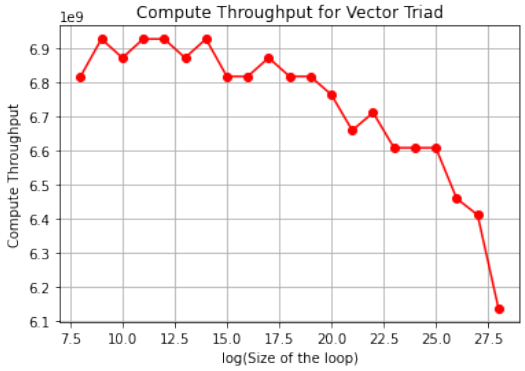
CS-301 High Performance Computing
Lab 1

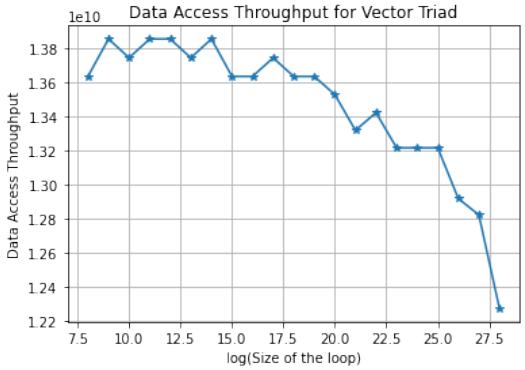
Pratvi Shah 201801407
Arkaprabha Banerjee 201801408

Experiments and the corresponding observations:

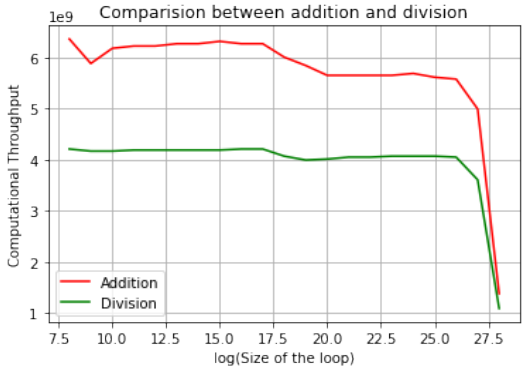
1. **Understanding your CPU architecture using *lscpu***
   - Architecture: **x86_64**
   - Byte Order: Little-endian
   - CPU(s): 4
   - Model Name: Intel(R) Core(TM) *i*5-4590 CPU 3.30GHz
   - L1d cache: 32K
   - L1i cache: 32K
   - L2 cache: 256K
   - L3 cache: 6144K
2. **Vector triad bench marking**
   - Computational throughput
     The trend is expected to be linearly decreasing if we think it in terms of ideal behaviour as rationally when the number of computations increase the throughput will decrease. But, when making such statement we do not consider the internal structure of the CPU. The graph can be explained by dividing it in the following parts:
     (a) Initial increasing trend: This is mainly due to the presence of pipelines and the efficient functioning that follows due to the use of the same.
     (b) Decreasing trend: As opposed to the pipeline efficiency the size of L1 cache also matters and as we noticed in Part 1 that the L1d cache for our architecture is of 32K hence, we notice a trough after x=15 which shows that as the fetching time increases when data is accessed from L2 cache the throughput decreases.
     (c) Drop followed by decreasing trend: Again reasoning along the same lines as the loop is ran for more iterations($\approx 2^{18}$) the L2 cache is full and hence, after this the time required to fetch the data increases thereby decreasing the throughput even more.
     (d) Final drop: The final drop is noted around 2^{23} wherein, the L3 cache is getting full and thus the sudden decrease in throughput is observed after that(due to fetching from Memory).



- Data access throughput
  Here, the trend is similar to that of compute throughput but as the number of data-access request is more than the number of operations being performed in the code hence, the throughput is scaled with respect to the same factor.



- When we compared the throughput of two operations namely, addition and division, we observed that the throughput of addition is greater than that of division. The trend for both the operations is similar as expected, but division can be considered as repeated subtraction which in itself is more expensive than addition, due to the process of conversion into 2's complement and then performing the regular addition. Hence, the low throughput of division as compared to addition.



3. **Profiling of the code (gprof)**

Code used:

```
#include<iostream>

using namespace std;

void func1(void )
{
    for(long int i=0;i<100000;i++);
    return;
}
void func2(void)
{
    for(long int i=0;i<200000;i++);
    return;
}

int main()
{
    for(int i=0;i<1000;i++)
    {
        func1();
        func2();
    }
    return 0;
}
```

Profiler output

As per the code we expect that the time taken by func2() is double that of func1() as the number of iterations are in the same ratio. The obtained output is in accordance with our expectation and we noticed that func1() took $\approx 39\%$ and func2() took $\approx 61\%$ of the total time.

```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.62                 main [1]
                0.37    0.00    1000/1000        func2() [2]
                0.24    0.00    1000/1000        func1() [3]
-----------------------------------------------
                0.37    0.00    1000/1000        main [1]
[2]     60.7    0.37    0.00    1000         func2() [2]
-----------------------------------------------
                0.24    0.00    1000/1000        main [1]
[3]     39.3    0.24    0.00    1000         func1() [3]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [17]
[10]     0.0    0.00    0.00       1         _GLOBAL__sub_I__Z5func1v [10]
                0.00    0.00       1/1           __static_initialization_and_destruction_0(int, int) [11]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I__Z5func1v [10]
[11]     0.0    0.00    0.00       1         __static_initialization_and_destruction_0(int, int) [11]
```