CS 301 : HIGH PERFORMANCE COMPUTING

LAB 1

PURVIL MEHTA (201701073)
BHARGEY MEHTA (201701074)

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar

FEBRUARY 3, 2020

Contents

1 Vector Triad Benchmarking

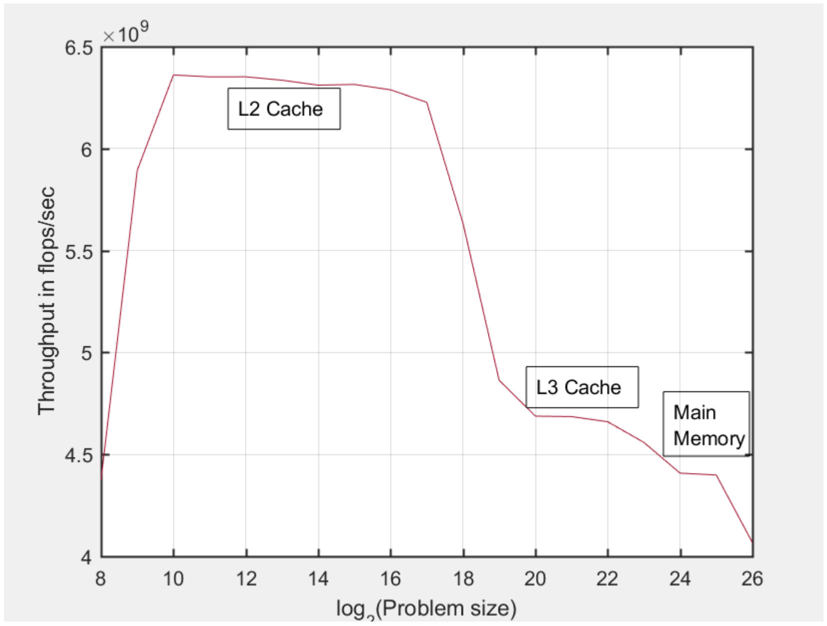


Figure 1: Drops due to memory

We find that the drops in the throughput occurs at the positions of the memory sizes of L2 and L3 caches. The drop corresponding to L1 cache is not visible. This is due to less difference in the latency of the L1 and L2 cache.

For L2 Cache: Drop should be at $\frac{\text{memorySize}}{NoOfVariables*doubleSize}\frac{256*2^{10}}{4*2} = 2^{16}$

For memory Drop should be at $\frac{\text{memorySize}}{NoOfVariables*doubleSize}\frac{6*1024*2^{10}}{4*2}$ 2^{21}

1.1 Clock Functions

We used the inbuilt clock functions from the time.h library. Calling the function twice around a code piece and taking the difference results in the clock ticks it took for the concerned code to get executed. By dividing it with the number of clock cycles per second, we get the approximate time in secs.

For smaller problem sizes, the clock ticks taken do not give an accurate measure of the elapsed time since the clock function itself is not that accurate and taking only one sample would result in high timing errors.

We do multiple runs of the code and measure the total time and average it over the number of runs of the code. Hence the absolute error remains the same but the averaging operation reduces the percentage error.

For larger problem sizes, fewer runs are enough since the percentage error would be small anyways.

1.2 Measuring the Compute Part

We modified the instruction line

$$a <= a + a * a$$

. This results in the value a being in the register always and hence we can the memory is not required since the variable is always found in the register.

The time thus taken would correspond to the time taken by the computation part.

1.3 Measuring the Memory Access Part

The total time taken by the normal code contains both the time required for computation and memory access. By finding the compute part of that time, we can subtract that time from the time taken by the unmodified code to found the approximate time spend in accessing memory.

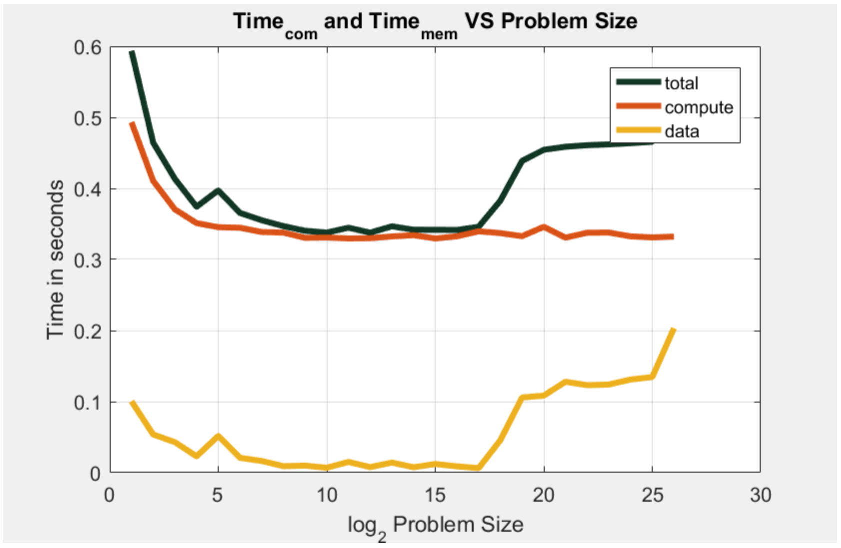


Figure 2: Compute V/s Memory Access Times

1.4 Comparison to Peak Performance

The peak performance for the system is given by

$$\text{Peak\_Flops/Sec} = \text{No\_of\_cores} * \text{Peak\_Freq} * \text{Instructions\_per\_Cycle}$$

$$\text{Peak\_Flops/Sec} = 4\text{Cores} * 3.3\text{GHz} * 4\text{IPC} = 54.8\text{GFlops/Sec}$$

The computational throughput achieved by the serial code was around 6.2GFlops/Sec. Hence only 11.74 % of the compute power is being utilized.

1.5 Cost of Addition and Division Operations

The addition operation is less expensive than the division operation. The line $a[i] <= b[i] + c[i] * d[i]$ was replaced with a static $a <= k/i$ where k is the inner loop variable. This results in the concerned variables always being present in the register.

The time taken thus would correspond the compute costs of the addition and division operations.

The throughput for the division operation was found out to be around 7.3 GFlops/Sec whereas the same for the addition operation was found out to be 8.95 GFlops/Sec. The throughput for the addition operation is more hence it is less expensive.

1.6 Setting Clock Frequency

We can set the maximum clock frequency of the processor with a linux command called cpupower. However this requires root access.

2 Optimizing Code

2.1 Strategies

- The assignment for each element is independent of each other. I.e. the value at mat[i][j] is dependent only on i and j. Hence we can interchange the inner and the outer loop which results in the code operating on one row at once.

- The routines for sin and the cos functions are being called twice each. Hence there are a total of 4 routine calls for each matrix element. We replace the expression by $-2\cos(2\theta)$ which results from a trigonometrical identity. This means that only one routine call is present for each matrix element.

2.2 Speedup

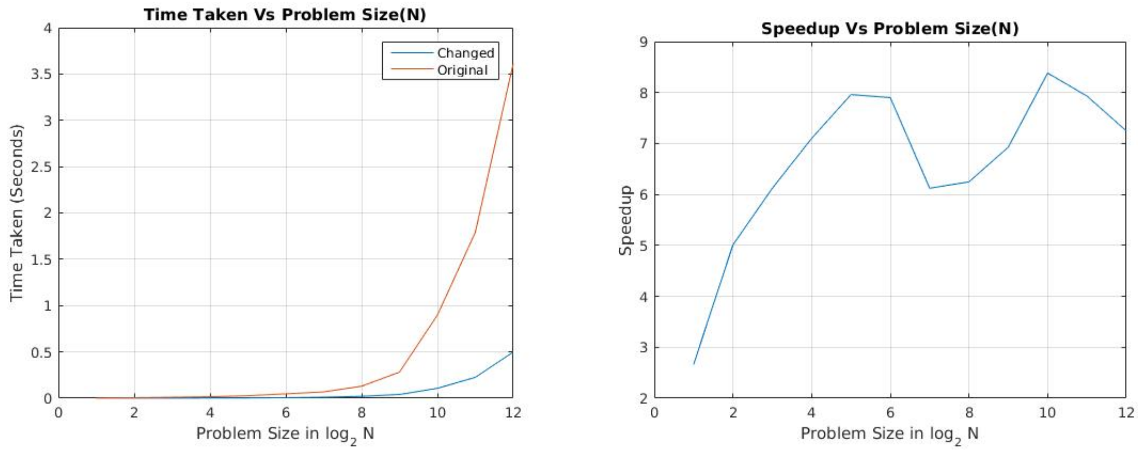


Figure 3: Time taken and speedup observed after optimizations

As seen here, the code achieved a maximum speedup of around 8 times after applying the above mentioned optimizations.

2.3 Dependence on Problem Size

- The first optimization works only on big problem sizes when the entire matrix wouldn't fit in the cache. Once the problem size exceeds this limit we start taking advantage of spatial locality and so we would see significant lags in the original code.
- The second optimization is problem size independent since it reduces the number of function calls and hence better utilization of compute power.

3 Block Matrix Multiplication VS Naive Approach

We noticed from the graph that, for the smaller problem size the whole $N*N$ matrix can be stored in cache memory and thus we will not get better throughput compare to naive approach in block multiplication code. Once problem size hits that critical mark then block matrix multiplication algorithm takes very less time compare to naive approach and thus throughput increases in the case of blocked matrix multiplication.

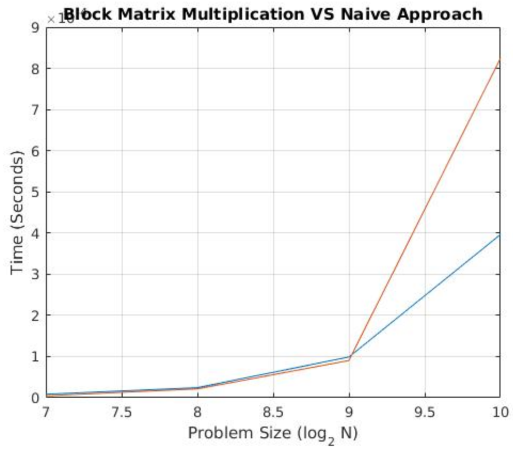
3.1 CMA Calculation for Block Matrix Multiplication

The CMA in the case of Naive approach is approx 2 whereas in the case of blocked chain multiplication is given by,

$$CMA = \frac{Totaloperations}{totalnumberoftimestogetslowmemory}$$

Total misses within one block will be = B^2.
Total misses to calculate = $(\frac{N}{B} + \frac{N}{B}) * B^2 = 2NB$
Total blocks = $(\frac{N}{B})^2$
Thus total misses to calculate whole matrix = $2NB(\frac{N}{B})^2 = \frac{2N^3}{B}$
Thus,

$$CMA = \frac{2N^3}{\frac{2N^3}{B}} = B$$



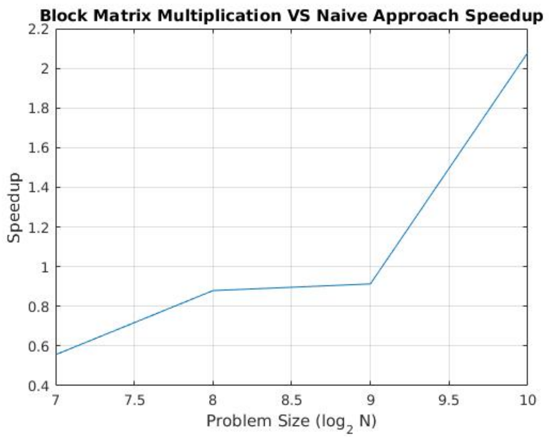


Figure 4: Time taken and speedup observed after optimizations

4 Loop Interchange in Matrix Multiplication

We would be using the following code while interchanging loops. C = A*B.

```
for i = 1:N
    for j = 1:N
        for k = 1:N
            c[i][j] += a[i][k] * b[k][j]
        end for
    end for
end for
```

4.1 Results

We can see that out of all possible combinations, we can segregate them into 3 groups depending upon the inner most loop's running variable.
I.e. i-j-k, j-i-k, k-j-i, j-k-i and i-k-j, k-i-j are the 3 groups of configurations that show similar time taken for a run.

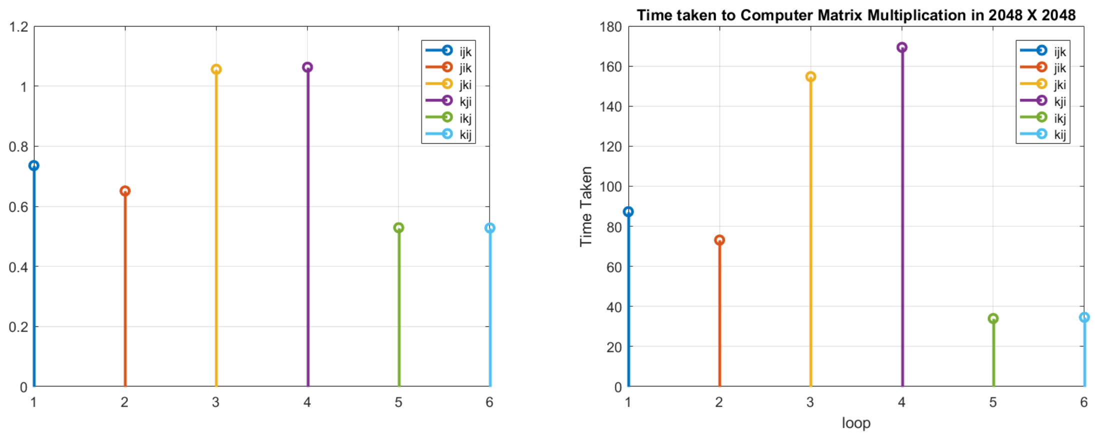


Figure 5: Time taken for various strategies- 512x512 and 2048x2048

4.2 Cache Miss Analysis

Configuration	Misses in A	Misses in B	Misses in C	Dominant Term
i-j-k	$\frac{1}{8}N^3$	N^3	$\frac{1}{8}N^2$	
j-i-k	$\frac{1}{8}N^3$	N^3	N^2	$\frac{9}{8}N^3$
i-k-j	$\frac{1}{8}N^2$	$\frac{1}{8}N^3$	$\frac{1}{8}N^3$	
k-i-j	N^2	$\frac{1}{8}N^3$	$\frac{1}{8}N^3$	$\frac{1}{4}N^3$
j-k-i	N^3	N^2	N^3	
k-j-i	N^3	$\frac{1}{8}N^2$	N^3	$2N^3$

The above analysis occurs by calculating the patter of cache misses for a matrix for each configuration.

- Configuration i-j-k
  - For each loop of i, the inner loop misses $\frac{N}{8}$ times and this happens N times since j runs from 1 to N and is independent of misses in A. There are i such rows, hence a total of $\frac{N^3}{8}$ misses are encountered.
  - For each loop of i, we access the matrix B in a column wise manner. For each column, we have N misses so for each loop of i, we have N^2 misses. Hence a total of N^3 misses.
  - For each loop of i, we access rows which result in $\frac{N}{8}$ misses and there are N such rows. For each element, we access the C matrix once and it remains in cache thereafter. Hence there are just $\frac{N^2}{8}$ misses.
- Configuration i-k-j
  - For all runs of j, we access rows of matrix A which result in $\frac{N}{8}$ misses and there are N such rows. For each element, we access the A matrix once and it remains in cache thereafter. Hence there are just $\frac{N^2}{8}$ misses.
  - For each loop of i, we access the matrix B in a row wise manner. For each column, we have $\frac{N}{8}$ misses so for each loop of i, we have $\frac{N^2}{8}$ misses. Hence a total of $\frac{N^3}{8}$ misses.

- For each loop of i, we access the row of C N times each. For each access of the row, there are $\frac{N}{8}$ misses, and at the beginning, the whole row is replaced in cache so for each run of k, we find $\frac{N}{8}$ misses which result in $\frac{N^2}{8}$ misses for each row to give a total of $\frac{N^3}{8}$ misses.

We can make such arguments for the rest of the configurations also. We find that the dominant term in each configuration set is different, hence the time taken is also different. However the term is independent of the outer two loop variables, hence the pairing of the configuration on the basis of the inner loop.

4.3 Throughput Analysis

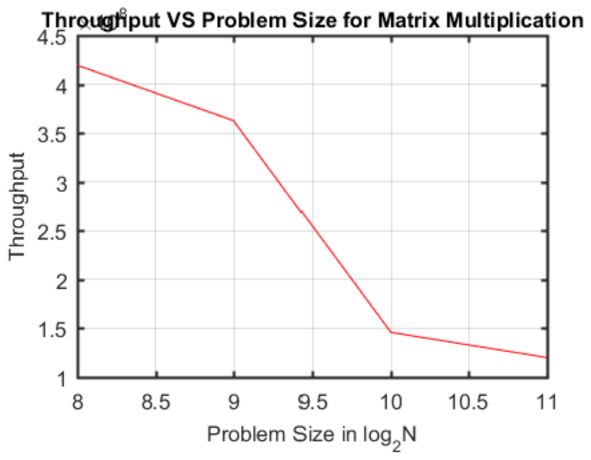


Figure 6: Throughput for different Problem size

Since the L1 cache size and the L2 cache for the machine is $32K$ and $256K$ respectively we observed similar trend in the throughput graph.Since the size of the problem initially is less than the size of the L1 cache we got higher throughput.As the size increases, it crosses the L1 cache size but still be able to fit in L2 cache.We got another drop for L2 cache in the graph.

4.4 Integer-Double Precision Analysis

Since we know that integer is 4 Byte has memory space and double precision has 8 Byte memory space it is clear that whenever misses occurs, system will bring half of the element in the case of double compare to integer.Thus total misses in double increases by the factor of 2 and thus throughput also increases. Exactly this kind of graph we also observed in our analysis which is shown below.

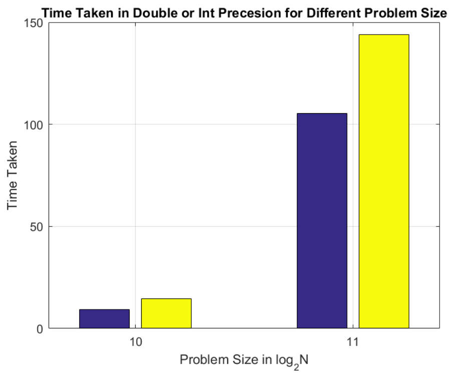


Figure 7: Time taken in double and integer precision by different problem size

5 Matrix Transposition

We use the block matrix transposition technique. We can see that for a block to get assigned to a correct position in the resultant block, we need to access both the blocks - i.e. in the source matrix and the target matrix. If both the blocks are in cache, the code is optimized.

$$2B^2 \leq C$$

$$2B^2 * \text{sizeof(int)} \leq C$$
$$2^3 * B^2 \leq 2^{14}$$
$$B \leq 2^6$$

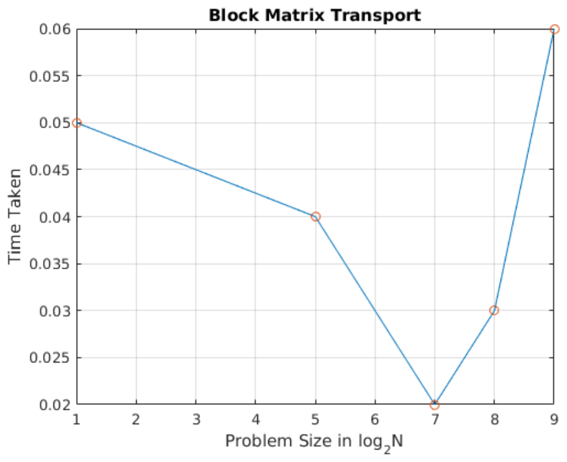


Figure 8: Block Transposition Run times for different block sizes

Hence we get the following curve. The code performs best for B = 6 and after that the code run time increases since the blocks do not fit the size of the cache.