Assignment 1
Sequential Matrix Multiplication

Syed Hassan Jalil - 11161604
Trisha Anand - 11434104

Introduction

For our first assignment we had to implement a simple sequential implementation of matrix multiplication, and then optimize it and report our optimizations. Moreover we also have to make a model which would help us predict how long the algorithm will take to execute with the given size of the matrix

Matrix Multiplication Implementation

Our First implementation was a very basic Matrix Multiplication code written in C. It reads the input matrix from text files and writes the answer for their product to a text file as we. It takes 5 arguments i.e Textfile1.txt M1 N1 TextFile2.txt M2 N2.
Here Textfile1.txt is the first text file containing matrix one. The text file should have numbers separated by blank spaces and each line representing one row of the matrix. M1 and N1 are the dimensions of Matrix 1 and M2 and N2 are dimensions of matrix 2 respectively.
We have also provided a python script that can be used to generate these text file out of random numbers. You can run this using the command
python datagenerator.py textfile.txt m n
Where textfile.txt is the textfile you want the code to write the matrix to, and m and n are the dimensions of the matrix respectively.

For our nive implementation we use a very simple nested loop to calculate the product of the two matrix

```
for(i=0;i<N1;i++){
        for (j=0;j<M2;j++){
            ans[i*M2+j]=0;
            for(int k=0;k<M1;k++){
                ans[i*M2+j]+=matrix[i*M1+k]*matrix2[k*M2+j];

            }

            fprintf(f,"%d\t",ans[i*M2+j]);
        }
        fprintf(f,"\n");
    }
```

So we use a simple nested loops to calculate the answer matrix and then write it to a file.

Performance Modeling

For the sequential matrix multiplication, calculation for the number of computations is as follows :

For each entry in the result matrix ans, number of operations = M1 multiplications + M1 - 1 additions.

So for all entries, total number of computations = N1 * M2 * (M1 + (M1 - 1)) = N1 * M2 * (M1 + M1)
*Assuming the cost of multiplication and addition is the same.

If the cache line size is 64 bytes, then each memory read for an element of the matrix should lead to 16 elements being fetched into the cache line. Since matrix1 is accessed row-wise, subsequent access in the inner loop of matrix1 should lead to cache hits. Matrix2 on the other hand accesses the elements column wise leading to accessing main memory on each read. Also, since result entry is updated after every multipilcation, its written to the main memory on each computation.

For each answer element, number of memory accesses = M1 + M1/16 + M1

So for all entries, total number of memory accesses = N1 * M2 * (M1 + M1/16 + M1)

FLOPs/byte = Total Computations / Total number of memory accesses * 4
= [N1 * M2 * (M1 + M1)] / [N1 * M2 * M1 * 33/16]*4
= 32 / 33*4
= 0.242 FLOPs/B

Clearly the sequential version of the program is memory intensive.

Optimization 1
One major issue for our implementation is that access of matrix2 is not sequential. It was being multiplied by k which gets incremented on each iterations. Which leads to cache misses and results in reduction in performance. A better implementation would be to access matrix2 row wise, which would ensure better cache utilization and give us better performance. To obtain this we first transpose matrix2 and then multiply it row wise. This can be seen in the following code

```
for (int i = 0; i < M2; ++i)
    {
        for (int j = 0; j < N2; ++j)
        {
            matrix2Transpose[i + j * N2] = matrix2[i * M2 + j];

        }
```

```
    }
    int * ans= malloc(sizeof(int)*N1*M2);
    for(i=0;i<N1;i++){
        for (j=0;j<M2;j++){
                ans[i*M2+j]=0;
                for(int k=0;k<M1;k++){
                        ans[i*M2+j]+=matrix[i*M1+k]*matrix2Transpose[k + j * N2];

                }

                fprintf(f,"%d\t",ans[i*M2+j]);
        }
        fprintf(f,"\n");
    }
```

Performance Modelling

Extra step added here is matrix transposing. The number of computations remain unchanged.

For transposing the matrix, number of memory accesses = N2 * M2 * 2

Now, since both the matrices are accessed row-wise, we get the benefit of cache hits. For calculating each entry for the result matrix, now the number of memory accesses = M1 + M1/16 + M1/16.

So for all entries, number of memory accesses = N1 * M2 * (M1 + 2*M1/16)

Total number of memory accesses = N2*M2*2 + N1*M2*(18 * M1 / 16)

Therefore, FLOPs/byte = [N1 * M2 * 2 * M1] / [(N2 * M2) + (N1 * M2 * 18 * M1 /16)]*4
= [2 * N1 * M1] / [N2 + N1 * M1 * 18/16]*4

For large enough N1 and M1, N2 can be ignored from the denominator giving us an approximate FLOPs/byte = 32/18*4 = 0.444.

This application is still memory bound but performs better than the basic sequential version because of higher compute per byte read from memory (0.444 > 0.242).

Optimization 2

Our second optimization was smaller. We used code motion and reduction to move calculations outside the loop as well as using additions instead of multiplication. This should give us a slight performance boost as well

```
int i2=0;
   for (int i = 0; i < M2; ++i)
    {

        for (int j = 0; j < N2; ++j)
        {
            matrix2Transpose[i + j * N2] = matrix2[i2+ j];

        }
        i2=i2+M2;
    }
    int * ans= malloc(sizeof(int)*N1*M2);
    int iM1=0;
    int iM2=0;
    for(i=0;i<N1;i++){
       int j2=0;
       for (j=0;j<M2;j++){
              ans[i*M2+j]=0;
              for(int k=0;k<M1;k++){
                     ans[iM2+j]+=matrix[iM1+k]*matrix2Transpose[k + j2];

              }

              fprintf(f,"%d\t",ans[iM2+j]);
              j2=j2+M2;
       }
       fprintf(f,"\n");
       iM1+=M1;
       iM2+=M2;
    }
```

Compiler Optimization and Vectorization

As part of our final optimization, we decided to use compiler optimizations and auto vectorization. We used the flag -02 for compiler optimization and then used -ftree-vectorize. The reason we used flag -02 instead of -03 is that we wanted to see the difference vectorization makes. -O3 implements vectorization by default, so we decided to have the same level of compiler optimization in all our implementations and then use a separate flag to auto vectorize the code to see the difference between the vectorized version and non vectorized version.

Performance Evaluation

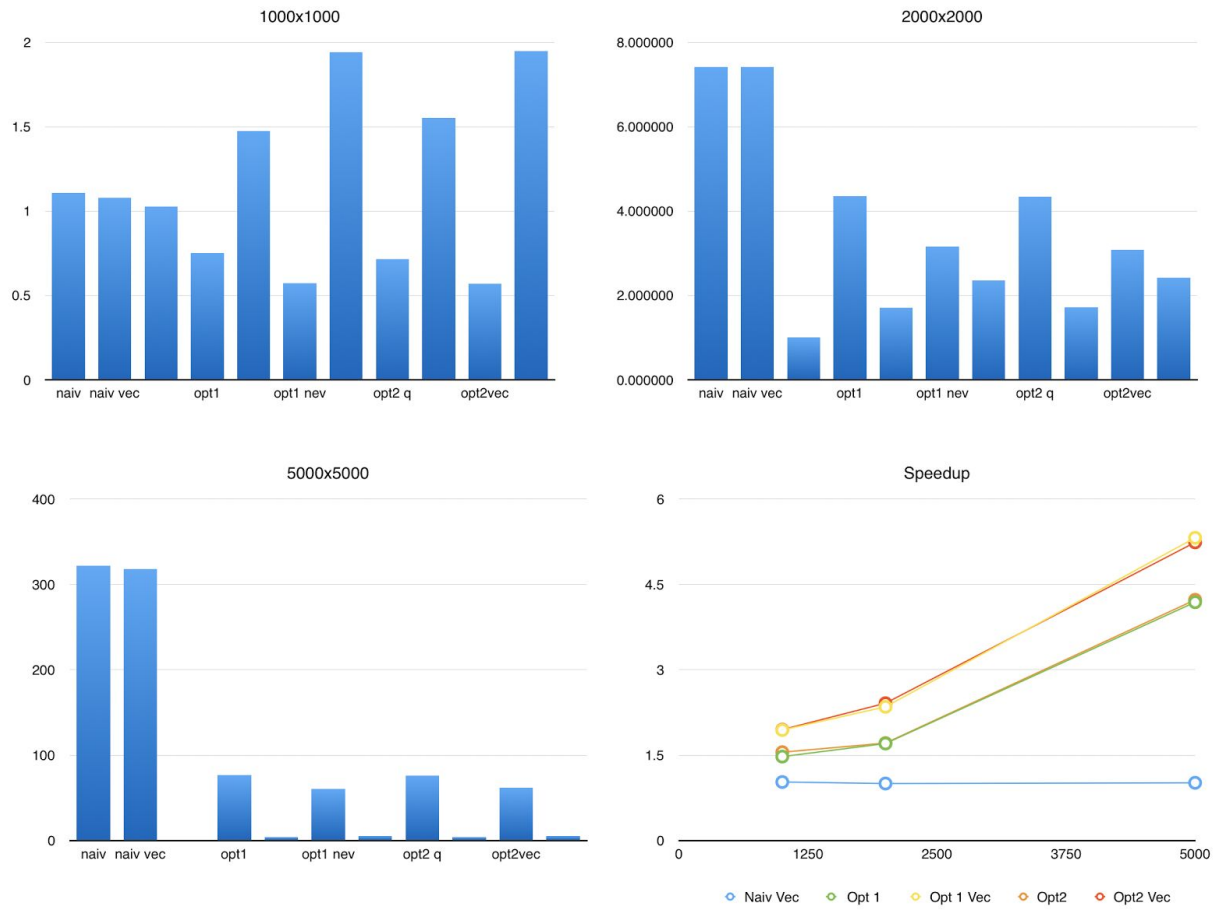
Finally we would like to evaluate the performance of all the three versions of the code, with and without optimization. For this purpose we used matrix of three sizes

1. 1000x1000
2. 2000x2000
3. 5000x5000

We generated these matrix using our datagenerator python script provided along with this assignment. Following table shows the results of these

Method	1000x1000	2000x2000	5000x5000
Naive	1.107883 sec	7.420560 sec	322.297312 sec
Naive vectorized	1.078764 sec	7.414174 sec	318.061248 sec
Opt1	0.751838 sec	4.355420 sec	76.953112
Opt1 vectorized	0.570762 sec	3.159462 sec	61.688312 sec
Opt2	0.714244 sec	4.335392 sec	76.194352 sec
Opt2 vectorized	0.568738 sec	3.076348 sec	61.506296 sec

The following charts can help you visualize the data better.



Runtime estimation

With a FLOPs/byte being as low as 0.44, the application is highly memory bound. This leads to the compute time being heavily influenced by the size of the matrix and the observed run time increases exponentially with increase in size instead of a linear increase. To estimate the run time we could employ a statistical model by gathering enough runtime data over multiple runs. Unfortunately we weren't able to collect enough data to build a statistical model ourselves for this assignment to predict run time.

Conclusion

The basic sequential version that was implemented had a FLOPs/byte ratio of 0.242. The application was clearly very memory bound. So, the first optimization implemented was to decrease the memory access per computation by transposing the matrix. This led to better cache hit ratio and higher FLOPs/byte of 0.44. The application is still memory bound and wouldn't benefit much from parallelization. Since we couldn't increase the FLOPs/byte any further, we used code motion and reduction to improve the runtime of the application.