: 15.94 GB/s

Using gcc compiler: 14.24 GB/s



BLAS Level 3

(xGEMM) Matrix-Matrix Multiplication:

$$C = \alpha op(A)op(B) + \beta C \text{ where } op(A) = A \text{ or } A^T$$

Dot Product Formulation (i-j-p):

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

Consider the dimensions of matrix A as M X N, B as N X K and C as M X K. We compute the dot product of ith row of matrix A with jth column of matrix B to get the ij element of matrix C. It makes stride 1 reference for matrix A and stride K reference for matrix B.

SINGLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed:

Considering all dimensions as N, we need to perform N multiplications and N-1 additions for each element of AB. There are N² elements in total, therefore, N²(2N - 1) operations. Then N² operations for multiplying alpha with AB and N² operations for multiplying beta with C. At last, we need N² operations to add these numbers.

—

Total no of FLOPS = $N^2(2N - 1) + N^2 + N^2 + N^2 = 2N^3 + 2N^2$

No of Memory Access:

$3N^2$ (N^2 for each matrix) * 4 Bytes = $12 N^2$

Operational Intensity = $(2N^3 + 2N^2)/12N^2 = N/6$

Execution Times:

N = 500

Using gcc -O3 :

- s. Execution Time: 148.45 ms
- t. GFLOPS: 1.69 GFLOPS/s
- u. Memory Bandwidth: 0.0197 GB/s

Using icc -O3 :

- s. Execution Time: 77.71 ms
- t. GFLOPS: 3.22 GFLOPS/s
- u. Memory Bandwidth: 0.038 GB/s


Baseline & Best Execution Time:

Baseline execution time is 390 ms. (without using -O3 option).

On using -O3 option, Execution time drops to 77 ms using icc and to 148 ms using gcc.

On Vectorization of inner for loop, the execution time remains unaffected.

On Multithreading using 4 threads along with vectorization of inner for loop, the execution time drops considerably to 25.93ms using icc and to 40.99ms using gcc.



On applying the same optimization to outer for loop, the execution time further drops to 20ms using icc and to 38.38ms using gcc.

The parallelization of outer for loop yields lower execution time because the threads can independently execute without having to wait for all the threads to complete as in the case of inner for loop. If the inner for loop is parallelized, then a thread which has completed execution will need to wait for all other threads to complete execution in order to start the next iteration of outer for loop. This affects the performance and the execution time.

Speedup:

In icc compiler: $390/20 = 19.5X$ speedup.

In gcc compiler: $390/38.38 = 10.16X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.63, Optimized GFLOPS/s = 12.02

Using gcc: Baseline GLOPS/s = 0.66, Optimized GLOPS/s = 6.47

Optimization Strategies:

Use the -O3 flag for compilation, Vectorize the for loop using `#pragma omp simd`, Parallelize using 4 threads by `#pragma omp parallel for simd`.

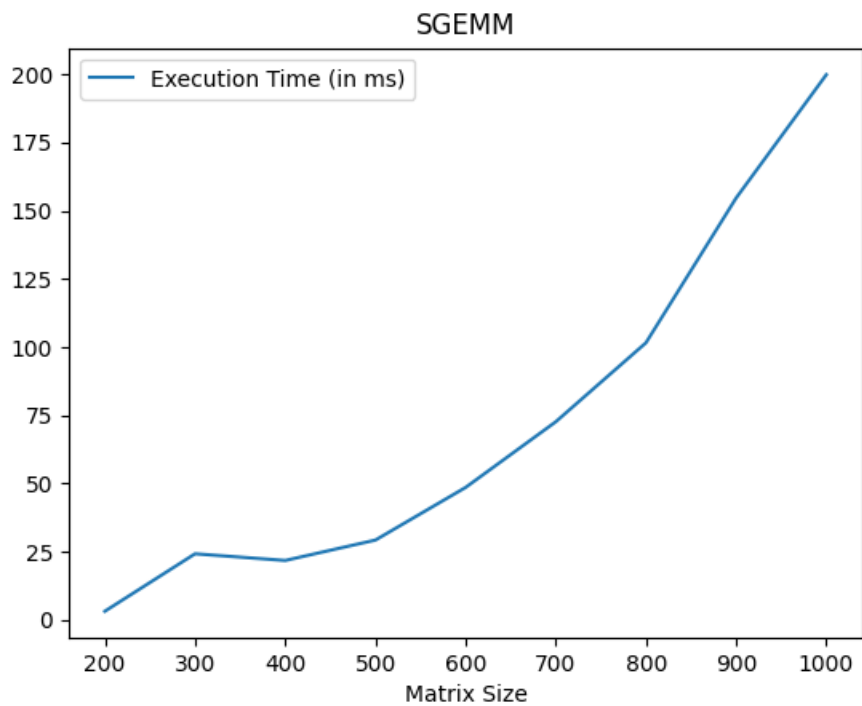
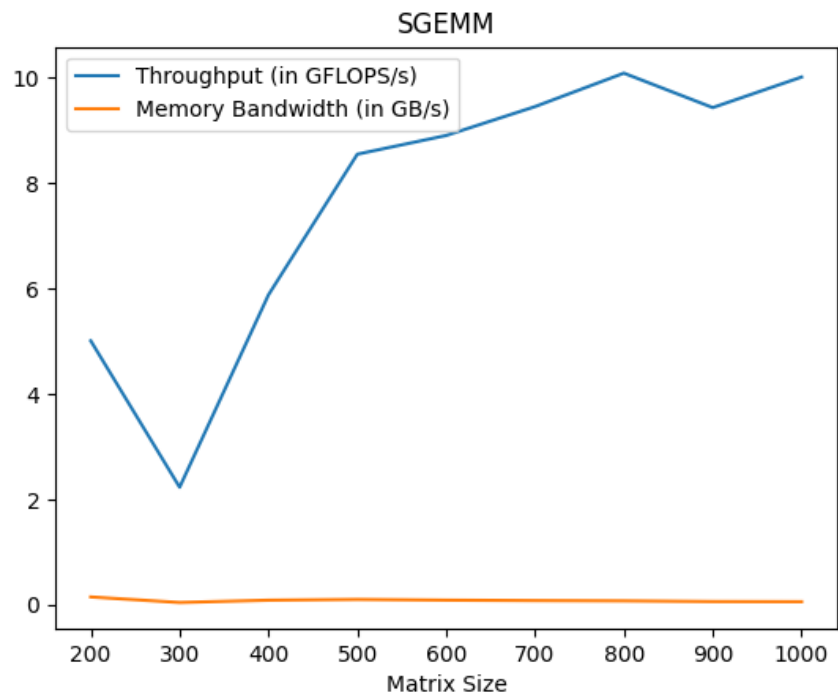
The best time is given by all these optimizations combined. The parallelization of outer for loop gives better result as compared to inner for loop. The cblas_dot routine is also vectorized using `#pragma omp simd`.

Memory Bandwidth Achieved:

Using icc compiler: 0.14 GB/s

Using gcc compiler: 0.047 GB/s

The program is memory bound.



DOUBLE PRECISION ANALYSIS

Operational Intensity:

Number of floating point operations performed:

Considering all dimensions as N , we need to perform N multiplications and $N-1$ additions for each element of AB . There are N^2 elements in total, therefore, $N^2(2N - 1)$ operations. Then N^2 operations for multiplying α with AB and N^2 operations for multiplying β with C . At last, we need N^2 operations to add these numbers.

$$\text{Total no of FLOPS} = N^2(2N - 1) + N^2 + N^2 + N^2 = 2N^3 + 2N^2$$

No of Memory Access:

$$3N^2 (N^2 \text{ for each matrix}) * 8 \text{ Bytes} = 24 N^2$$

$$\text{Operational Intensity} = (2N^3 + 2N^2)/24N^2 = N/12$$

Execution Times:

$$N = 500$$


Using gcc -O3 :

- v. Execution Time: 556 ms
- w. GFLOPS: 0.45 GFLOPS/s
- x. Memory Bandwidth: 0.01 GB/s

Using icc -O3 :

- v. Execution Time: 141.71 ms
- w. GFLOPS: 1.76 GFLOPS/s
- x. Memory Bandwidth: 0.042 GB/s

Baseline & Best Execution Time:



Baseline Execution Time is 681 ms (using gcc) and 145 ms (using icc).
(without using -O3 option).

Best Execution time is 39 ms (using icc) and 76 ms (using gcc).

Speedup:

In icc compiler: $145/39 = 3.71X$ speedup.

In gcc compiler: $681/76 = 8.96X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.63, Optimized GFLOPS/s = 6.41

Using gcc: Baseline GLOPS/s = 0.66, Optimized GLOPS/s = 3.26

Optimization Strategies:

Use the -O3 flag for compilation, Vectorize the for loop using `#pragma omp simd`, Parallelize using 4 threads by `#pragma omp parallel for simd`.

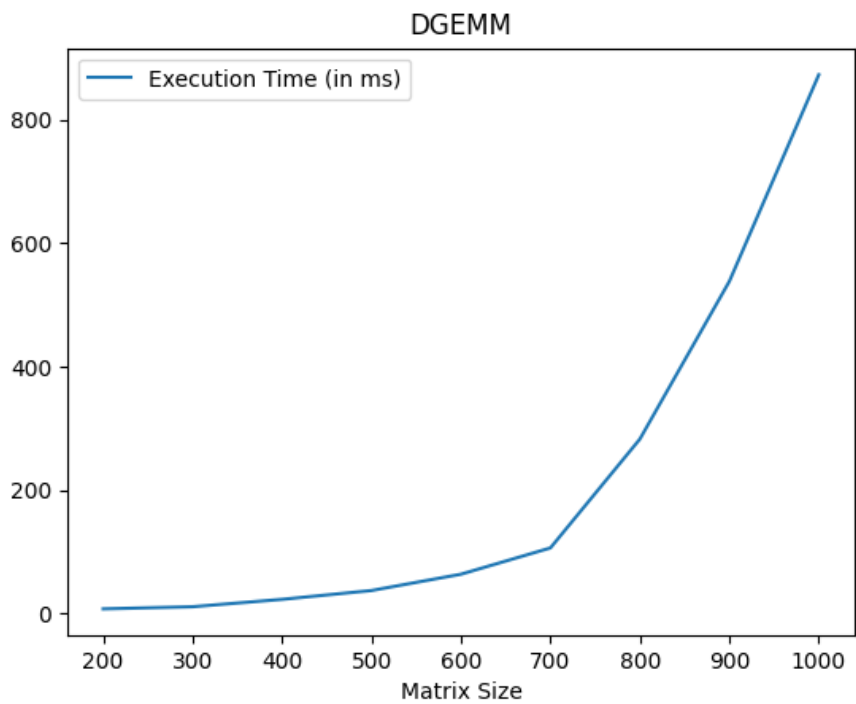
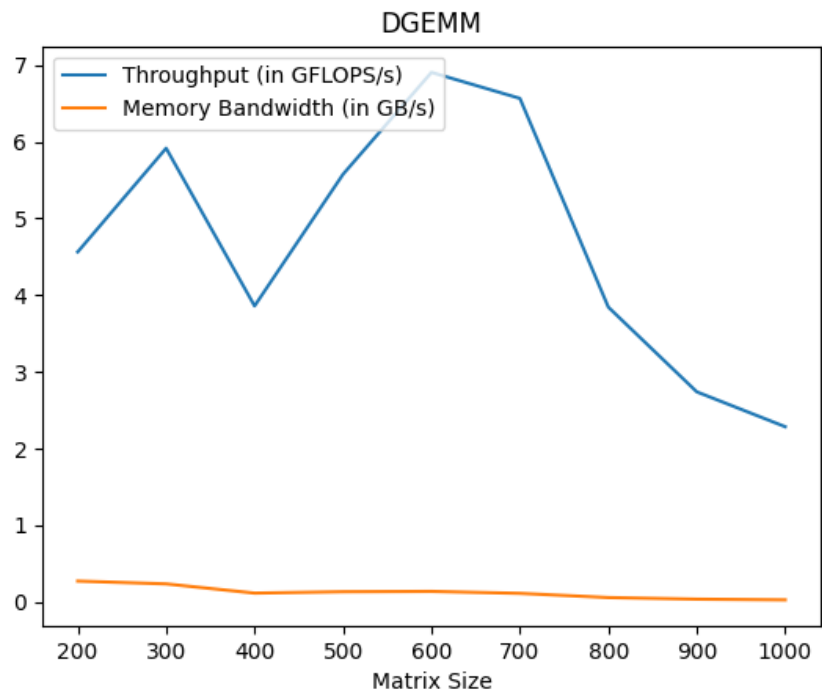
The best time is given by all these optimizations combined. The parallelization of outer for loop gives better result as compared to inner for loop. The cblas_dot routine is also vectorized using `#pragma omp simd`.

Memory Bandwidth Achieved:

Using icc compiler: 0.05 GB/s

Using gcc compiler: 0.05 GB/s

The program is memory bound.



Outer Product Formulation (i-j-p):

```
for (int p = 0; p < N; ++p)
{
    cblas_sger(CblasRowMajor, M, K, 1, A + p, lda, B + p *
ldb, 1, temp, ldc);
}
```

Cblas_sger:

```
for(int i = 0; i < M; i++)
{
    for(int j = 0; j < K; j++)
    {
        A[i*lda + j] += X[i*incX] * Y[j*incY];
    }
}
```

We express the product of two matrices as the outer product of columns and rows of the two matrices. Each outer product produces a $M \times K$ matrix and there will be N such matrices. Adding them will give the resultant matrix AB . The advantage of this method is that we can discard the row and column from memory as soon as its outer product is finished. Thus, we can easily parallelize this method on different computers to get high performance.

Operational Intensity: $N/6$

Execution Times:

$N = 800$

Using gcc -O3 :

y. Execution Time: 101.76 ms

- z. GFLOPS: 10.06 GFLOPS/s
- aa.Memory Bandwidth: 0.075 GB/s

Using icc -O3 :

- y. Execution Time: 395.92 ms
- z. GFLOPS: 2.58 GFLOPS/s
- aa.Memory Bandwidth: 0.04 GB/s

Baseline & Best Execution Time:

Baseline Execution Time is 478 ms (using icc compiler) and 403 ms (using gcc compiler). (without using -O3).

On using -O3 option, Execution time drops to 392 ms using icc and to 100 ms using gcc.

On Vectorization of cblas_sger inner for loop, execution time drops to 300.34ms (using icc). On Vectorization and parallelization of outer for loop, the execution time remains almost same.

On Vectorization and parallelization of outer for loop using gcc compiler, the execution time significantly drops to 24.90 ms and GFLOPS/s rise to 41.13.

On doing the same optimization for icc, the execution time drops to 101.42ms.

On Vectorizing and multithreading the main for loop of outer product formulation, the gain in performance is very insignificant. This is due to the use of #pragma omp critical to avoid race conditions between threads trying to manipulate the same data. The critical section compensates for the increase in speed achieved by parallelization and hence there is no significant improvement over the unparallelled version.

Therefore, the best execution time is using gcc compiler = 24.90 ms. Using icc compiler = 101.42ms.

Speedup:

In icc compiler: $478/101.42 = 4.71X$ speedup.

In gcc compiler: $478/24.90 = 19.19X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 0.58, Optimized GFLOPS/s = 10.06

Using gcc: Baseline GLOPS/s = 0.56, Optimized GLOPS/s = 41.13

Optimization Strategies:

- a. Use the -O3 flag for compilation
- b. Vectorize and Parallelize the outer for loop of cblas_sger using
#pragma omp parallel for simd.

Observation: On parallelizing and vectorizing the for loop of outer product formulation, we don't get the optimized performance. This is due to the use of #pragma omp critical to avoid race conditions between multiple threads trying to manipulate the same value. The critical section is used to sum all the N matrices generated by the threads individually. The critical section forces sequential processing and there is a significant cost associated with entering and exiting the critical section. This has a great effect on the performance of the program and hence the execution time increases.

Memory Bandwidth Achieved:

Using icc compiler: 0.05 GB/s

Using gcc compiler: 0.295 GB/s

The program is memory bound.

Stencil

```
for(int ii = 0; ii < k; ii++)
{
    for(int jj = 0; jj < k; jj++)
    {
        Y[(i - 1) * imgcols + (j - 1)] += S[ii*k + jj] *
        padded_X[(i - 1 + ii) * (imgcols + 2) + (j - 1 + jj)];
    }
}
```

We apply the K X K Stencil on the image with padding using two for loops. This is repeated for every pixel on the input image and the result is stored in output pixel.

Operational Intensity:

Number of floating point operations performed:

Consider the Source Image size of M X N, stencil size of K X K. To generate each output pixel, we need to perform K^2 multiplications and $K^2 - 1$ additions.


Therefore, total FLOPS = $MN(2K^2 - 1) = MN*2K^2$ (Approximately)

No of Memory Access:

To generate each pixel, We need K^2 elements from the source image matrix and K^2 elements from the stencil.

Therefore, total no of bytes accessed = $MN*2K^2 * 4$ bytes each

Operational Intensity = $MN*2K^2 / MN*2K^2*4 = \frac{1}{4} = 0.25$



Execution Times:

For HD Image: Using Stencil size of 31X31

Using gcc -O3 :

- a. Execution Time: 2282.58 ms
- b. GFLOPS: 1.74 GFLOPS/s
- c. Memory Bandwidth: 7.10 GB/s

Using icc -O3 :

- a. Execution Time: 1215.69 ms
- b. GFLOPS: 3.27 GFLOPS/s
- c. Memory Bandwidth: 13.11 GB/s

For UHD Image: Using Stencil Size of 31X31

Using gcc -O3 :


- a. Execution Time: 8743 ms
- b. GFLOPS: 1.82 GFLOPS/s
- c. Memory Bandwidth: 7.29 GB/s

Using icc -O3 :

- a. Execution Time: 4821 ms
- b. GFLOPS: 3.30 GFLOPS/s
- c. Memory Bandwidth: 13.22 GB/s

Baseline & Best Execution Time:

Baseline Execution Time is 1223 ms (using icc compiler) and 8244 ms (using gcc compiler). (without using -O3 option).



On using -O3 option, execution time remains constant for icc and drops to 2300 ms for gcc.

On Vectorizing the Stencil loop, execution time drops to 635 ms for icc and remains almost same for gcc.

On parallelizing the for loop (j), execution time drops to 80 ms for icc and 325 ms for gcc.

Therefore, the best execution time is 80 ms using icc.

Speedup:

In icc compiler: $1223/80 = 15.18X$ speedup.

In gcc compiler: $8244/325 = 25.32X$ speedup.

Baseline & Optimized GFLOPS/s:

Using icc: Baseline GFLOPS/s = 3.25, Optimized GFLOPS/s = 50.03

Using gcc: Baseline GLOPS/s = 0.46, Optimized GLOPS/s = 12.02

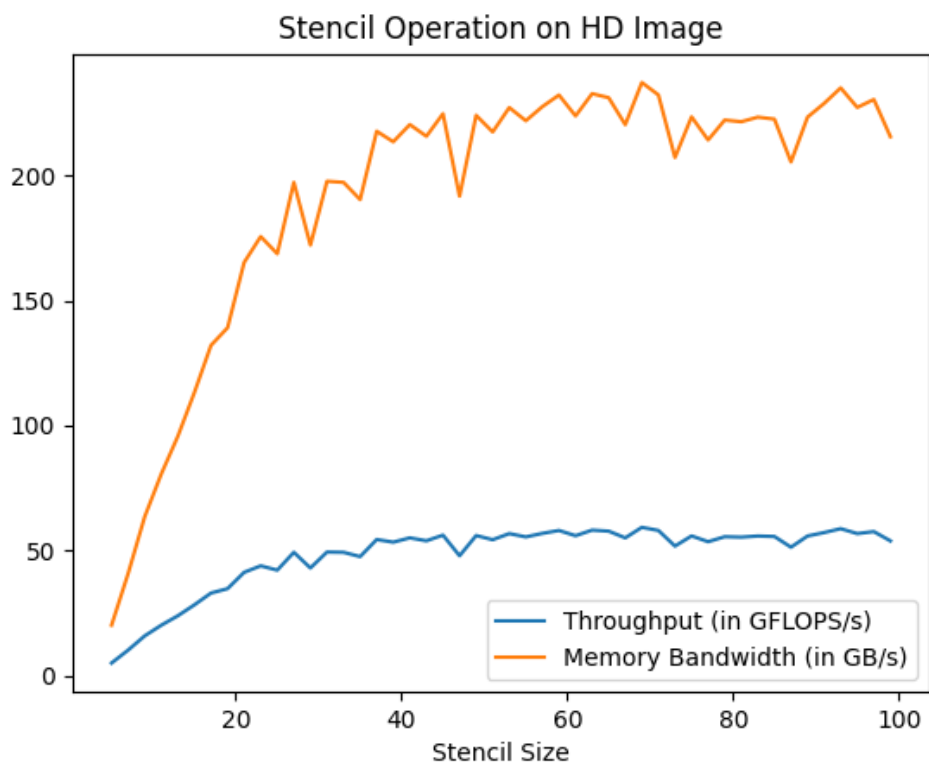
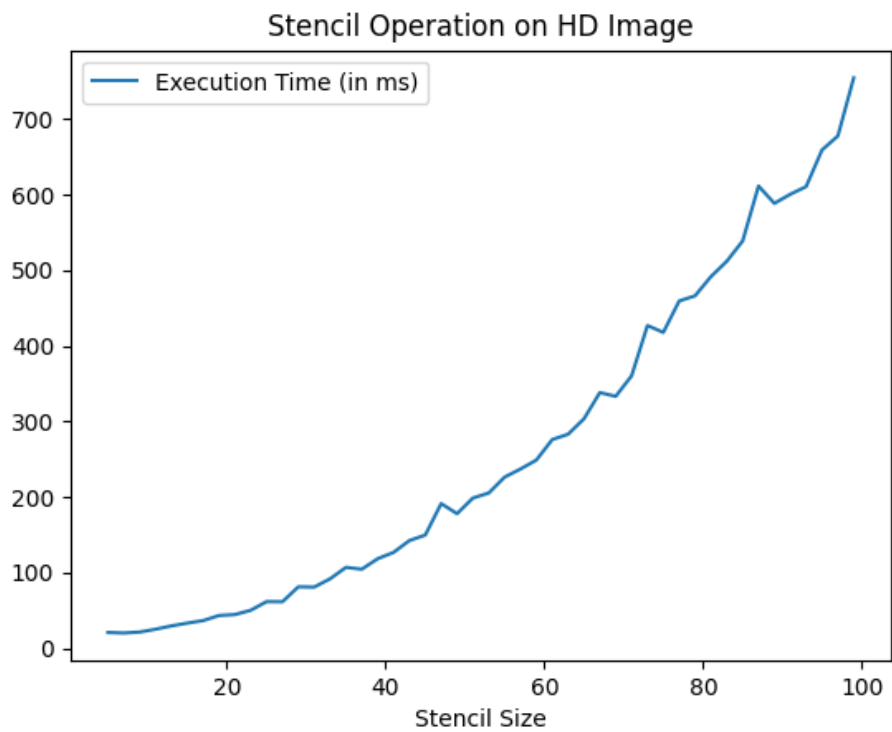
Optimization Strategies:

- a. Use the -O3 flag for compilation
- b. Vectorize the innermost loop (jj) of stencil program
- c. Vectorize and Parallelize the inner for loop (j) of stencil program

The program is memory bound as the peak flops for 8 threads is 204 GFLOPS/s and we are able to achieve only 53 GFLOPS/s.

Graphs:

HD Image:



UHD Image:

