Matrix Multiplication Optimization using Parallel Computing

Om Amar (BT23CSE106)
Ashish Pakhale (BT23CSE107)
Debasish Mondal (BT23CSE108)
Tanay Umre (BT23CSE109)

Abstract

Matrix Multiplication is a frequently used operation in various scientific and engineering applications. Yet it is not feasible for large matrices given the $O(n^3)$ time complexity. This algorithm explores the optimization of matrix multiplication using parallel computing through the usage of multithreading to reduce the time complexity from $O(n^3)$ to $O(n^2)$. Most of the modern computers and laptops have multiple cores and GPUs are designed to handle many tasks at once by using this, we can speed up matrix multiplication. This algorithm is inspired by Square Root Decomposition and its core logic is to divide a problem having complexity $O(n^3)$ to n problems each having a time complexity of $O(n^2)$ and running them parallelly to give an overall time complexity of $O(n^2)$

I. Introduction

Matrix multiplication is a frequently used operation but is really time taking and requires a lot of computation power, the traditional approach has a time complexity of $O(n^3)$, which is computationally expensive for large scale metrics, so we aim to reduce the theoretical time complexity from $O(n^3)$ to $O(n^2)$ while keeping the answer accurate by using multiple cores, we have also analyzed the computation time taken by the usage of different number of threads used and we have tried to find an optimum relation between the number of cores we have and the number of threads we are creating by ranging the number of threads from 1 to n and comparing the time taken by them and the traditional method while ensuring the accuracy of the output matrix. The most optimized already existing Matrix multiplication method, namely Coppersmith-Winograd algorithm has a time complexity of $O(n^{2.376})$ and another famous algorithm being Strassen's Matrix multiplication having a time complexity of $O(n^{2.81})$ which works on the principle of divide and conquer. Which makes an algorithm working in $O(n^2)$ a huge improvement over the already existing algorithms. To optimize the algorithm twice and convert two of the $O(n)$ factors to $O(\sqrt{n})$ each while ensuring accuracy and minimizing the overhead generated by the creation of so many threads makes it a lot more complex to implement while keeping it optimized. This is more of a theoretical approach, given the limited number of CPU Cores even in modern machines, and in an ideal environment, we would need at least n cores to get the most optimized output.

II. Related Work

There have been a few studies related to this, the latest being "Matrix Multiplication Analysis on Sequential and Parallel Computation using CUDA" published by Robertus Hudi, Alessandro Luiz Kartika, Dave Joshua Marcell, Winston Renatan which discusses on the trade off between the memory and time consumption while using parallel programming and finding the optimal relation between them by trial and error testing as well as finding the optimum number of threads to get the best performance with help of CUDA (parallel computing platform developed by NVIDIA), another being "Analysis of multi-threading time metric on single and multi-core CPUs with Matrix Multiplication" by Dhruva R. Rinku, M. Asha Rani which aims on improving the performance of the system with time constraints by running parallel threads on different cores. There are not many papers primarily focused on matrix multiplication, making it a less explored topic.

III. Problem Analysis

We have to optimize the traditional Matrix Multiplication method from $O(n^3)$ to $O(n^2)$ by making use of all the CPU Cores by using multithreading. On first thought, it seems impossible to do so, but it is quite similar to Square Root Decomposition, in which we divide a problem of size n to $\sqrt{n}$ problems of size $\sqrt{n}$ each and compute individually the answer for each block and the combine them to get the final answer, in a similar way, we aim to simplify the rows and columns matrix into blocks of $\sqrt{n}$ each, and run the multiplication of matrices independently, which is implemented using multithreading so that these blocks run simultaneously. Our aim is to divide the blocks optimally so as to minimize the time taken. We also aim on finding the optimal number of threads/blocks that we must divide our matrix into to get the quickest solution. Not much research has been done in this, since this is a theoretical approach, to get the ideal outcome, we would need a huge number of cores, making it quite impractical. But it might be feasible as new technology come and the computation power becomes more, supporting parallel computing. As of now, one of the best platform for parallel computing is CUDA, which has 16384 cores, which are particularly optimized for parallel computing, which would make it efficient for even larger matrices having the value of n up to 10^4 which is enough for practical purposes and would give us a matrix multiplication having a time complexity of $O(n^2)$.

IV. Algorithm

We calculate each element of the answer array by creating n threads each processing n elements.

$for\ all$

$$0 \leq x < \sqrt{n}, 0 \leq y < \sqrt{n},$$

$$Ans[i][j] = \sum_{k=0}^{n} A[i][k] * B[k][j]$$

$where$

$$i \in \left(x * \sqrt{n}, (x+1) * \sqrt{n} - 1\right)$$

$$j \in \left(y * \sqrt{n}, (y+1) * \sqrt{n} - 1\right)$$

This equation is applied for all possible values of x and y from 0 to $\sqrt{n}$, these equations are run in parallel, making the total number of threads as the number of combinations of x and y, which is n, and in each of these threads, we perform the traditional matrix multiplication, which requires a time complexity of $O(n^2)$, which makes the total time complexity as $O(\max(threads, computation\ of\ thread))$ which is $O(n^2)$.

V. Pseudocode

The functions create_thread computes the answer for the given range in the parameters.

```
$start$

$take\ A\ as\ input$
$take\ B\ as\ input$
$initialize\ array\ ans$

$func\ create\_thread(i_{start}: i_{end}, j_{start}: j_{end})$
  $for(int\ i = i_{start}; i < i_{end}; i++)$
   $for(int\ j = j_{start}; j < j_{end}; j++)$
    $for(int\ k = 0; k < n; k++)$
     $ans[i][j] += sum([A[i][k] * B[k][j]])$
  $return$
```

We create all possibles thread so that all ranges are covered from 0 to n.

```
$for(i = 0; i < sqrt(n); i++)$
 $for(j = 0; j < sqrt(n); j++)$
   $create\_thread(i: i + \sqrt{n}, j: j + \sqrt{n})$
$end$
```

This way, we create a total of n threads and each thread(the function create_thread) will have a complexity of $O(n^2)$.

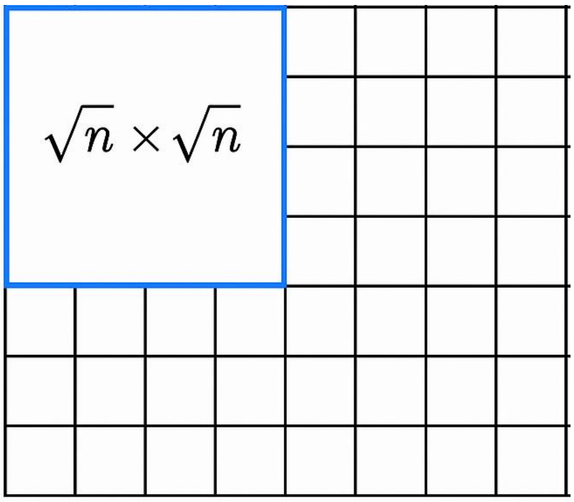

Fig 1: Image representing the division of matrices

As shown in fig. 1, we divide a $n * n$ matrix into n matrices each of size $\sqrt{n} * \sqrt{n}$, and create a thread for this submatrix, which is then computed independently of other parts of the matrix, thus ensuring parallelism.

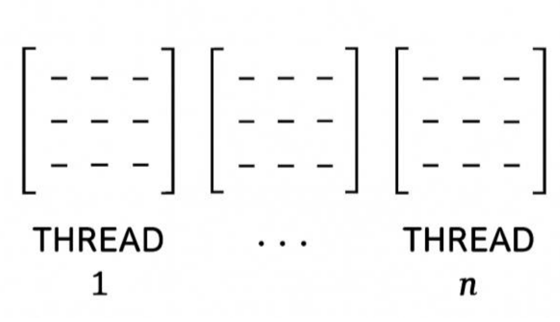

Fig 2: Image representing different threads running simultaneously

Fig. 2 shows the n threads created parallelly, which will run independently, giving us a total complexity of an individual thread.

VI. Observation

We have considered the time taken for different values of n (different matrix sizes) and compared the time taken by the traditional and the optimized Matrix multiplication Algorithm as shown in fig. 3.

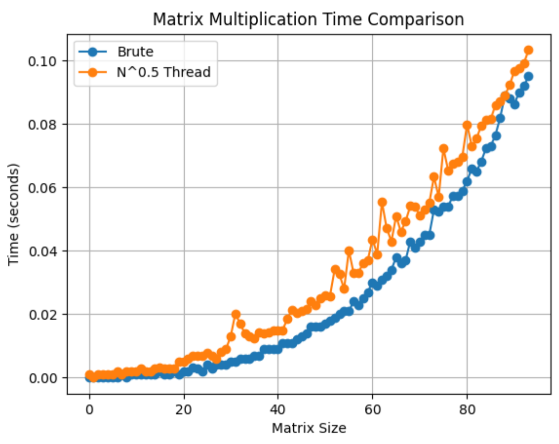

Fig. 3: Comparison of Traditional vs Multithreading approach

As seen in fig. 3, the multithreading approach is consistently slower than the traditional method, the reason being the overhead caused by the creation of the threads and the limited number of cores in the machine used to run the code.

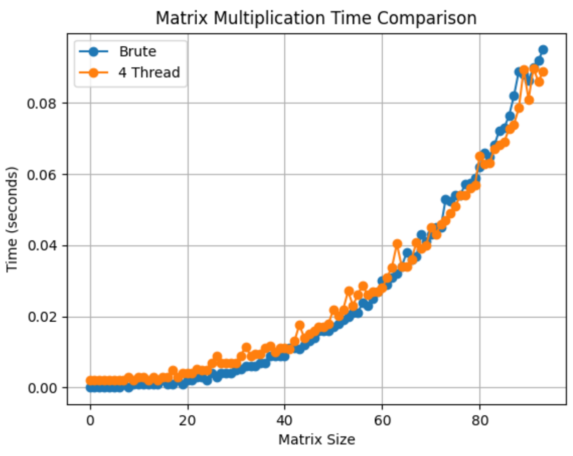

Fig. 4: Comparison of Traditional Method vs Multithreading while using 4 threads.

In fig. 4, we can notice that the computation time becomes lesser while using 4 threads than traditional method as the size of the matrix increases, meaning that the overhead cause due to the creation of threads is not much, since the number of threads are limited, giving us a faster solution for bigger values of n. Whereas for the smaller value, the multithreading approach takes more time, since overhead due to creation of threads, takes a lot more time compared to the multiplication of the matrices, making it the dominant factor.

This shows us that when the number of threads we are using is within the number of cores of the machine, it gives us a more optimized approach then traditional matrix multiplication algorithm. An in an ideal machine, which would support parallell computing, the time complexity would reduce from $O(n^3)$ to $O(n^2)$, making it a lot more faster.

Table 1 consists of different number of cores versus their computation time, which indicates that with increasing number of threads, the total computation as well as average computation time over a range of matrix having different sizes decreases, indicating that the overhead of threads has a big factor and affects the total time taken, thus not giving the expected output.

Table 1. Table consisting of Number of threads, total computation time, average computation time over one matrix each of size ranging from 6 to 100

Number of Threads	Total Computation(s)	Average Computation(s)
4	1.9033	0.0202
9	2.0024	0.0213
16	2.0959	0.0222
25	2.2153	0.0235
n	2.5013	0.0266

VII. Conclusion

The overhead of the creation of thread and the limited number of cores restricts us from getting the expected theoretical optimized solution having $O(n^2)$ complexity, meaning that the theoretical approach would give a better outcome most of the time, but when using a limited number of threads less than the number of cores, the execution was faster than the traditional method, thus implying that theoretically if we had a machine with a huge number of cores, we would be able to achieve a time complexity of $O(n^2)$ for matrix multiplication, which would be quite beneficial.