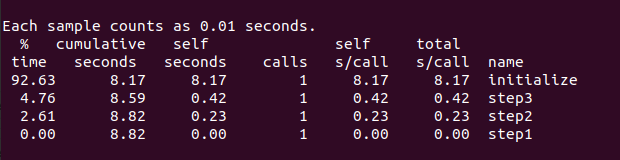
Coursework

# 1A)



Most computationally expensive

^

|

| Initialize -> 92.63%

| Step3 -> 4.76%

| Step2 -> 2.61%

| Step1 -> 0.00%

|

Least computationally expensive

The most computationally expensive are Initialize, Step3 and Step2.

# 1B)

### a)

Below are screenshots of the FLOPs values given. The average FLOPs value will be calculated based on these.

Fig.1

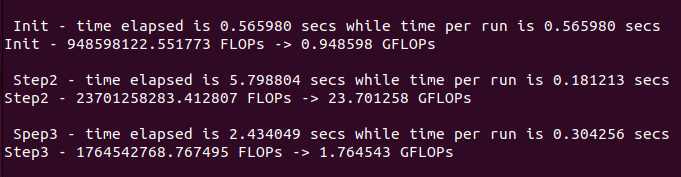


Fig.2

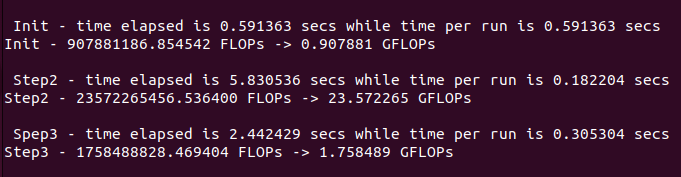
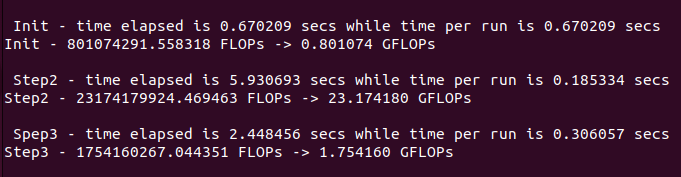


Fig.3



Most computationally expensive routines are Initialize, Step3 & Step2.

**Finding the average FLOPs for Initialization (to 2 decimal places):**

**Finding the average FLOPs for Step2 (to 2 decimal places):**

**Finding the average FLOPs for step 3 (to 2 decimal places):**

### b)

#define BILLION 1000000000

double flop, flops, timeper;

**For Initialization:**

start=omp\_get\_wtime();

for (i=0;i<1;i++){

initialize();

}

end=omp\_get\_wtime();

printf("\n Init - time elapsed is %f secs while time per run is %f secs\n",end-start, (end-start)/1);

flop = N + 2 \* N \* N;

timeper = (end-start)/1;

flops = flop/timeper;

printf("Init - %f FLOPs -> %f GFLOPs\n",flops, flops/BILLION);

**For Step2:**

start=omp\_get\_wtime();

for (i=0;i<32;i++){

step2();

}

end=omp\_get\_wtime();

printf("\n Step2 - time elapsed is %f secs while time per run is %f secs\n",end-start, (end-start)/32);

flop = 16.0f \* N \* N; // ((8\*8) \* N / 4) \* N

timeper = (end-start)/32;

flops = flop/timeper;

printf("Step2 - %f FLOPs -> %f GFLOPs\n",flops, flops/BILLION);

**For Step3:**

start=omp\_get\_wtime();

for (i=0;i<8;i++){

reduction=step3();

}

end=omp\_get\_wtime();

printf("\n Spep3 - time elapsed is %f secs while time per run is %f secs\n",end-start, (end-start)/8);

flop = 2 \* N \* N + 1 \* N / 8;

timeper = (end-start)/8;

flops = flop/timeper;

printf("Step3 - %f FLOPs -> %f GFLOPs\n",flops, flops/BILLION);

### c)

|  |  |  |
| --- | --- | --- |
|  | Windows Machine | Virtual Machine (running on windows machine) |
| CPU | Intel Core i7 8750H 2.20GHz (Turbo 4.10 GHz) | Intel Core i7 8750H 2.20GHz (Turbo 4.10 GHz) |
| Memory | DDR4 16GB 1200GHz | DDR4 8GB 1200GHz |
| OS | Windows 11 Home (64-bit) | Ubuntu 22.04 LTS (Jammy Jellyfish) (64-bit) |

# 1C)

**Theoretical peak FLOPs:**

# 1D)

Fig.4

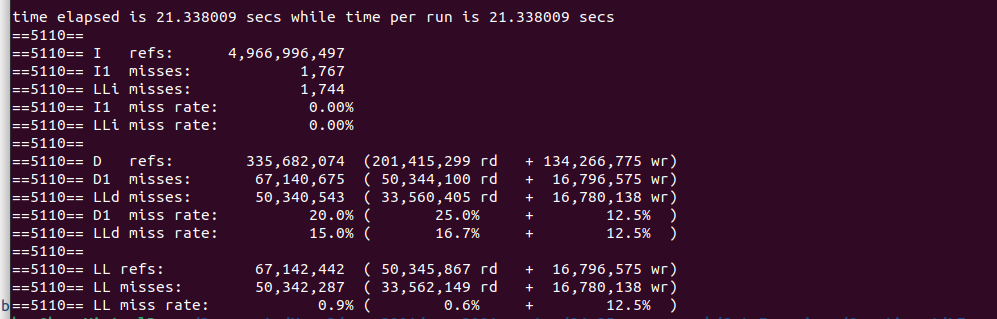


Fig.4 shows the Cachegrind output for the program with the LL misses being the number of L3 cache misses.

The Cachegrind output shows that 50,342,287 total L3 cache misses occurred, with 33,562,149 read misses and 16,780,138 write misses. The number of misses can be explained by looking at the memory and how it is accessed.

**Memory**

Given that N is large (at 16384) and the 2D array A has dimensions N \* N. This results in there being a total of 268,435,456 elements in A. Since each element is a float value taking up 4 bytes

The total size of A is:

This is much too big to fit inside the L3 cache (which is 9.0 MB for this machine), meaning that only a small part of A can be inside the cache at a given moment. The result of this is such that whenever a new part of A is accessed it will almost always not be in the L3 cache, giving a cache miss.

In contrast the 1D arrays X and Y can both easily fit within the cache. With each arrays size being:

Because of this array A will have the greatest contribution to the L3 cache misses.

**Estimating Cache Read Misses**

Array A is used in Step2 and Step3 and is accessed through row by row.

Given that a cache line is 64 bytes long and each float is 4 bytes, each cache line holds 16 float values.

Therefore, as there will be little if not any spatial locality significant cache thrashing will occur resulting in a L3 cache miss every 16 elements the are required from A. So, for each row of A accessed there is:

For the whole array there are:

For both step2 and step3 the total cache misses when accessing A would be:

The estimated total cache misses are close to the value of 33,562,149 given by Cachegrind.

**Estimating Cache Write Misses**

The biggest impact on write misses would be during the initialization of array A as it cannot all fit into the L3 cache. Therefore, like with the cache read misses a cache write miss will occur every 16 elements.

This is close to the value given by Cachegrind of 16,780,138.

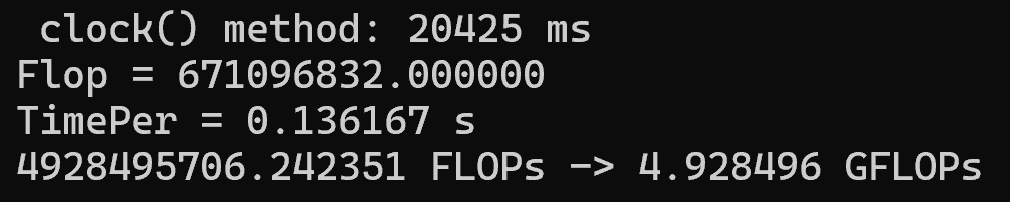
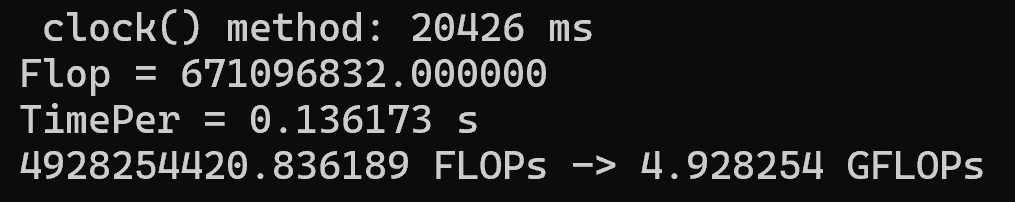
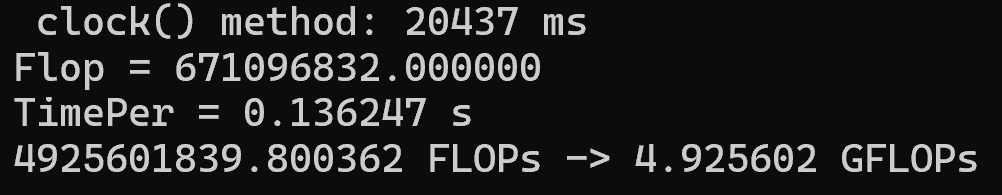
**Total Estimate**

Combining the read and write misses the estimated value is:

These estimates justify the large number of L3 cache misses occurring.

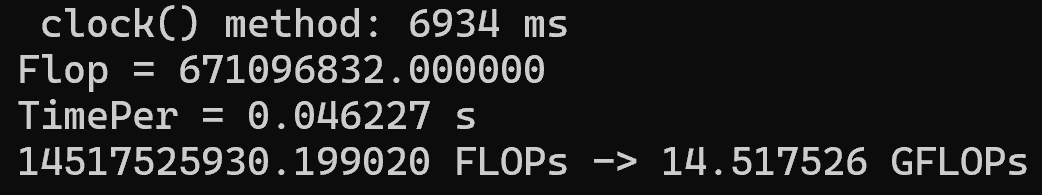
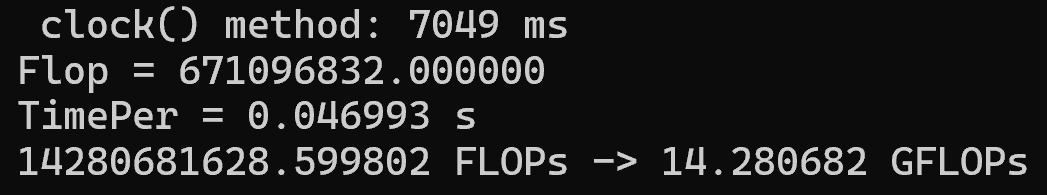
# Q2)

## Before Optimisation:



**Finding the average FLOPs before optimisation (to 2 decimal places):**

## After Optimisation:



**Finding the average FLOPs after optimisation (to 2 decimal places):**