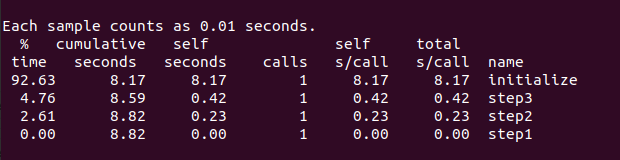
Coursework

# 1A)



Most computationally expensive

^

|

| Initialize -> 92.63%

| Step3 -> 4.76%

| Step2 -> 2.61%

| Step1 -> 0.00%

|

Least computationally expensive

# 1B)

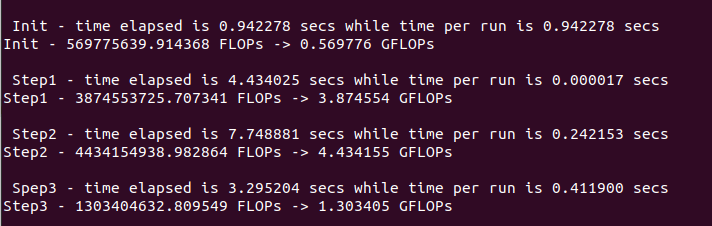
### a)

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. This means that we will ignore Step1.

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

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

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

### b)

#define BILLION 1000000000

**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);

**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 = ((8+8) \* N / 4) \* N;

timeper = (end-start)/32;

flops = flop/timeper;

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

**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);

### c)

CPU: Intel Core i7 8750H 2.20GHz (Turbo 4.10 GHz)

DDR: DDR4 16GB 1200GHz

OS: Ubuntu 22.04 LTS (Jammy Jellyfish) (64-bit)

## 1C)

**Theoretical peak FLOPs:**

Explain the results obtained in Step B.

## 1D)

Fig.4

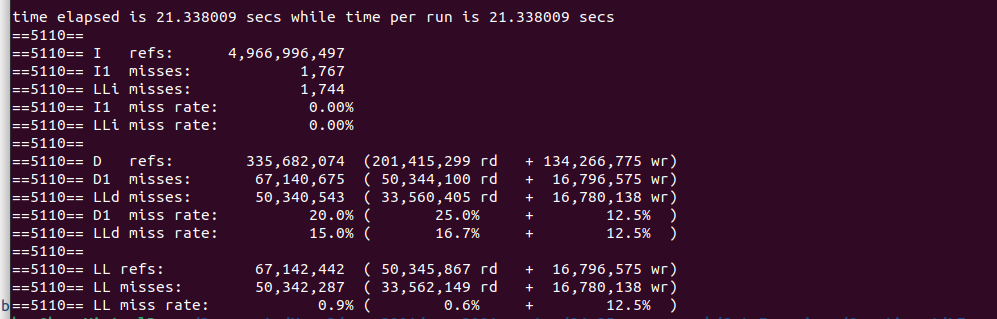


Fig.4 shows the Cachegrind output for the program running 1 time with the LL misses being the number of L3 cache misses.

Size of A[N][N]

N\*N = 268,435,456 floats

Each float is 4 bytes meaning the 2d array A takes up

N\*N\*4 bytes = 1,073,741,824 bytes

Roughly 1.07 GB

With the L3 Cache being significantly smaller (my device being 9.0 MB)

Since A is much larger than the L3 cache almost every access to new parts of A will result in a L3 cache miss as not all of A can fit within the L3 cache.

Access Patterns

Step 2 & step 3 access A[i][j]

Step 2 access A row by row (this means there isn’t any spacial locality past the vectorised 4)(no temporal as they arnt reused) (lack of locality means)

Step 3 has

THIS CAUSES CACHE THRASHING (repeatedly access data whose size is larger than the cache size, the data gets pushed out of the cache even if the intent is to use it)

Basicly N is large which make A[N][N] very large (16384 x 16384 = 268,435,456). That’s N^2 floats = 1,073,741,824 bytes = 1.07… Gb which def doesn’t fit into L3 cache. Basically almost every call to A will be a cache miss

Estimating cache misses for each access pattern

Step 2 – goes through each row of A four floats at a time (j+=4) meaning 1 cache miss occurs every 16 floats.

For 1 row: N/16 = 1024

For all rows: N/16 \* N = 16,777,216 cache misses

Step 3 – A is access over i & j. as it goes through each row its similar to step 2 and thus will also have 16,777,216 cache misses.

AS SHOWN BY THE CACHEGRIND 2 x 16,777,216 = 33,554,432 cache misses. THE CACHEGRIND SHOWS 33,562,149 read misses. THEY ARE BASICLY THE SAME NUMBER

For write misses step 3 is writing to y[i]