Values of some important parameters of our system are written below:

1. Architecture: x86_64
2. Byte Order: Little Endian
3. CPU(s): 4
4. Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
5. L1d cache: 32K
6. L1i cache: 32K
7. L2 cache: 256K
8. L3 cache: 6144K

# A: Vector Triad Benchmarking

1. We have used 'time.h' library to measure the run time. The resolution of time can be increased by using timespec datatype, but we instead divided the time obtained using time.h by *CLOCKS_PER_SECOND*. This would give us time with high resolution. For measuring the run time, we noted the the start time of the procedure and then the end time of the procedure. We then subtracted end time from start time to obtain the run time.

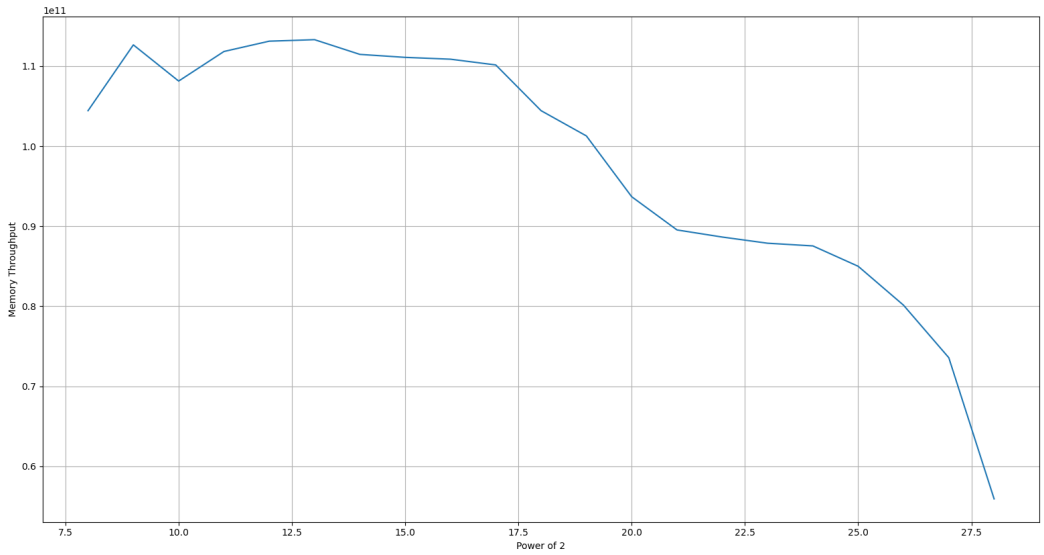


Figure 1: FLOPS for varying sized vectors

2. Time taken by the computation part is shown in the diagram along with memory access time.

3. Time taken by the memory access part is shown in the diagram below.

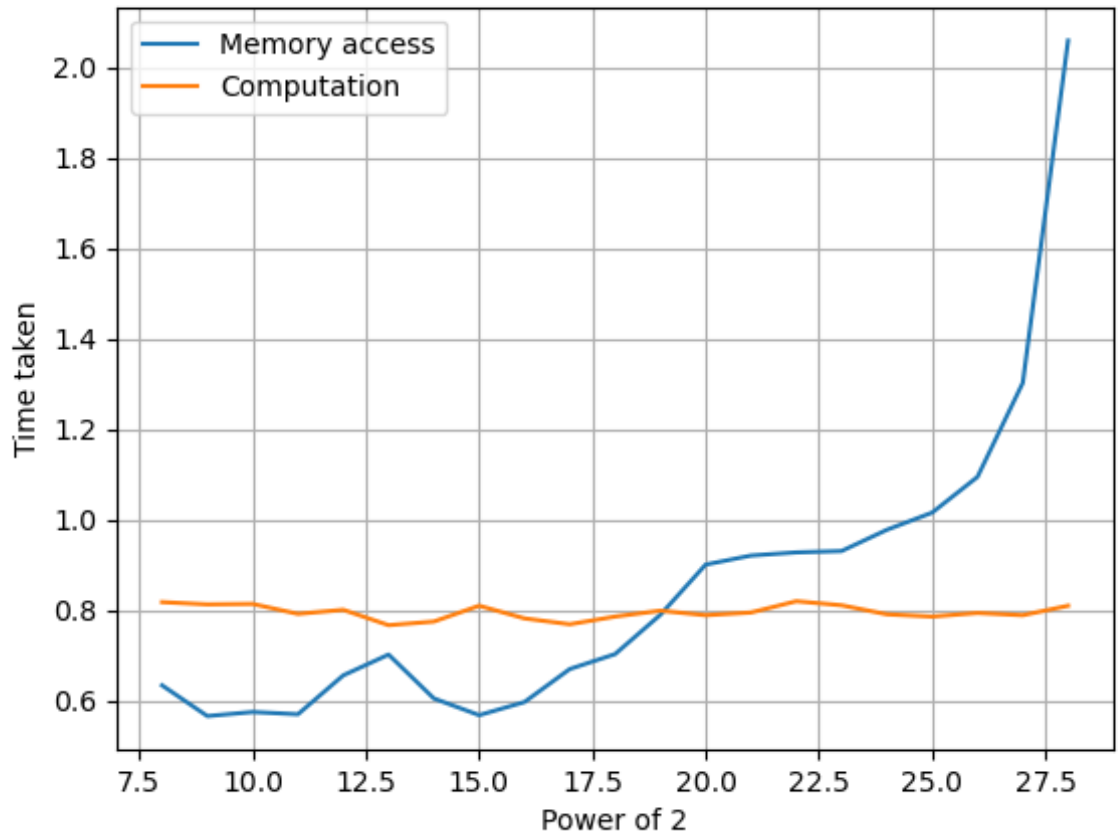


Figure 2: FLOPS for varying sized vectors

4. Peak theoretical performance = CPU clock frequency * no. of CPUs * Cores per CPU * Instructions per cycle Putting these values, we get $PTP \approx 211 GFLOPS$ which is significantly more than the measured performance. This is because the CPU clock frequency keeps fluctuating and also on how efficiently the parallel core algorithms run and getting a consistent 4 FLOPs per cycle.

5. Cost of operations is directly proportional to he number of transistors switch for an operation. there are only 2 operations ,addition and subtraction. Multiplication is repetitive addition and division is repetitive subtraction. Addition is just straight forward addition in binary. But subtraction is done by first converting it into 2's complement and then performing addition. So the conversion to 2's complement takes few more operations. Thus Subtraction is more expensive than addition. in turn making Division which is repetitive subtraction most expensive of arithmetic operations. Generally, processors strive to parallelize bit-pairs operations in order the minimize the clock cycles required. Multiplication algorithms can be parallelized quite effectively. Division algorithms can't be parallelized as efficiently.

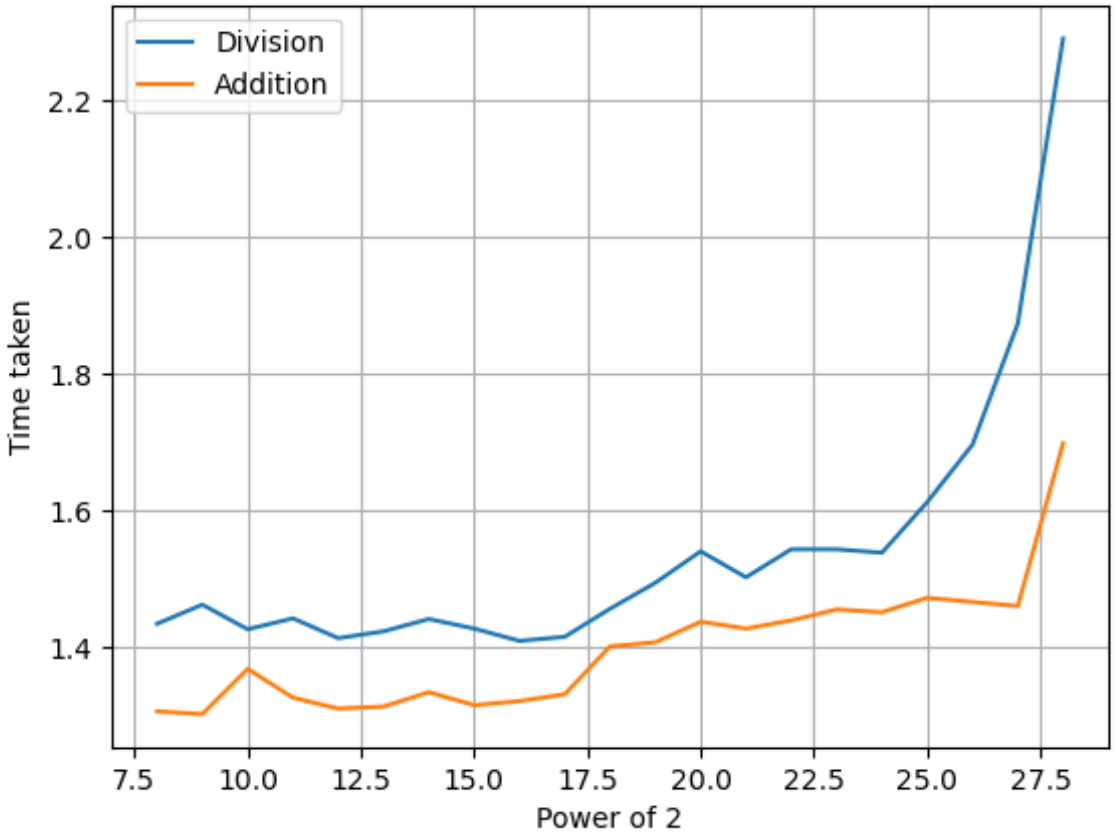


Figure 3: Time taken for varying sized vectors

6. The clock speed of 2GHz is less than the maximum clock speed of the CPU, i.e.3.3GHz which means if the clock speed is fixed at 2GHz , it's throughput should decrease according to the formula mentioned in the precious parts. However sustaining a given clock frequency is difficult since the clock frequency keeps fluctuating. One clever trick that can b used in our C programm is that if we know that the CPU is running at more than 2Ghz then we can let it sleep for some calculated amount of time in such that the average clock speed becomes 2Ghz. There

## B: Optimization Strategies

Two optimisation strategies have been used.

1. First we change the order of loop. This would help us to reduce cache miss rate.
2. Second we try to reduce the number of computations. We can also write $sin(\theta) * sin(\theta) - cos(\theta) * cos(\theta)$ as $-\cos(2\theta)$. And further this calculation is done repeatedly for same values of theta, so we do that computation only once by storing the values in an array for same values of theta. This computation is done in another loop outside the main loops. This reduces the number of computation drastically and hence we would reduce the time taken to run whole code with its help.

For smaller values of matrix dimensions, the elements are loaded into the cache to most of the references are HITS. In this case optimisation does not improve the performance significantly. However we see a very noticeable improvement in the performance as the matrix dimensions grow exponentially, in which our optimisation strategy helps to significantly improve the performance.

```
for(i=0;i<size;i++)
{
    double val=v[i]%256;
    v[i]=(-cos(2*val));
}
for(j=0;j<size;j++)
{
    for(i=0;i<size;i++)
    {
        if(true)
        {
            mat[j*size + i] = s[j*size + i]*v[i];
        }
    }
}
```

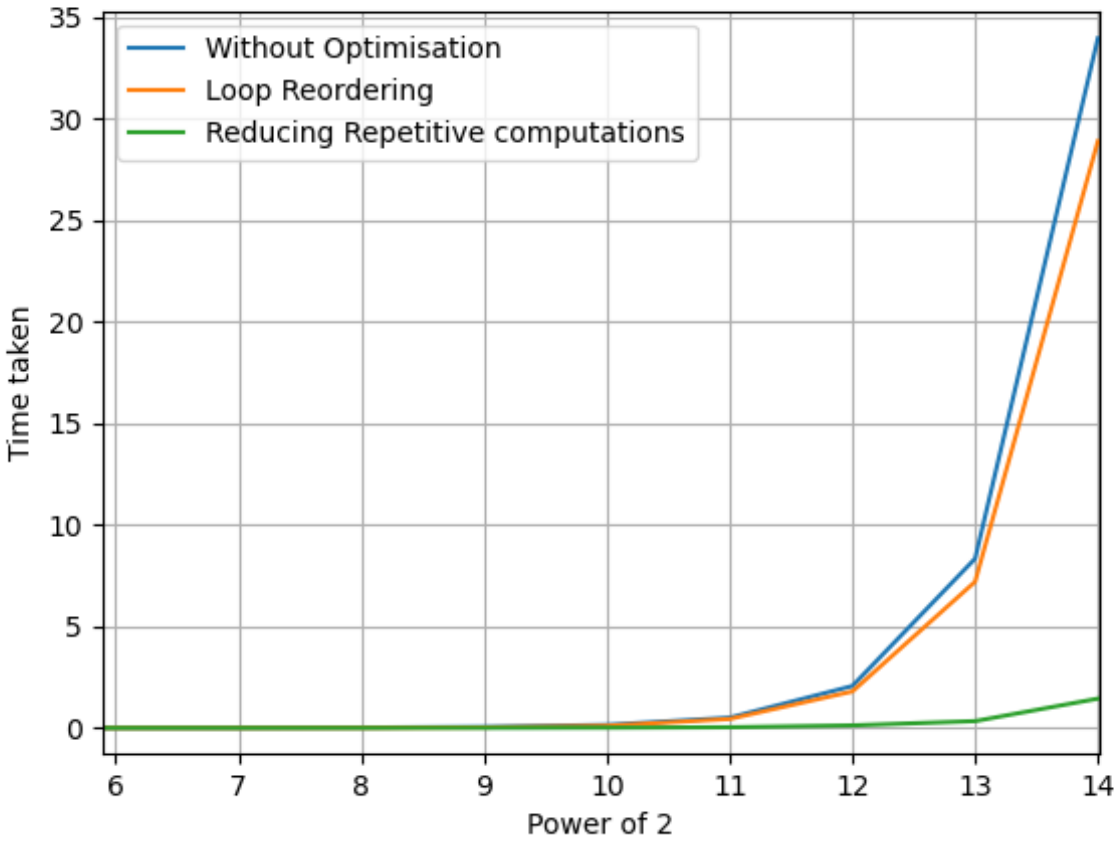


Figure 4: Time taken for varying sized vectors

In the first strategy we are reordering the loops. In this case, in case of matrices of very small size, we won't see a difference in the throughput. This is because cache is small enough to handle whole data, but in case of a larger matrices we would observe that the first strategy would lead to a more optimized version. This is because the time of spatial locality is helps in the case when we use loop reordering. Here for N = 2048, throughput is 128,170,675.57
In the second strategy, where we reduce the number of computations, we would definitely be better of than the non-optimized version irrespective of the size of the matrices, this is because we would be computing less number of things and hence less time would be taken. Here for N = 2048, throughput is 404,270,265.06

## C: High Performance Matrix Multiplication on a CPU

Our goal is to optimize matrix multiplication to run on a single core and understand how fast we can perform important linear algebra kernels/ routines. We have two algorithms for matrix multiplication.

First one is the simple matrix multiply (Unblocked) C=A*B algorithm having three loops. Here the the innermost loop is doing a dot product of row i of A and column j of B. and the two outermost loops are giving the i,j index values of the solution matrix.
Next is the square blocked matrix multiply which gives effective cache reuse.

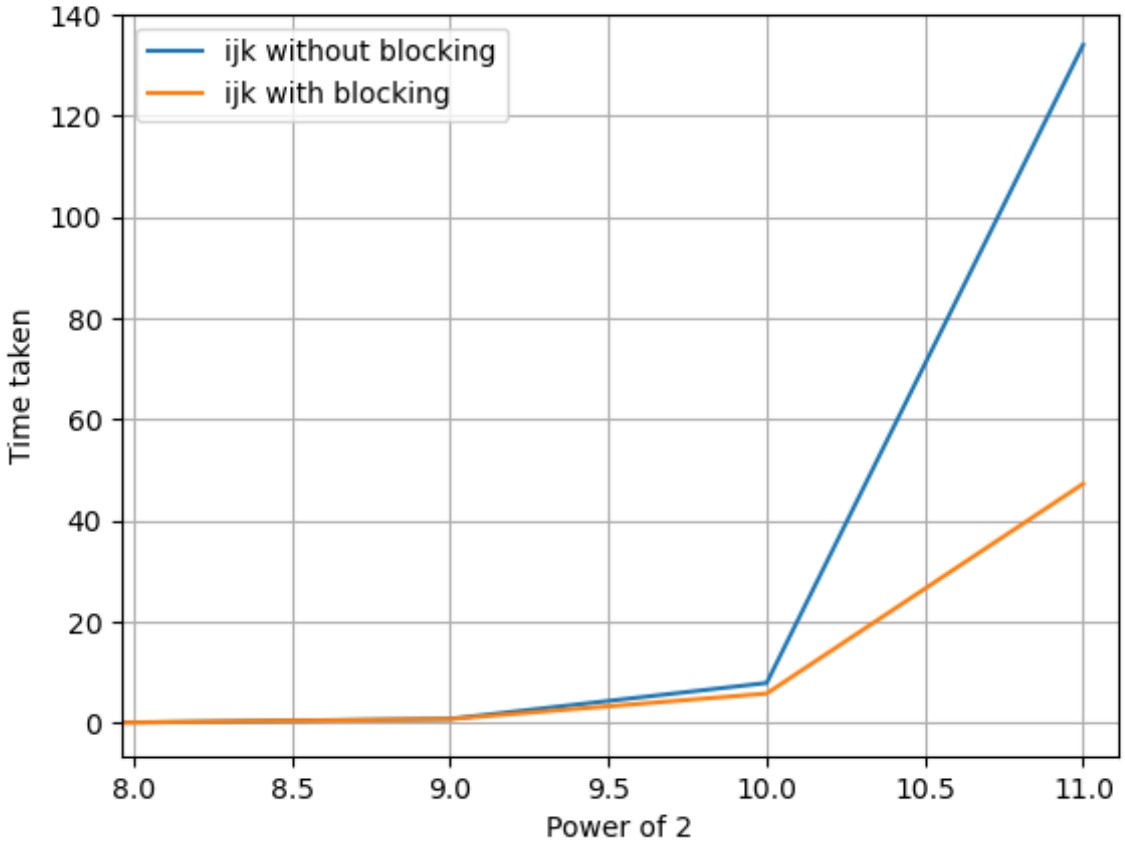


Figure 5: Time taken for varying sized vectors

As we can see in the graph, blocking is faster method to multiply matrices. However for smaller matrices the size fits into the cache so there is no difference between these two algorithms. As the matrix size increases above 512, we see significant differences in performance.

1. The best ordering for size = 256 is k-i-j loop ordering. The graph given below shows the relative performance of 6 different loop orderings for smaller matrix sizes.

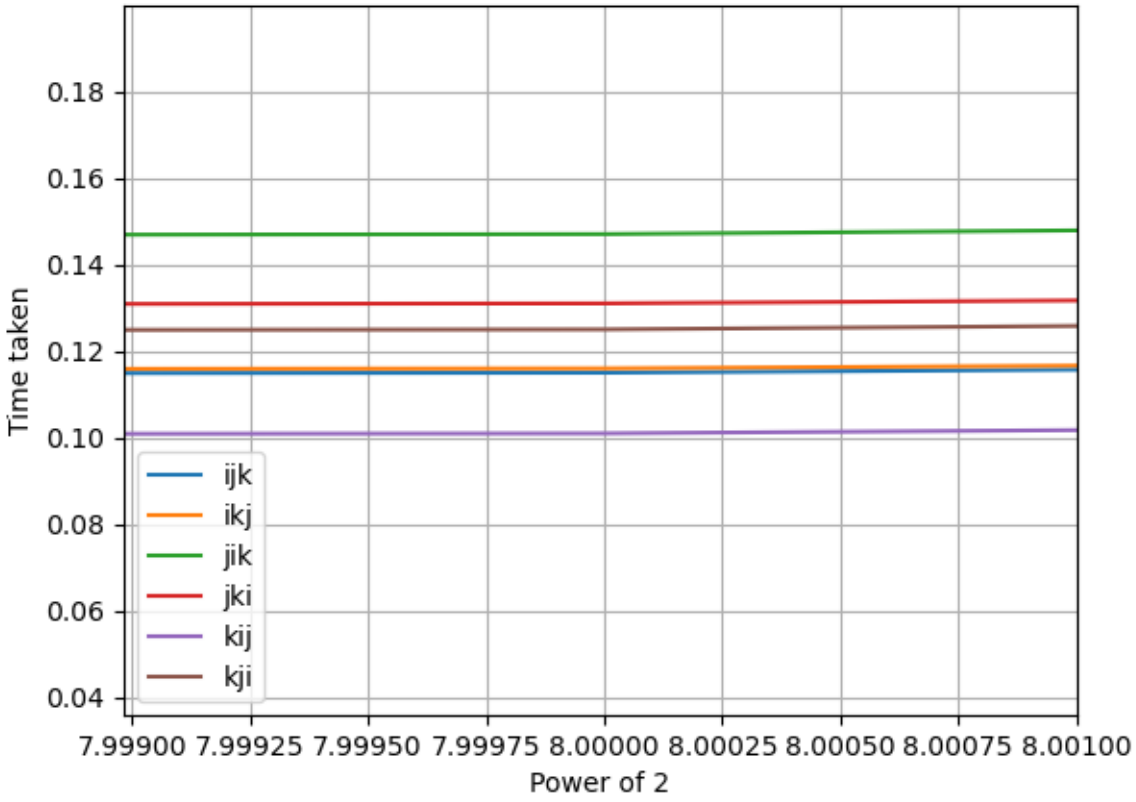


Figure 6: Time taken for varying sized vectors

Whereas, the best ordering for size = 2048 is is k-i-j loop ordering. The graph given below shows the relative performance of 6 different loop orderings for smaller matrix sizes.

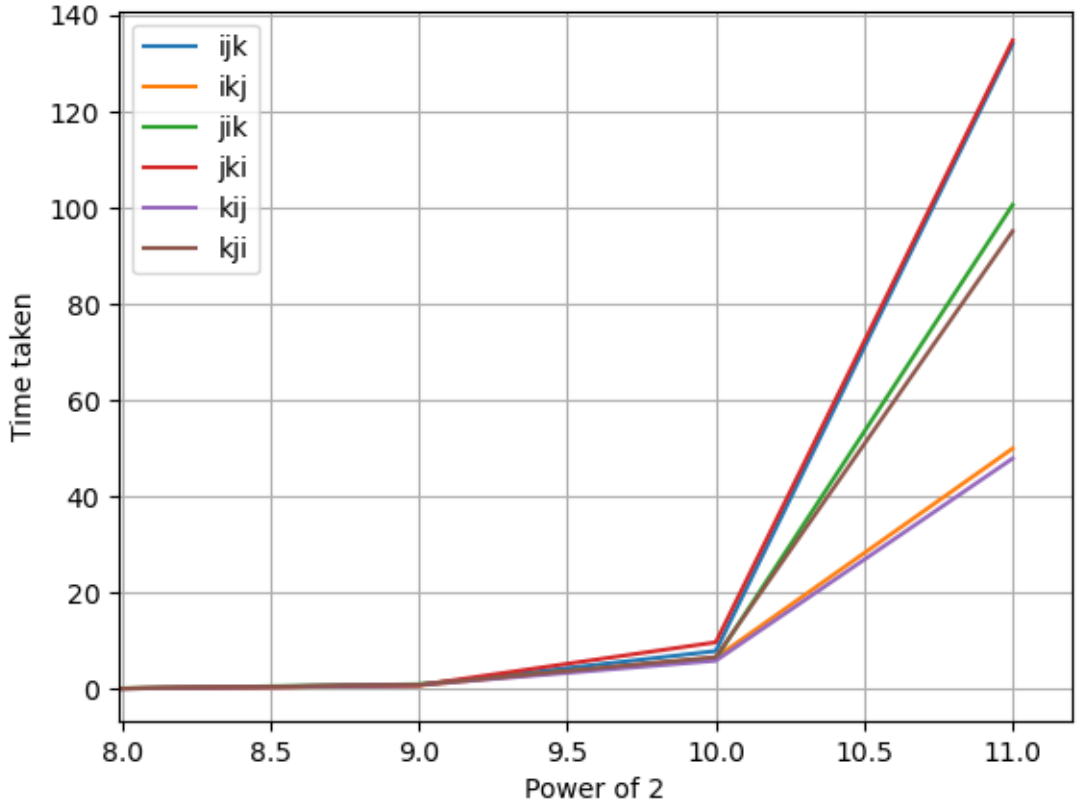


Figure 7: Time taken for varying sized vectors

In both cases, best performance is seen when j-loop is the innermost loop. This is giving the best cache locality as compared to other orderings.

2. The performance drop significantly for large values of matrices starting with size = 256. Larger matrices cannot be completely loaded into cache and thus the performance starts to decline significantly. As the size of the matrix increases, the time taken to calculate increases and hence the the throughput decreases. Since matrix cannot fit into cache, the memory access time becomes larger and larger thus increasing the multiplication time which is already in the order of 3. Here, we have done calculation for loop ordering i-j-k. For N = 256, the throughput is 291,777,669 and for N = 2048 throughput is 128,170,675.57. Throughput ratio for this values of N is 2.2764.

3. The following graph shows the comparison between integer and double data values. As we can see the time taken for computation is less for integer values as compared for double values. Int type operations are quicker as compared to double operations. This difference becomes significant for larger matrix sizes.

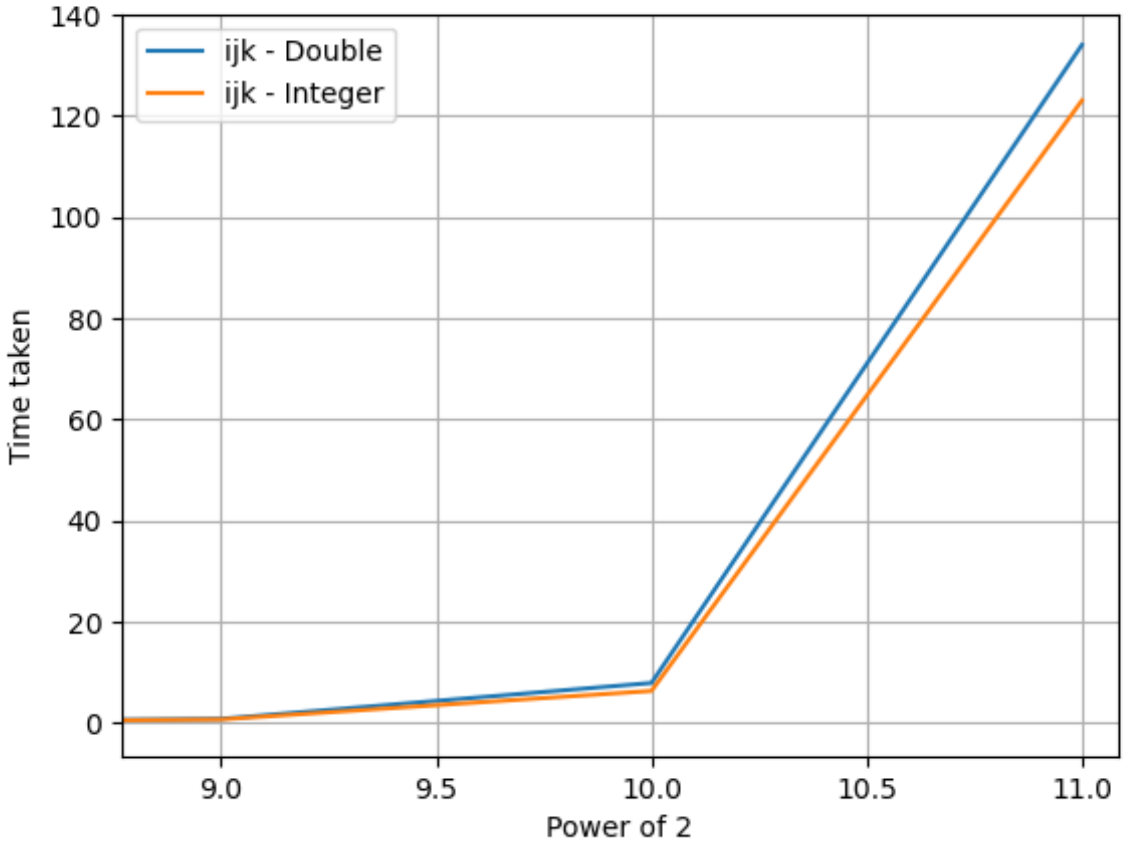


Figure 8: Time taken for varying sized vectors

Here for N = 2048 and integer values, throughput is 139,621,517.02 which is more than that of double, i.e. 128,170,675.57 for the same size.