High Performance Computer Architecture (E0 - 243)
Assignment 2 (Part - A)
MTech - CSA (IISC Bangalore)

Submitted By:

Anushka Pateriya (23130)

Sahil Lathiya (22691)

Optimizing Performance of Dilated Convolution

1. GitHub repository link

https://github.com/SahilLathiya/CA_Assignment_2

2. Introduction

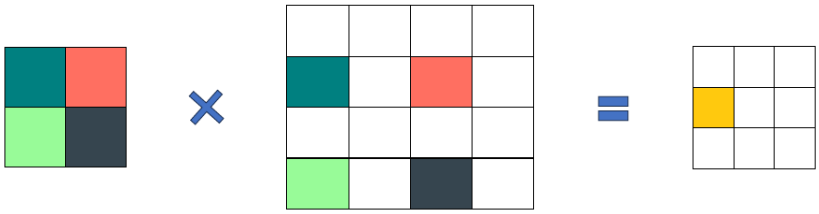
Here, we are given an implementation of Dilated Convolution which involves an input matrix and a kernel matrix.

Input: Input Matrix (I) of dimensions: Input_Row x Input_Column. &
Kernel Matrix (K) of dimensions: Kernel_Row x Kernel_Column.

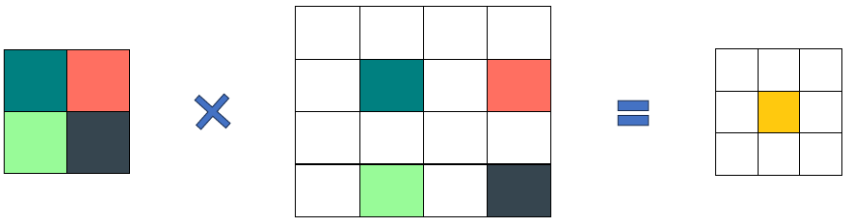
Output: An Output Matrix (O) of dimensions:
(Input_Row - Kernel_Row + 1) x (Input_Column - Kernel_Column + 1)

The Kernel Matrix (K) slides over the Input Matrix (I) and produces each cell of the Output Matrix (O) as shown below. In the diagrams below, The Kernel Matrix (K) is the left-most, the Input-Matrix (I) is in the middle, and the Output Matrix (O) is the right-most.

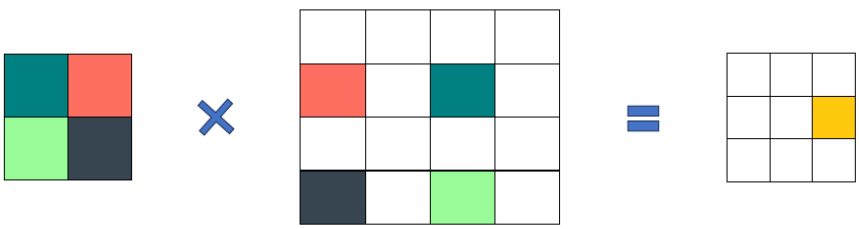
$$O_{10} = I_{10} * K_{00} + I_{12} * K_{01} + I_{30} * K_{10} + I_{32} * K_{11}$$



$$O_{11} = I_{11} * K_{00} + I_{13} * K_{01} + I_{31} * K_{10} + I_{33} * K_{11}$$



$$O_{12} = I_{12} * K_{00} + I_{10} * K_{01} + I_{32} * K_{10} + I_{30} * K_{11}$$



3. Processor Specification

We used two different systems with the given specifications to execute our code and perform optimization.

System 1:

CPU	11th Gen Intel(R) Core (TM) i5-11500 @ 2.70GHz
Memory	16 GB
Cache	L1d cache = 288 KB, L1i cache = 192 KB, L2 cache = 3 MB, L3 cache = 12 MB
OS	Ubuntu 22.04.3 LTS
Core	12
Threads	2 per Core

System 2:

CPU	AMD Ryzen 7 6800H
Memory	16 GB
Cache	L1d cache = 256 KB, L1i cache = 256 KB, L2 cache = 4 MB, L3 cache = 16 MB
GPU	NVIDIA GeForce RTX 3050
GPU Memory	4 GB
CUDA	12.2
OS	Ubuntu 22.04.1
Core	16
Threads	2 per Core

4. Observation and Analysis

We have given a naive approach for DC. The primary concern with the naive implementation is its potential performance limitations, especially for large datasets. The use of nested loops and the lack of optimization techniques might result in higher computational overhead, impacting the overall efficiency. The given code does not leverage certain techniques, such as loop unrolling or SIMD instructions, which could enhance computational efficiency.

Part A(I):

1. Optimizing the DC

1.1 How can we optimize this code?

a. First, we change the order of loop execution. So, we had seen that order output_i -> output_j -> kernel_i -> kernel_j run better compared to other loop execution order.
b. Secondly, the application of compiler optimization options, specifically the utilization of the -O3 flag during execution, enables the compiler to autonomously optimize the code.
c. Thirdly, optimization techniques such as Loop Unrolling, Reduced Conditional Statements, Strength Reduction, Variable Hoisting, Minimized Redundant Computations, and Avoidance of Redundant Array Access can be employed to enhance the efficiency of this code.

1.2 Loop Unrolling

The primary optimization technique employed was loop unrolling. By unrolling the output_j loop and kernel_j loop by degree, the code benefits from reduced loop overhead and improved instruction scheduling. This results in enhanced computational efficiency.

1.3 Reduced Conditional Statements

To streamline the code and enhance efficiency, the number of conditional statements was minimized. Loop unrolling played a crucial role in reducing the need for conditionals, as the code could handle multiple iterations in a more straightforward manner.

1.4 Variable Hoisting

Certain calculations, such as 'output_i_into_output_col' and 'kernel_ i_into_kernel_col', were hoisted outside the innermost loops to reduce redundant computations. This contributes to overall performance improvement by avoiding repeated calculations.

1.5 Minimized Redundant Computations

This optimization aims to eliminate unnecessary or repeated calculations, reducing overall computational overhead and enhancing the efficiency of the program.

1.6 Avoiding Redundant Array Access

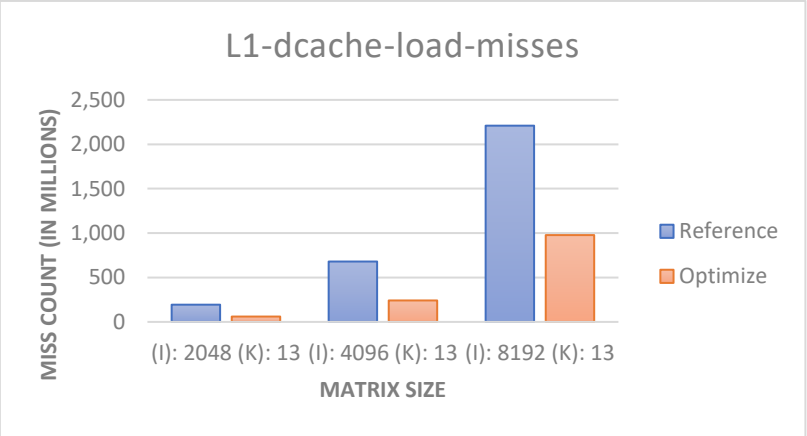
Store intermediate results, represented by local variables ('ans_0', 'ans_1', 'ans_2', 'ans_3'), were introduced to store values before updating the output array. This minimizes redundant array access and enhances cache locality.

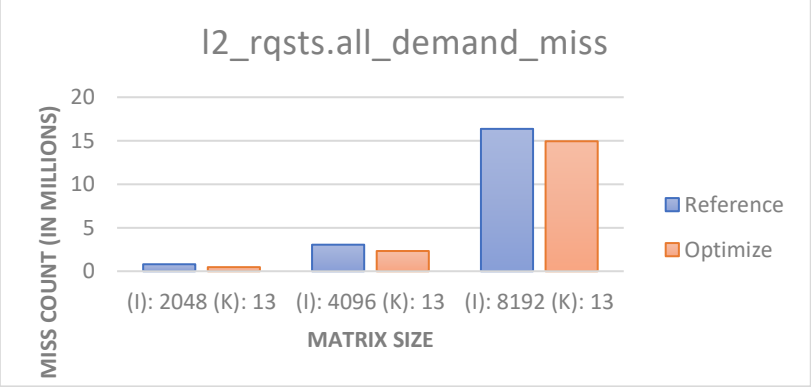
1.7 Strength Reduction

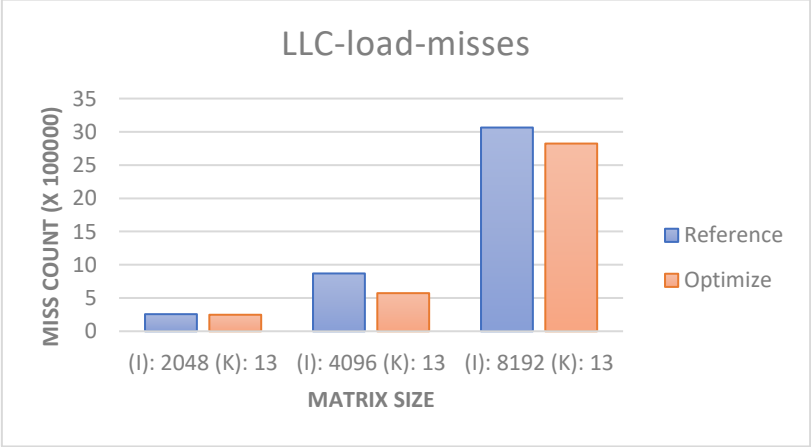
It aims to decrease the computational intensity by substituting high-cost operations, such as multiplication, with lower-cost operations like addition or shift operations. This optimization strategy is employed to enhance the efficiency and performance of the code.

2. Observation with Single Thread Optimization

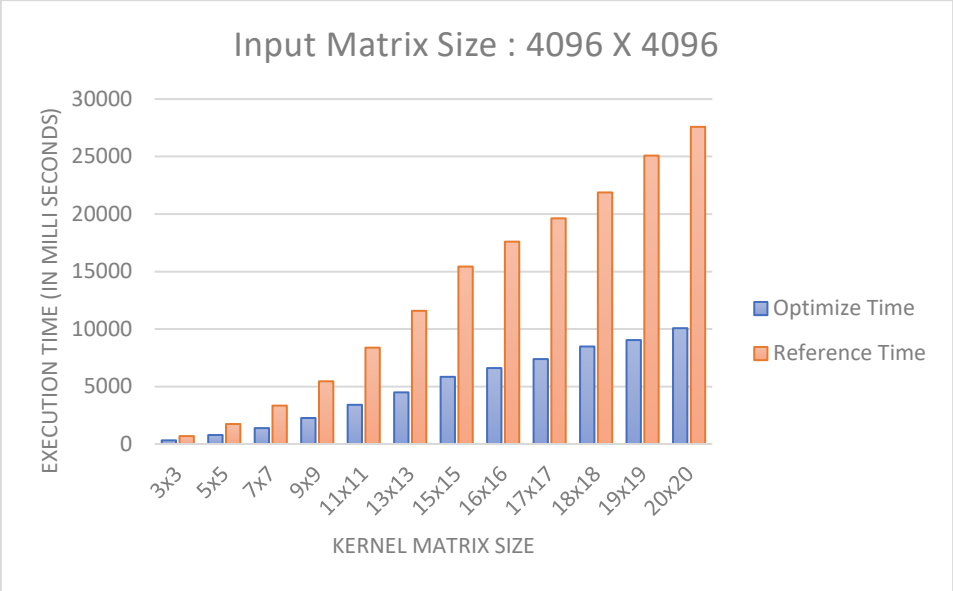
Upon conducting an analysis of both the naive implementation and the optimized single-threaded code using the **perf** tool, the obtained results are as follows.

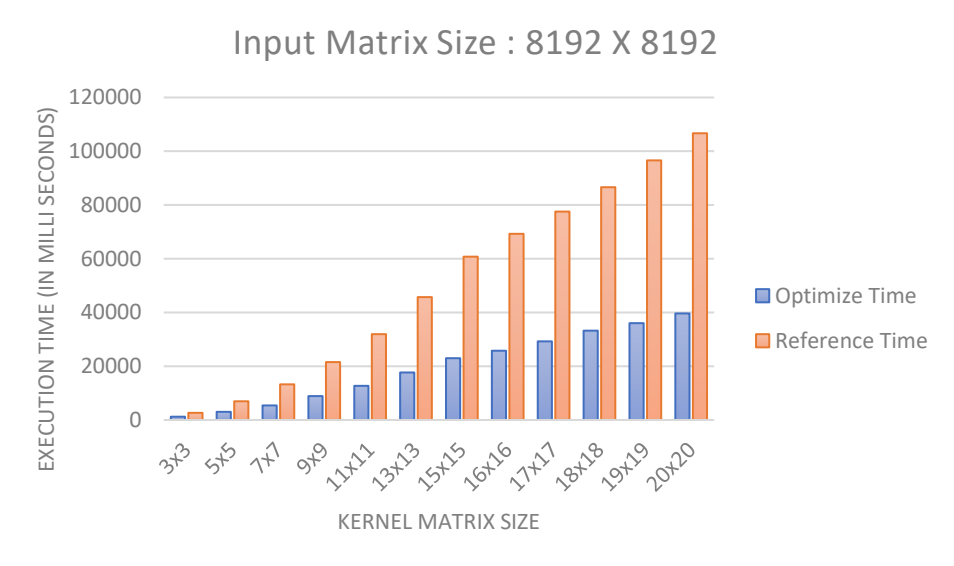


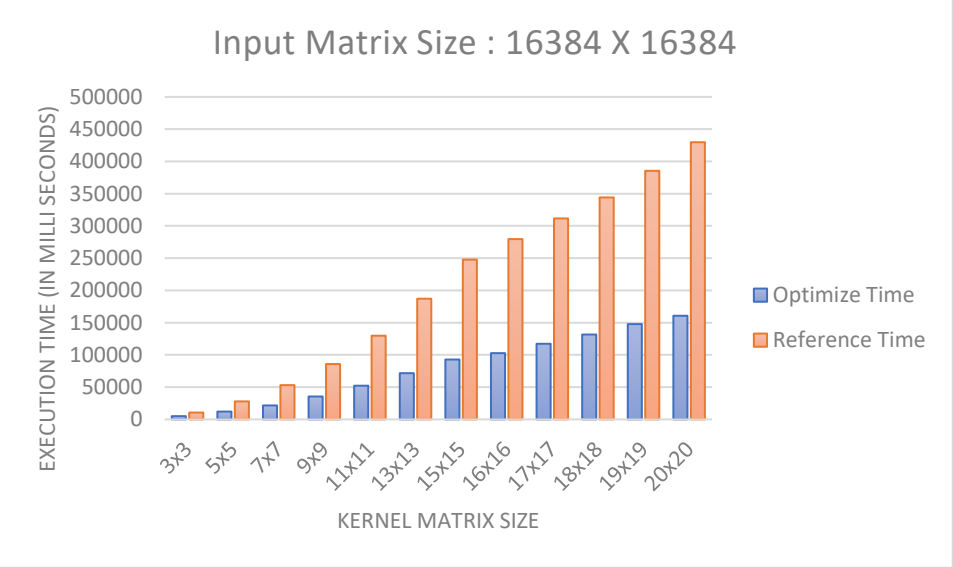




Execution Time for different size of Input matrix and Kernel matrix







The analysis of the optimized single-threaded code reveals significant improvements in execution time compared to the naive implementation across various kernel matrix sizes. As evidenced by the results, the optimized code consistently outperforms the reference implementation, showcasing its effectiveness in enhancing computational efficiency. Notably, the speedup is more pronounced for larger kernel matrix sizes, with a substantial reduction in execution time observed. This performance boost is indicative of the success of the applied optimization techniques, illustrating their impact on the overall responsiveness of the code. The results demonstrate a clear advancement in the optimized code's ability to handle computationally intensive tasks, making it a preferable choice for scenarios where efficiency is paramount.

From the graphs, it is observed that we get the best speed-up with respect to Input matrix size 16384, that is approximately 3.

Input Matrix Size	*Kernel Matrix Size*	*Speed-up*
4096 X 4096	13 X 13	2.987
8192 X 8192	13 X 13	3.014
16384 X 16384	13 X 13	3.152

Part A(II):

1. Optimizing the DC with Multithreading

1.1 Why do we need threading?

Threading is crucial in software development for several reasons. Firstly, it enables concurrent execution, allowing different parts of a program to run simultaneously, leading to improved system performance and responsiveness. Threading also facilitates parallelism, leveraging the processing power of multi-core hardware for faster execution of computationally intensive tasks. Additionally, threading enhances the efficiency of resource utilization, supports asynchronous operations for non-blocking execution, and contributes to code modularity and maintainability. It is essential for background processing, enabling tasks to run independently without disrupting the main program flow. Overall, threading is indispensable for optimizing application performance, responsiveness, and scalability in a wide range of computing scenarios.

1.2 How to Optimize code using multi-threading?

Optimizing code using multithreading involves parallelizing computations to make more efficient use of available CPU resources.

Identify Parallelizable Sections:

Examine the code to identify sections that can be executed independently or concurrently. Look for loops, data processing tasks, or computations that do not have dependencies on each other.

Divide Workload:

Divide the workload into smaller tasks that can be executed in parallel. Consider the nature of the computations and data dependencies when determining how to split the workload.

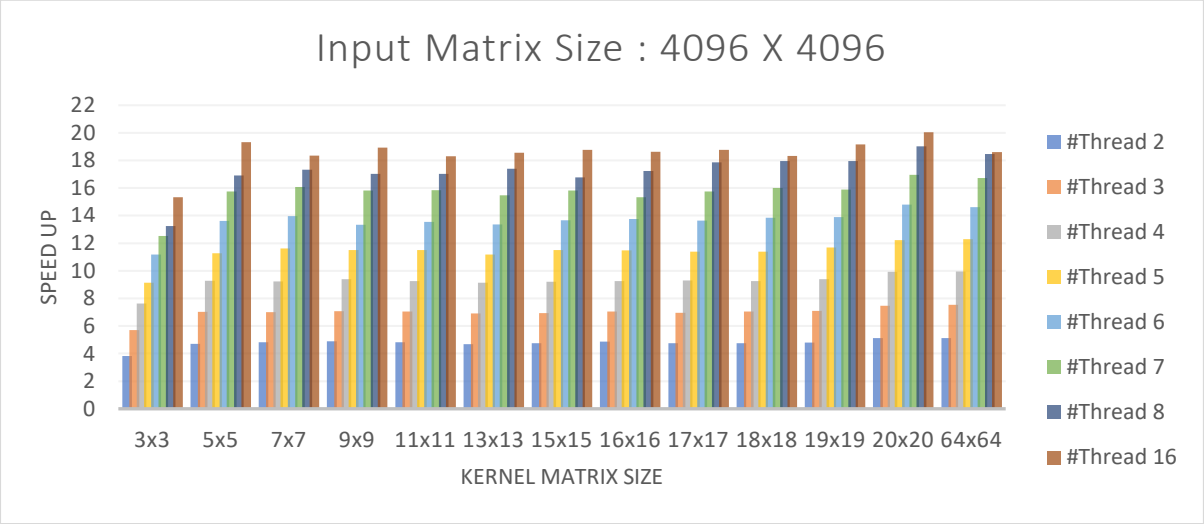
Thread Creation:

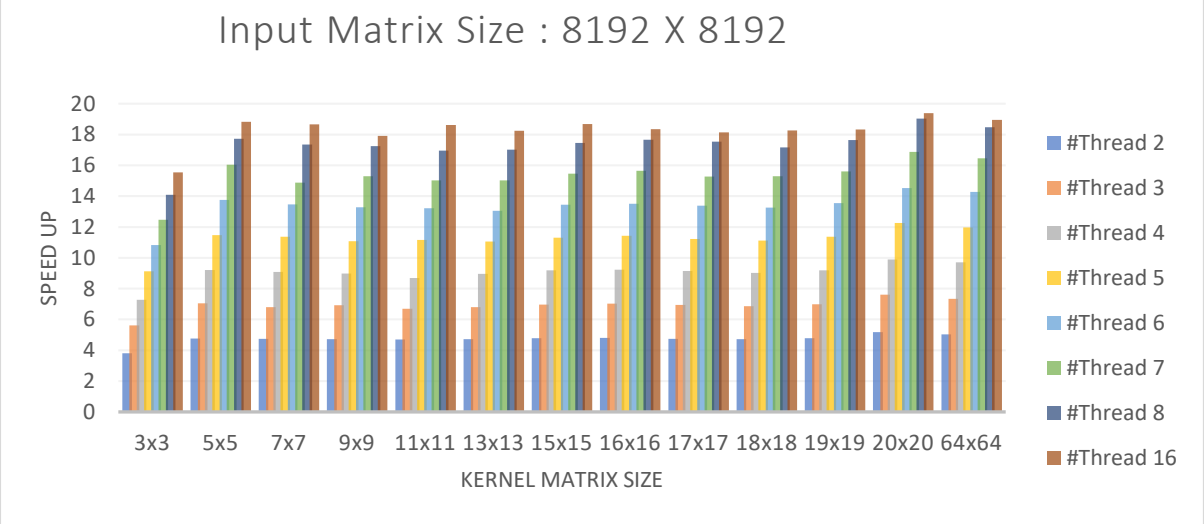
Create threads to handle the parallel tasks. The number of threads created should be based on the available hardware resources (e.g., the number of CPU cores). Use a thread pool or a similar mechanism to manage thread creation and reuse, minimizing the overhead of thread creation and destruction.

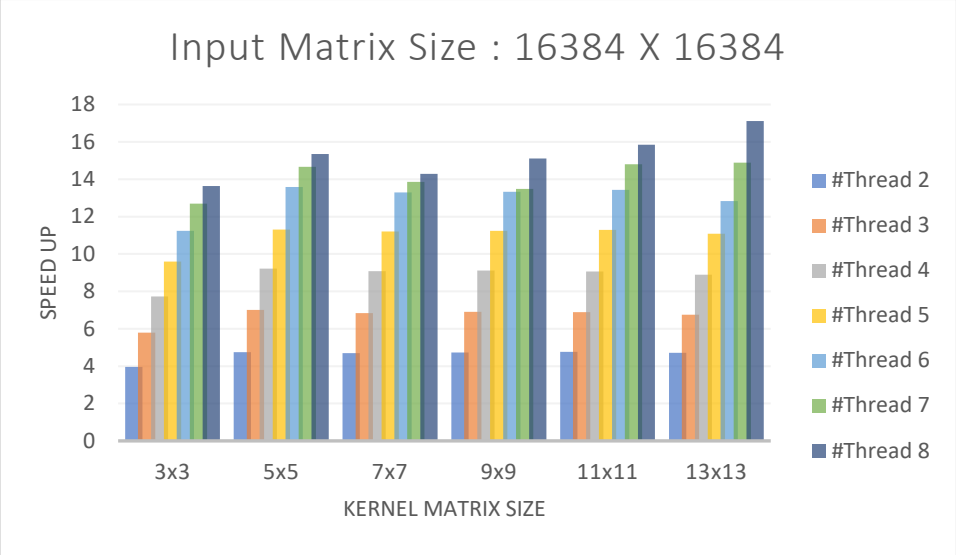
In the multithreading code, the workload is divided by allocating distinct ranges of output rows to individual threads. Each thread independently calculates a portion of the output matrix, which enables concurrent execution and parallel processing of the convolution, and these calculations can be done simultaneously. This parallelization can lead to significant speedup, especially on multi-core processors, where multiple threads can execute concurrently.

However, it's important to note that the effectiveness of multithreading depends on various factors, including the size of the input matrices, the number of available CPU cores, and the overhead associated with thread creation and synchronization.

1.3 Observation and Analysis with Multi-Threading







- Multithreading unveils diverse speedup dynamics in the context of a 4096x4096 input, where larger kernels, notably 64x64, consistently exhibit enhanced performance. This trend suggests that the benefits of parallelization are particularly pronounced for computationally intensive tasks.
- As the input size expands to 8192x8192, the advantages of multithreading become more evident across various kernel sizes, highlighting its scalability, especially for larger kernels. This pattern persists for a 16384x16384 input, indicating that larger kernels consistently yield superior speedup. However, achieving linear speedup might encounter challenges due to potential resource limitations or contention.
- The influence of kernel size emerges as crucial, with larger kernels facilitating more effective workload distribution among threads, thereby enhancing parallel processing efficiency. It is noteworthy that the optimal number of threads may vary, emphasizing the necessity to fine-tune the parallelization strategy based on specific input and kernel characteristics.
- Smaller input sizes may not consistently benefit from multithreading, signifying that the advantages become more prominent in larger and computationally demanding scenarios. In convolution tasks involving expansive input sizes and kernels, multithreading emerges as particularly effective.
- The improvement in performance (decrease in time) is not uniform as the number of threads increases. There is a diminishing return, where the reduction in time becomes less significant with higher thread counts.
- The diminishing returns observed as the number of threads increases can be attributed to factors such as overhead, contention, and the inherent characteristics of the problem being parallelized. As the number of threads grows, the coordination and communication between threads can lead to increased overhead, which partially offsets the gains achieved through parallelization. Additionally, contention for shared resources and potential bottlenecks in the system can limit the scalability of parallel execution.

Perf Analysis:

Upon conducting an analysis of Naive implementation, Optimized single-threaded code and Optimized Multi-threaded code using the **perf** tool, the obtained results are as follows.