# SPP Assignment

Vidit Jain (2020101134)

# KYC

## CPU Specifications

**CPU Model -** i7-9750H

**Frequency Range -**

**Base -** 2.60Ghz

**Turbo -** 4.50Ghz

**Core Count -** 6

**Hyperthreading -** Available

**SIMD ISA -** AVX2 & FMA (256-bit)

**Main memory bandwidth -** 41.8 GB/s

### Estimating FLOPS

$$
\begin{aligned}
\text{FLOPS for one core} &= \frac{\text{Floating point operations}}{\text{cycle}} \times \frac{\text{cycles}}{\text{second}} \\
&= \frac{32}{1} \times \frac{4.5\text{GHz}}{1} \\
&= 144 \text{ GFLOPS}
\end{aligned}
$$

$$
\begin{aligned}
\text{FLOPS per processor} &= \text{FLOPS for one core} \times \frac{\text{cores}}{\text{socket}} \\
&= 144 \times 6 \\
&= 864 \text{ GFLOPS}
\end{aligned}
$$

### Benchmarking

**Whetstone Benchmark -**

With `gcc` - 3125 MIPS

With `icc` - 62500 MIPS

This benchmark is suboptimal for measuring computing performance due to the lack of standardization of the instructions used for testing, and the accuracy for testing as it counts in seconds, leading to a large amount of error.

## Own CPU Benchmark

To benchmark my own cpu, I had a couple things in mind that I could optimize to ensure that my CPU usage is maximized while making sure that it isn't being bottlenecked by anything else like memory.

The first thing that I thought of was making use of FMA instructions, as they can be quite fast.

Secondly, I thought of making use of SIMD functions to maximize vectorization.

Thirdly, I didn't want memory to be a constraint, so I kept the vector size low, and computed the same instruction over the same array multiple times so that it remains in cache, and the CPU wouldn't have to wait as long to fetch the data from RAM.

First, I started off with writing a single core benchmark with no parallelization, and I got around 16 GFLOPS. Then I thought of tinkering with the vector size, and upon trying different values, I found `128` to be the optimal size for performance for single core, and it got me 36 GFLOPS.

When I switched to multi-core, I basically used the same code that I had made for the single core benchmark, but ran multiple instances of it. I tuned the vector size again, and found 64 to be optimal, which gave me approximately 182 GFLOPS.

Comparing it to the theoretical GFLOPS achievable

| Single/Multi | Actual | Theoretical |
|---|---|---|
| Single | 36 | 144 |
| Multi | 182 | 864 |

## Memory

**Main memory size** - 16GB

**Memory Type -** DDR4

Maximum main memory bandwidth - 41.8 GB/s

**Stream Benchmark -**

```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           8292.7     0.019946     0.019294     0.020755
Scale:          9093.0     0.017771     0.017596     0.018008
Add:           11366.3     0.021568     0.021115     0.021945
Triad:         10943.9     0.022776     0.021930     0.024393
-------------------------------------------------------------
```

Hence I get a best case of 11 GB/s in memory bandwidth when making use of the stream benchmark.

**Own Memory Benchmark -**

I couldn’t think much of what could I do to increase memory bandwidth, hence I went for a simple approach, where I declared two large arrays and added them. This gave me a performance of 22.389 GB/s. The theoretical maximum is 41.8 GB/s

## Storage Size

I have a storage device (HDD) that is of 1TB capacity. It has 6 Gb/s speed (note Gb, not GB)

I have another storage device (NVME SSD) that is of 256GB capacity, and has 2GB/s speed.

## ABACUS and ADA supercomputers

- ABACUS: 14 TFLOPS
- ADA: 70.66 TFLOPS

# BLAS Problems

First off, I decided to install `icc` so i could compare the performance from both compilers.

Then, I brainstormed some ideas that may help me in improving the speed of my implementation. Here were some of the initial ideas I had before starting out the assignment

1. Making code more cache friendly.
2. Multithreading the function to make it faster as the iterations were dependent of each other
3. Opening the assembly code to get a better understanding of what may be happening.
4. Making use of SIMD functions to speedup
5. Tinkering with the FMA instruction (fused add and mulitply) and see if it provides a speedup.
6. Doing redundant storage, such as storing the transpose of a matrix in case we access it column wise.

# Creating a Benchmark

To even compare the performance of various implementations of BLAS, the first thing I'll need is a way to quantify the performance of a certain implementation, which is done using benchmarking.

In the benchmark, we'll need 2 main things,

1. The performance of the computation in terms of GFLOPS
2. The performance of the computation in terms of the memory bandwidth used

As there was a very simple implementation of saxpy already done by the prof, I decided to start off with it to get a very simple benchmarking tool up and running.

As we have to test multiple functions, manually inserting tick, tock before and after a function call seemed laborious and something that could be improved / abstracted out.

Remembering how I made use of pthreads during my Operating Systems course for concurrency, I took inspiration from how they made functions multi-threaded by passing a function pointer and the arguments through a struct, which is passed to the function as a void pointer.

Another issue is finding out how many floats were created / used , as the calculating that number could be different for different BLAS levels. Hence, I thought of calculating that number within the function itself. It would add a bit to the time taken by the functions, but knowing that this calculation involves only 3-4 operations, and the magnitude of the actual BLAS level operations would be much much higher, it would make the 3-4 operations negligible in calculations.

However, I came to the realization that we're supposed to adhere to the function signatures provided in `cblas.h`. Therefore, I went for a simpler route, giving a wrapper function for each BLAS operation, and benchmarking the function in it.

# BLAS Level 1

## Working with SSCAL, DSCAL

$$X = \alpha X$$

SSCAL is the simplest operation in BLAS, as it just involves multiplying a vector with a scalar.

### Operational Intensity

$$O.I = \frac{N}{4 * N} = 0.25$$

Where $N$ is the size of the vector.

If we're using double precision, the $O.I$ would be half that, at $0.125$.

#### Simple Compilation

I first started off with the simplest possible code, no optimizations and no special flags when compiling the code. The results I got were

| Compiler | N | Memory Bandwidth | GFLOPS |
|---|---|---|---|
| `gcc` | 1000000 | 1.804 | 0.451 |
| | 10000000 | 1.765 | 0.441 |
| | 100000000 | 1.737 | 0.434 |
| `icc` | 1000000 | 6.231 | 1.558 |
| | 10000000 | 7.107 | 1.777 |
| | 100000000 | 6.985 | 1.746 |

Here, I see that the performance is quite low which can be explained with the help of the operational intensity. There is a clear speed up when using the `icc` compiler.

## `-O3` compilation

For $N = 1e8$

`gcc -O3` execution time - 28.644 ms

`icc -O3` execution time - 28.858 ms

| Compiler | N | Memory Bandwidth | GFLOPS |
|---|---|---|---|
| `gcc` | 1000000 | 14.545 | 3.636 |
| | 10000000 | 14.065 | 3.516 |
| | 100000000 | 13.302 | 3.326 |
| `icc` | 1000000 | 7.561 | 1.890 |
| | 10000000 | 7.537 | 1.884 |
| | 100000000 | 7.157 | 1.789 |

We can see a huge spike in the speed of the program when compiled with `gcc -O3`, 8-9x faster than without it. However, we don't see much of a difference in `icc`, where the performance is only slightly better than compiling with no flags.

## Baseline vs Best execution

**Baseline -** 28.821 ms

**Best -** 28.644 ms

## GFLOPS Baseline vs Best

**Baseline -** 3.470 GFLOPS

**Best -** 3.491 GFLOPS

## Making Optimizations

### The `-ffast-math` flag

I tried compiling with the `-ffast-math` flag as well, however it didn't provide any speed up with both `gcc` and `icc` . This does make sense, as using `-ffast-math` allows the compiler to assume floating point operations are associative, however there is no operation in this program that would benefit from it.

### Using parallelization

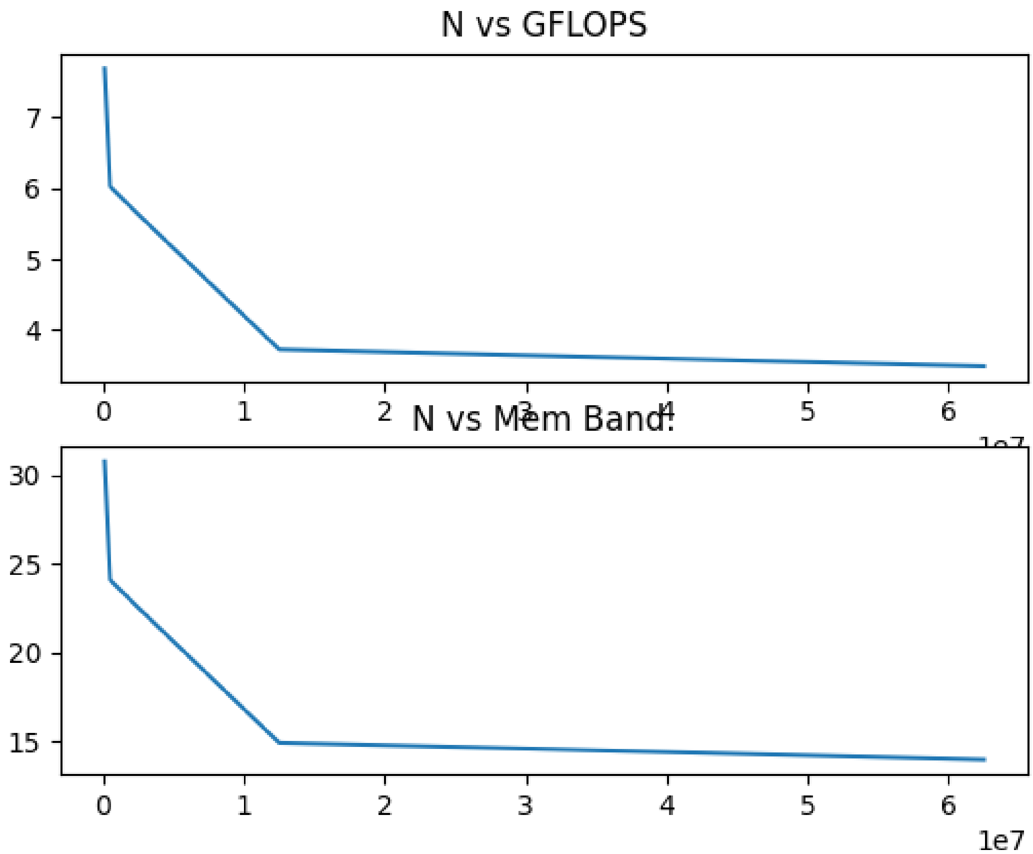
When you make use of `#pragma omp parallel for`, we see a considerable speedup, going from $0.45$ GFLOPS to $2$ GFLOPS, but no noticeable speedup is seen when you use both `-O3` and the parallelization pragmas, which is quite intriguing. This could be because the memory is acting as a bottleneck, hence speeding up the computing operations doesn't provide any benefit.

### Using `simd`

Surprisingly, when I used `simd`, it turns out that I get no speed up in the computing power. I wasn't able to figure out why though, and double checked the flags that had to be used to make use of the `simd` pragma from OpenMP. My best guess is that the compiler is able to analyze that `simd` can be used for the loop, and was already making this optimization.

### Using intrinsics

When using the pragma didn't work, I decided to go a bit more low level, and decided to implement the vectorization myself. I found out how to make use of `_mm` operations and implemented them for both floats and doubles. Unfortunately, I didn't see any speedup here either. I thought that I could possibly get a speedup if I make use of aligned loads and stores, but using those intrinsics caused the benchmark to have a segmentation fault, hence I couldn't benchmark that.




### Memory Bandwidth Achieved

**Baseline -** 13.879 GB/s

**Best -** 13.965 GB/s

### BLIS Performance

The BLIS implementation gave me these results for $N = 1e8$

`gcc -O3` - 3.470 GFLOPS

`icc -O3` - 3.476 GFLOPS

There isn't much off a difference between my implementation's performance and BLIS as there isn't much that can't be optimized by you that isn't already optimized by the compiler.

## Working with SDOT, DDOT

$$\text{dot} = X^T Y$$

### Operational Intensity

There are $N$ multiplication operations done, and $N - 1$ additions. There are $2 \times N$ floats/doubles. Hence, when using single precision, the operational intensity will be given as.

$$O.I = \frac{2 \times N}{8 \times N} = 0.25$$

And $O.I = 0.125$ when using double precision.

### `-O3` compilation

Taking $N = 1e8$

`gcc -O3` execution time - 99.625 ms

`icc -O3` execution time - 58.832 ms

### Baseline vs Best execution

**Baseline -** 229.574 ms

**Best -** 43.446 ms

### Speedup

$$\text{Speed up} = \frac{\text{Baseline}}{\text{Best}} = \frac{229.574}{43.446} = 7.979$$

### GFLOPS baseline vs base

**Baseline -** 0.871 GFLOPS

**Best -** 7.193 GFLOPS

### Optimization Strategies

Parallelization is unsafe in this case as each operation is adding to one variable, hence it would lead to a race condition.

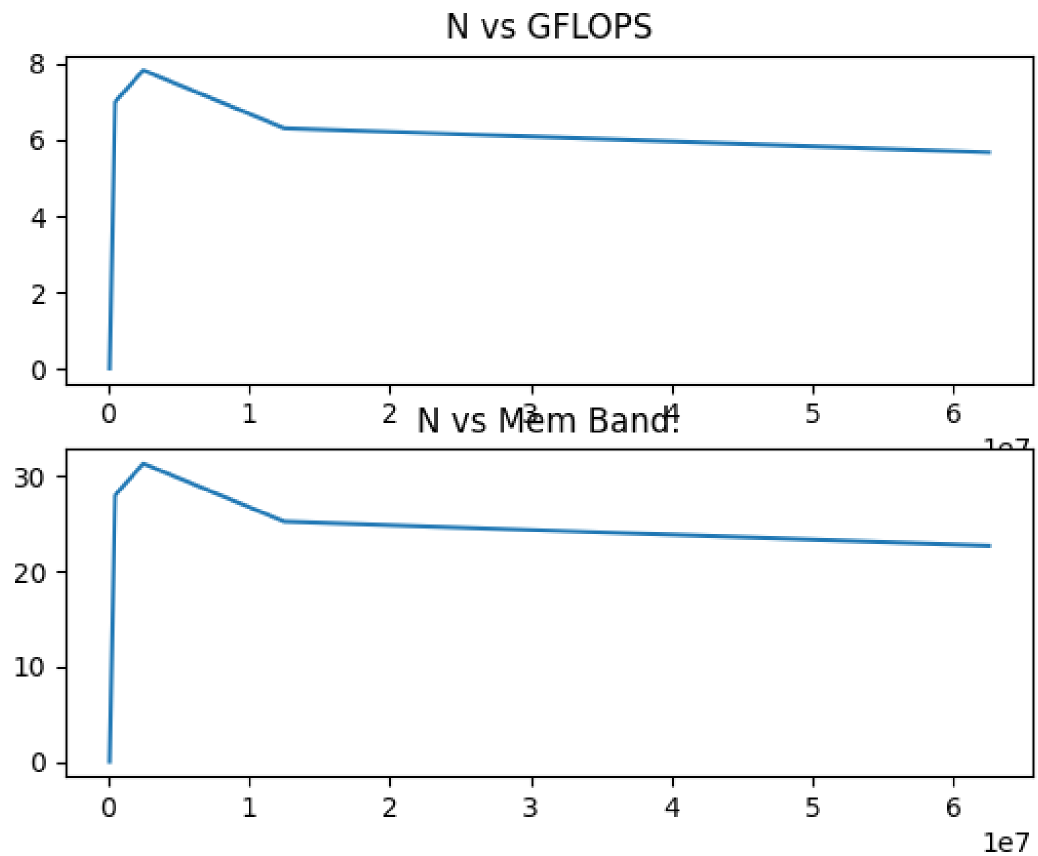
However, if you make use of the `reduction` argument in the pragma, you'll be able to add to it safely. Hence, making use of the parallelizing pragma, I ended up getting around $7.2$ GFLOPS.

The `simd` pragma provided no benefit to performance, and `-ffast-math` provided no speedup as well.

**Instrinsics give a speed up, but lose accuracy**

However, when I tried to make use of intrinsics here, I ended up with a very significant increase in GFLOPS, at approx $6.1$ GFLOPS. However, when I compared the answers I got, they were accurate for smaller values of $N$, but deviated significantly for larger values. This is most likely because of the fact that floating point operations aren't associative, hence the order in which I was performing the operations was different when compared to the standard way of computing the dot product. Hence, I did not record this speed as my best execution due to the inaccuracies in the answer.

Hence, in this case parallelization with `-O3` ends up giving me the best optimization




### Memory Bandwidth Achieved

**Baseline -** 3.485 GB/s

**Best -** 28.787 GB/s

### BLIS comparison

Taking $N = 1e8$, the performance I get from BLIS is

`gcc -O3` - 6.334 GFLOPS

`icc -O3` - 7.219 GFLOPS

## Working with SAXPY, DAXPY

$$Y = \alpha X + Y$$

### Operational Intensity

There are two operations done for every element in the vector, multiplying $X$ by $\alpha$ and adding it to $Y$. There are also $2 * N$ floats/doubles, hence when using single precision, the operational intensity will be given as.

$$O.I = \frac{2 \times N}{8 \times N} = 0.25$$

And $O.I = 0.125$ when using double precision.

## `-03` compilation

Taking $N = 1e8$

`gcc -03` execution time - 44.815 ms

`icc -03` execution time - 73.349 ms

| Compiler | N | Memory Bandwidth | GFLOPS |
|---|---|---|---|
| `gcc` | 1000000 | 27.875 | 6.969 |
| | 10000000 | 18.311 | 4.578 |
| | 100000000 | 17.736 | 4.434 |
| `icc` | 1000000 | 12.012 | 3.003 |
| | 10000000 | 11.003 | 2.751 |
| | 100000000 | 10.553 | 2.638 |

### Baseline Performance

| Compiler | N | Memory Bandwidth | GFLOPS |
|---|---|---|---|
| `gcc` | 1000000 | 3.366 | 0.841 |
| | 10000000 | 3.318 | 0.829 |
| | 100000000 | 3.368 | 0.842 |
| `icc` | 1000000 | 13.699 | 3.425 |
| | 10000000 | 12.195 | 3.049 |
| | 100000000 | 11.790 | 2.947 |

## Baseline vs Best time execution

**Baseline -** 235.868 ms

**Best -** 44.815 ms

## Speedup

$$\text{Speed up} = \frac{\text{Baseline}}{\text{Best}} = \frac{235.868}{44.815} = 5.26$$

Hence we get a $5.26$ times speedup

## GFLOPS Baseline vs Best

**Baseline -** 0.842 GFLOPS

**Best -** 4.434 GFLOPS

## Making optimizations

### Trying fma

FMA could possibly speed up the execution as it reduces the operations that have to be performed. I found out that the `fma` function actually slows things down by a bit, as it takes the `-O3` performance from 4.5 GFLOPS to around 3.4-3.5. The `-ffast-math` flag also didn't provide any speedup.

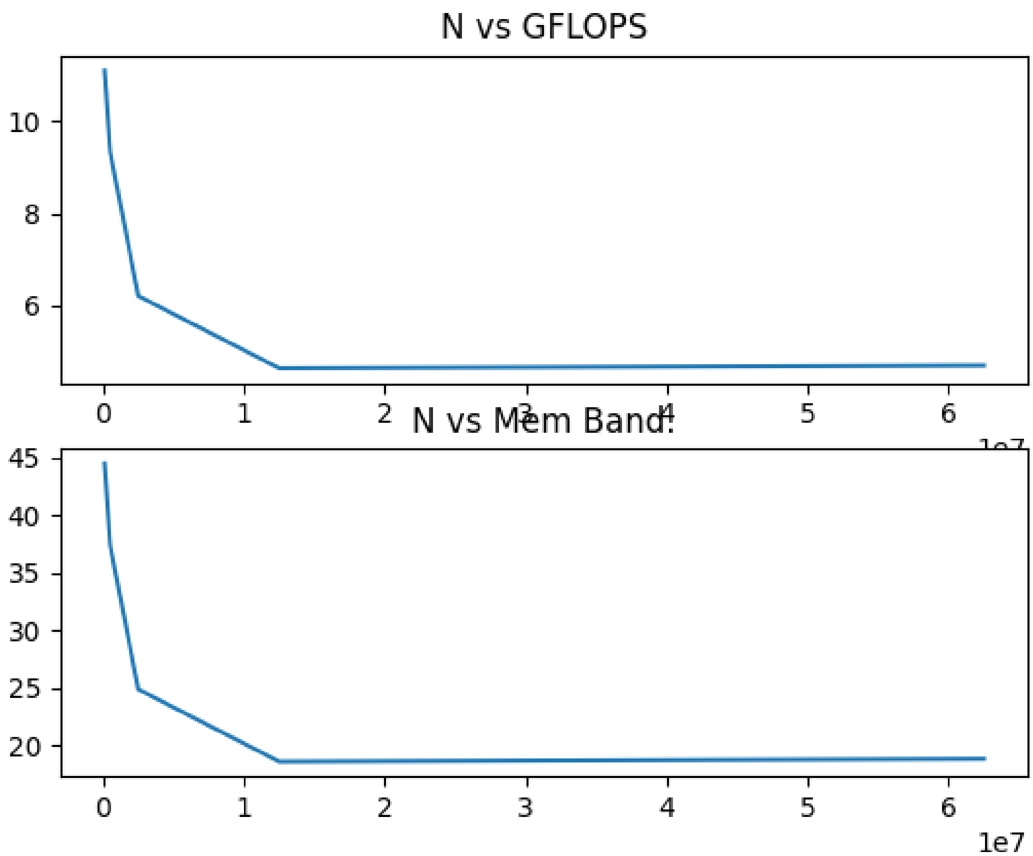
### Pragmas

I tried parallelizing the for loop by using pragma, but surprisingly, that slowed down the speed a little as well.

There was no difference when making use of the simd pragma as well.

### Instrinsics

I tried to make use of intrinsics as well and felt that making use of the fma intrinsic `_mm256_fmadd_ps` could possible speed things up, but saw no improvement again.




Therefore, I ended up with `-O3` as the best speedup for this BLAS problem.

### Memory bandwidth achieved

**Baseline -** 17.736 GB/s

**Best -** 3.368 GB/s

### BLIS Comparison

For $N = 1e8$, the performance I got from BLIS was

`gcc -O3` - 4.553 GFLOPS

`icc -O3` - 4.614 GFLOPS

# BLAS Level 2

$$Y = \alpha AX + \beta Y$$

### Operational Intensity

Assuming $N = M$ for simpler operational intensity calculations, we get

$$O.I = \frac{2N^2 + 3N}{4(N^2 + 2N)} \approx \frac{1}{2}$$

for floats, and $O.I = 0.25$ for doubles.

### `-O3` compilation

Taking $N = M = 25000$

`gcc -O3` execution time - 696.964 ms

`icc -O3` execution time - 411.079 ms

### Baseline vs Best Time Execution

**Baseline -** 706.110 ms

**Best -** 68.308 ms

### Speedup

$$\text{Speed up} = \frac{\text{Baseline}}{\text{Best}} = \frac{706.110}{68.308} = 10.337$$

### GFLOPS Base vs Best execution

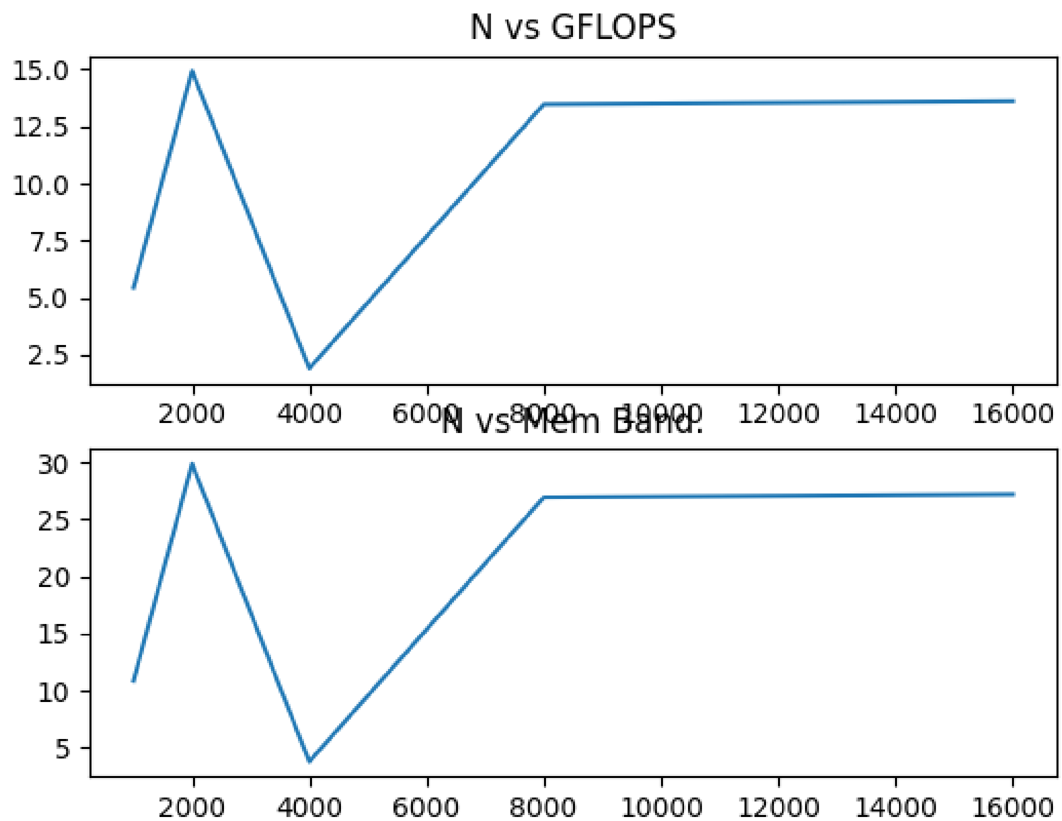
**Baseline -** 1.770 GFLOPS

**Best -** 18.301 GFLOPS

### Optimizations Made

I started off with doing the matrix vector multiplication without using sdot and any other functions to simplify it. After parallelizing it, I got some top speeds of $18.3$ GFLOPS, however this was quite inconsistent, as it varied from anywhere from $11$ GFLOPS to $18$ GFLOPS.

Then, I decided to try to make use of `sdot` and `sscal` to simplify the implementation, and I parallelized that as well. Interestingly enough, I saw that the performance actually reduced, and was maxing out at around $16.5$ GFLOPS, with much more significant dips to even $3$ GFLOPS at times. Hence, I went with the raw matrix vector multiplication code that was parallelized instead.




## Memory Bandwidth Achieved

**Baseline -** 3.541 GB/s

**Best -** 36.602 GB/s

## BLIS Comparison

For $N = M = 25000$, I got the performance

`gcc -O3` - 13.252 GFLOPS.

`icc -O3` - 15.513 GFLOPS

# BLAS Level 3

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

## Operational Intensity

Assuming $N = M$ for simpler operational intensity calculations, we get

$$O.I = \frac{2N^3 + 3N^2}{4 \times 3N^2} \approx \frac{N}{6}$$

for floats, and $N/12$ for doubles

## `-O3` compilation

For $N = M = K = 1000$

`gcc -O3` execution time - 167.095 ms

`icc -O3` execution time - 134.061 ms

## Baseline vs Best Time execution

**Baseline -** 147.681 ms

**Best -** 34.577 ms

## Speedup

$$\text{Speed up} = \frac{\text{Baseline}}{\text{Best}} = \frac{147.681}{34.577} = 4.271$$

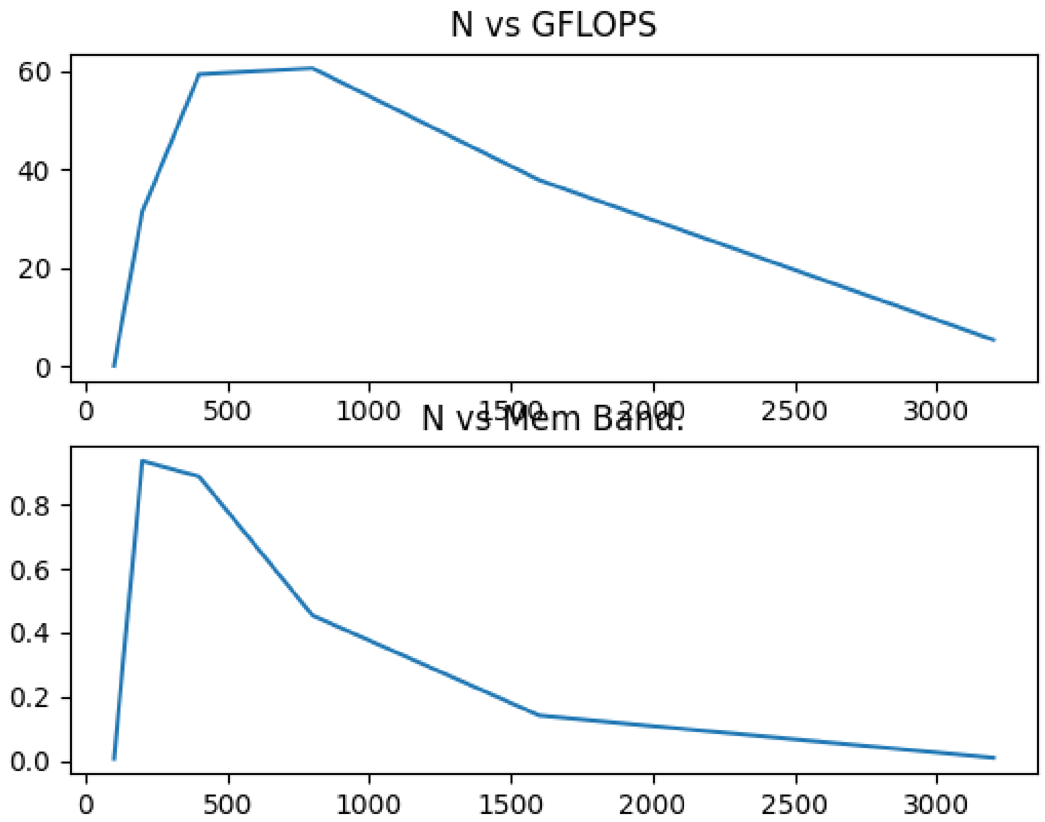
## GFLOPS Base vs Best execution

**Baseline -** 13.563 GFLOPS

**Best -** 57.929 GFLOPS

## Optimizations Made

The optimization in this level was pretty straightforward, as just applying parallelization gave me a significant improvement. I had to ensure that I was parallelizing the correct loop, or it could lead to a race condition which would lead to an incorrect answer.




### Memory Bandwidth Achieved

**Baseline -** 0.081 GB/s

**Best -** 0.347 GB/s

### BLIS Comparison

For $N = M = K = 1000$, BLIS gives me this performance

`gcc -O3` - 94.051 GFLOPS

`icc -O3` - 103.093 GFLOPS

# Stencil Computation

### Operational Intensity

$$O.I = \frac{\text{Pixel Count} \times k^2}{\text{Pixel Count} + k^2}$$

Benchmarking only UHD

### `-O3` compilation

Taking $k = 100$

`gcc -O3` - 18937.236 ms

`icc -O3` - 42552.105 ms

### Baseline vs Best Time Execution

**Baseline -** 128404.406 ms

**Best -** 16860.682 ms

### Speedup

$$\text{Speed up} = \frac{\text{Baseline}}{\text{Best}} = \frac{128404.236}{16860.682} = 7.615$$

### GFLOPS Base vs Best Execution

**Baseline -** 0.161 GFLOPS

**Best -** 1.230 GFLOPS

### Optimizations Made

I noticed that both the outer loops can be made parallel, as those two loops together are basically just calculating the value of a single pixel, hence can be done independently without any race conditions arising.

I used the pragma both for the first loop and the second loop, and it got me to my best performance.

### Memory Bandwidth Achieved

**Baseline -** < 0.001 GB/s

**Best -** < 0.001 GB/s