E0243: High Performance Computer Architecture
Assignment 2 (Part A)

Yash Patel
SR - 22759
yashj@iisc.ac.in

Anjali Chauhan
SR - 22703
anjalic@iisc.ac.in

November 26, 2023

Github Repository : https://github.com/yash8071/CA Assignment 2.

OPTIMIZING PERFORMANCE OF DILATED CONVOLUTION

Abstract

This project aims to optimize the Dilated Convolution (DC) algorithm with a particular emphasis on improving the method's time and space complexity performance. The optimization methodologies used, the reasoning behind each optimization, and the improvements in execution time and memory utilization that resulted are all described in depth in the paper.

Introduction

The optimization of the Dilated Convolution (DC) method, a popular convolution operation variant in deep learning and image processing, is the focus of this assignment. The main goal is to maximize the convolution process's efficiency while accounting for the dimensions of the input matrix and kernel matrix.

This project's main objective is to optimise the DC algorithm, with particular attention paid to parallelization, vectorization, and memory and algorithmic optimisations. The project's goal is to improve the unoptimized implementation's performance, which will result in calculations that are quicker and more effective.

Dilated Convolution

Dilated Convolution involves sliding a kernel matrix over an input matrix to produce an output matrix. The dimensions of the output matrix are determined by the sizes of the input and kernel matrices.

Calculation of Output Matrix Dimension

The output matrix dimensions are calculated using the formulas:

$$\text{Output\_Row} = \text{Input\_Row} - \text{Kernel\_Row} + 1$$

$$\text{Output\_Column} = \text{Input\_Column} - \text{Kernel\_Column} + 1$$

Visualisation of Algorithm

Dilated Convolution involves sliding a kernel matrix over an input matrix to produce an output matrix. The dimensions of the output matrix are determined by the sizes of the input and kernel matrices.

The Input Matrix (I) is traversed by the Kernel Matrix (K), generating each cell of the Output Matrix (O) as depicted in the following diagrams. In these illustrations, the leftmost element is the Kernel Matrix (K), the middle element is the Input Matrix (I), and the rightmost element is the Output Matrix (O).

The product of the elements of matrices K and I is calculated and accumulated to determine the corresponding cell in the Output Matrix (O).

For example :

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

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

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

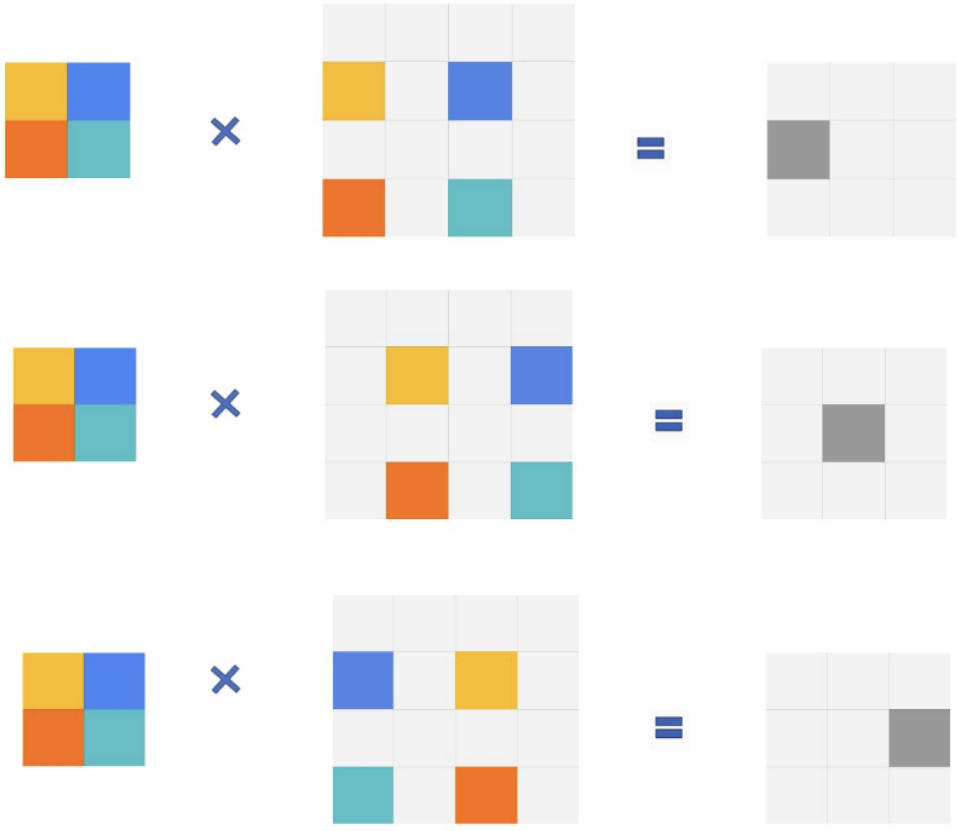

Figure 1: Visualisation of Algorithm

Processor Specification

We used a system with the given specifications to execute our code and perform optimization.

CPU	AMD Ryzen 7 6800H
Memory	16 GB
Cache	L1-d cache = 256KB L1-i cache = 256KB L2 cache = 4MB L3 cache = 16MB
OS	Ubuntu 22.04.1
Core	16
Threads	2 per Core

Table 1: Processor Specification Details

[Part A-I] Optimizing Single-Threaded DC (CPU)

Why do we need to optimize this?

- The inefficiencies of the naive convolution technique result in frequent TLB misses, L1 and L2 cache misses, and inadequate memory use. Its inconsistent memory access patterns lead to subpar cache localization.
- Code optimisation is essential to minimise L1 and L2 cache misses and reduce TLB misses by improving temporal and spatial locality and restructuring memory access patterns.
- The optimised code minimises redundant calculations and improves data access patterns in an effort to increase memory efficiency.
- These enhancements lower the total memory footprint and increase memory bandwidth utilisation, which results in a more effective use of memory. Furthermore, time efficiency is increased via optimisations such as loop unrolling, which causes the convolution operation to execute more quickly.

All things considered, improving the naive algorithm is crucial for resolving performance issues associated with TLB misses, cache misses, and memory inefficiencies.

What have we done for optimization?

1. **Loop Unrolling:** Processing several elements in each inner loop iteration is known as loop unrolling. The loop is unrolled in this instance by a factor of 2, combining two of the initial loop's iterations into one. Because there is less overhead associated with loop control, performance may improve.

```
//Original Code
for(int kernel_j = 0; kernel_j< kernel_col; kernel_j++)

//Optimized Code
for(int kernel_j = 0; kernel_j< kernel_col; kernel_j+=2)
```

Figure 2: Loop Unrolling Code Snippet

2. **Vectorization:** The SIMD (Single Instruction, Multiple Data) instructions are intended to be utilised by the optimised code. A SIMD processor enables the processor to operate on several data components at once. Here, a single vectorized operation replaces the eight separate operations.

```
//Original Code
output[output_i * output_col + output_j] +=
input[input_i*input_col +input_j] * kernel[kernel_i*kernel_col +kernel_j];


//Optimized Code
output[y + output_j] += input[z + (output_j + kernal_j_into_2) % input_col] * kernel[x];
output[y + output_j+1] += input[z +(output_j+1 + kernal_j_into_2) % input_col] * kernel[x];
// ... (similar pattern for output_j+2 to output_j+7)
```

Figure 3: Vectorization Code Snippet

3. **Efficient Memory Access:** The optimized code precomputes the index ‘z‘ outside the innermost loop, eliminating redundant calculations and improving memory access patterns. This helps in reducing the overhead associated with repeated modulo operations.

```
//Original Code
int input_i = (output_i + 2*kernel_i) % input_row;
int input_j = (output_j + 2*kernel_j) % input_col;

//Optimized Code
int z = (output_i + (kernel_i<<1)) % input_row * input_col;
```

Figure 4: Memory Access Code Snippet

4. **Efficient Arithmetic Operations:** The optimized code performs certain arithmetic operations outside the loop, reducing redundant calculations and enhancing efficiency. Left-shifting is used instead of multiplication by 2.

5. **Memory Access Pattern:** The optimized code uses a more efficient memory access pattern by utilizing the precomputed variable ‘y‘. This eliminates the need for repeated multiplication (output_i * output_col) within the innermost loop.

```
//Original Code
output[output_i * output_col + output_j] += ...;

//Optimized Code
output[y + output_j] += ...;
```

Figure 5: Memory access pattern example

Analysis

We have run our single-threaded optimized code for three input matrices, and for each input matrix, we have calculated the execution times of the reference code and optimized code for various kernel sizes. Below are graphs representing the execution time of single-threaded code with the given reference code.

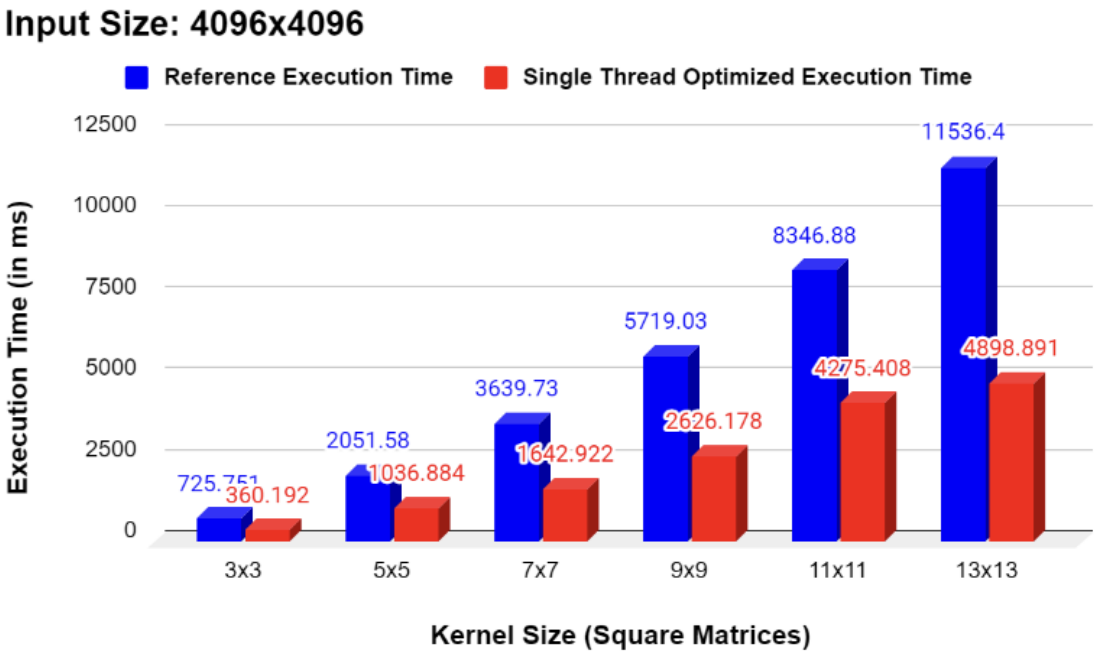

Figure 6: Execution Time Analysis with Input matrix size 4096 x 4096

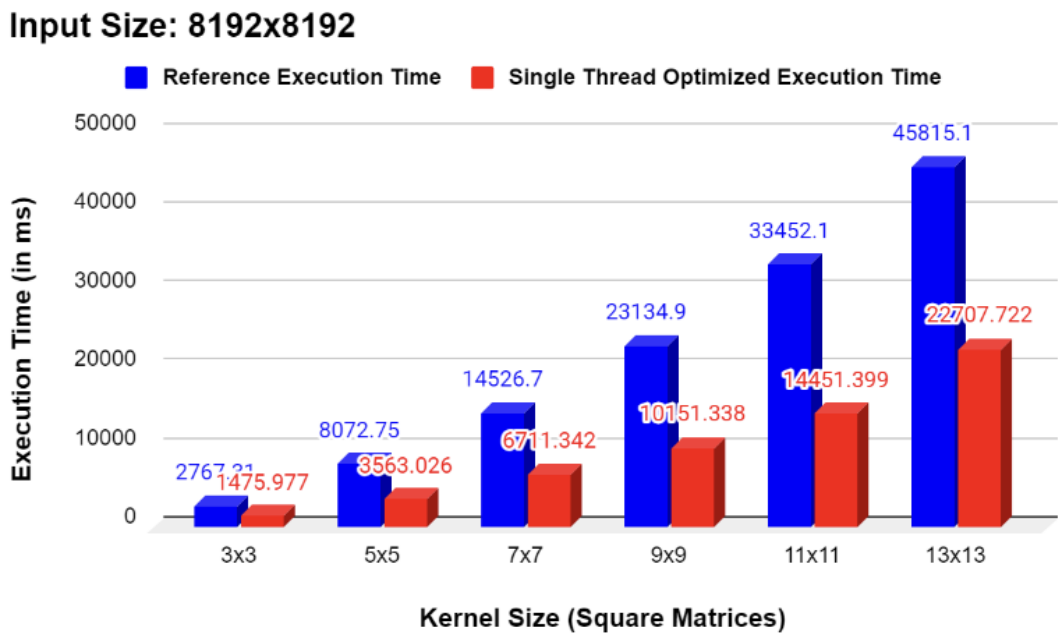

Figure 7: Execution time Analysis with Input matrix size 8192 x 8192

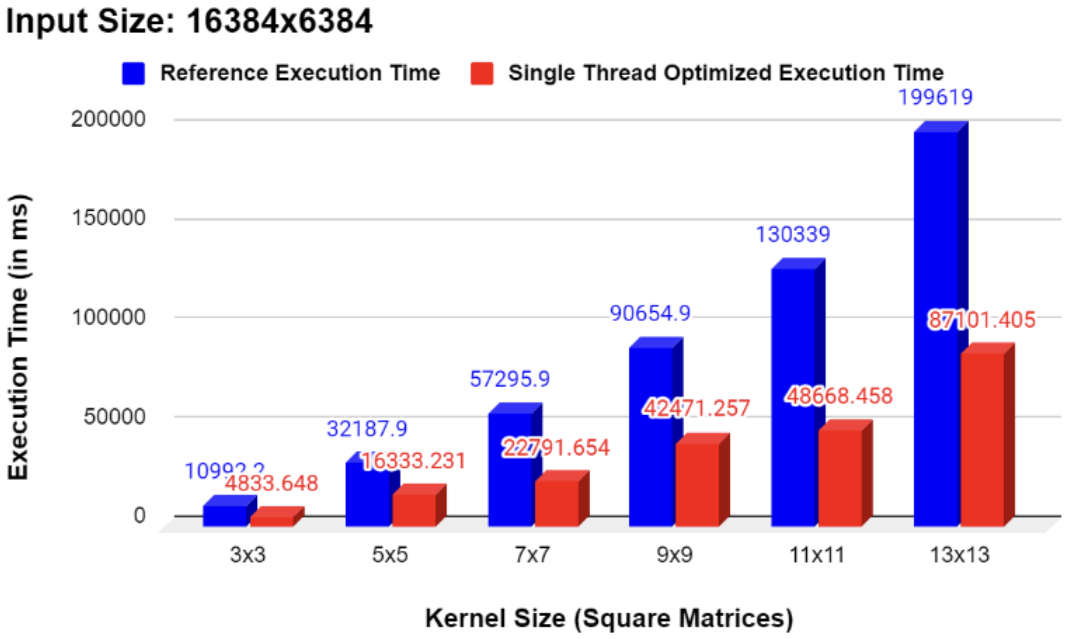

Figure 8: Execution time Analysis with Input matrix size 16384 x 16384

The optimization consistently improves performance across diverse input matrix sizes (8192x8192, 16384 x 16384) and kernel dimensions (3x3 to 64x64). Larger input matrices generally result in increased speedup, indicating scalability. The optimization is particularly impactful for larger kernels, showcasing its efficacy in handling computationally intensive operations. Overall, significant time reductions are achieved, with speedup values ranging from approximately 1.87 to 2.68. This consistent pattern underscores the robustness of the optimization strategy. The optimized algorithm's scalability

is evident in its ability to efficiently handle varying computational workloads, making it a versatile enhancement for convolution operations.

[Part A-II] Implementing and Optimizing Multi-Threaded DC (CPU)

Why do we need threading?

Till now CPUs use only one thread for the whole program. So, our CPU cannot utilize full strength. Also, most of the threads in CPU remain ideal in this naïve approach. For better optimization we implement thread in our code.

Multithreading

Multithreading is the method to run different parts of a program in parallel across different cores, ensuring parallelism. It does this through using the help of threads which are nothing but lightweight processes with the process.Modern processors often have multiple cores, and multithreading helps utilize these cores efficiently. A thread can run on different cores leading to maximum utilization of CPU.Each thread can be assigned to a different core, maximizing the computational power available. The advantages of using multithreading :

1. **Resource Sharing:** There are two ways for processes to share resources either Message passing or shared memory. This has to be done explicitly by a programmer. Therefore, the benefit of sharing data allows processes to have several threads within the same address space.

2. **Scalability:** The benefits of multithreading excel in multiprocessor architecture. Multi-threading on multiple cores increases the parallelism.

3. **Responsiveness:** Let's say multithreading is not implemented on a system which receives a request, performs a set of tasks and returns it sequentially. If a long process comes, the system would be busy doing that and make other requests wait. While in case of multithreaded system may allow run a program even if a part of is blocked or doing length operation, thus increasing responsiveness.

What have we done for Multithreading?

Multithreading is the method to run different parts of a program in parallel across different cores, ensuring parallelism. It does this through using the help of threads which are nothing but lightweight processes with the process.Modern processors often have multiple cores, and multithreading helps utilize these cores efficiently. A thread can run on different cores leading to maximum utilization of CPU.Each thread can be assigned to a different core, maximizing the computational power available. The advantages of using multithreading :

- To optimize the code, we divide the code into threads such that a given thread runs in parallel for better performance.

- The total number of output rows (outputRows) is divided among the threads.

- rowsPerThread is calculated as the integer division of outputRows by the number of threads (NUM_THREADS), and any remaining rows are distributed among the first few threads by incrementing their assigned rows by 1.

- By creating multiple threads, each responsible for computing convolution on a subset of output rows, the code achieves parallel execution.
- Different threads operate independently and concurrently, potentially utilizing multiple CPU cores for faster computation.
- The workload (output rows) is distributed among threads in a way that attempts to balance the computation evenly among them. The rowsPerThread value is used to determine the number of rows each thread should process.

Analysis

We have run our multithreaded code for three input matrices and for each input matrices we have calculated execution time of reference code and multithreaded code for various kernel sizes and with different number of threads. Below are graphs representing performance on speed up of multithreaded code with given reference code.

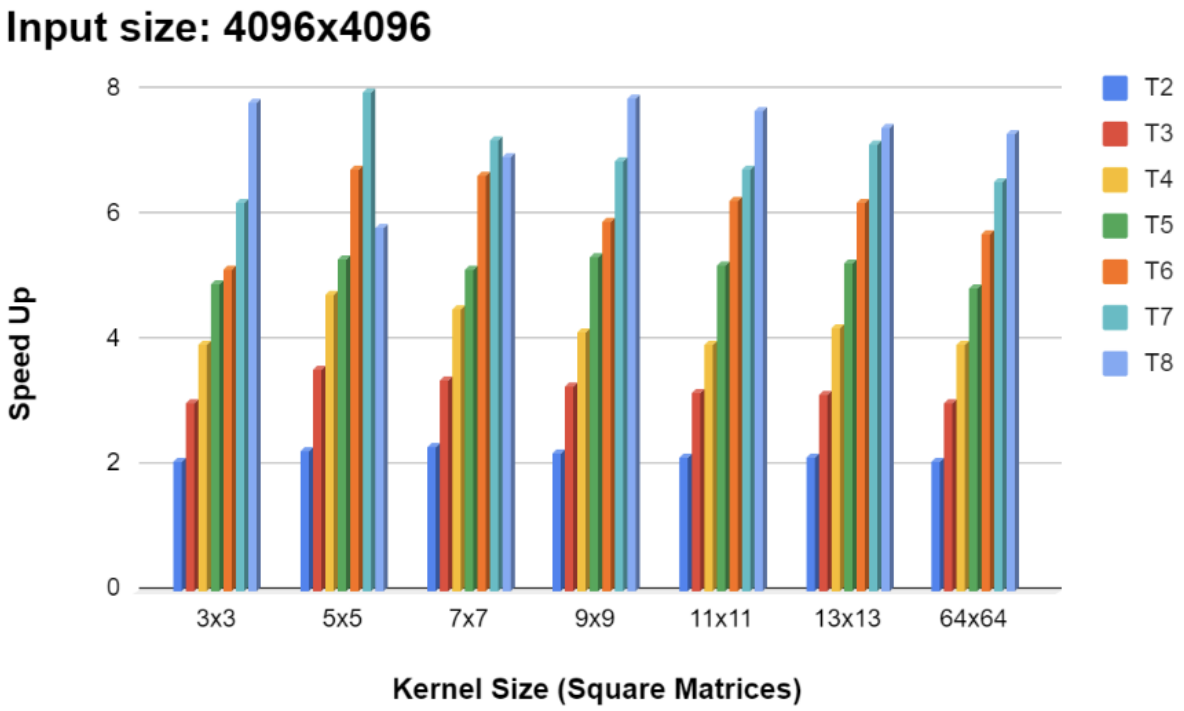

Figure 9: Speed up Analysis with Input matrix size 4096 x 4096

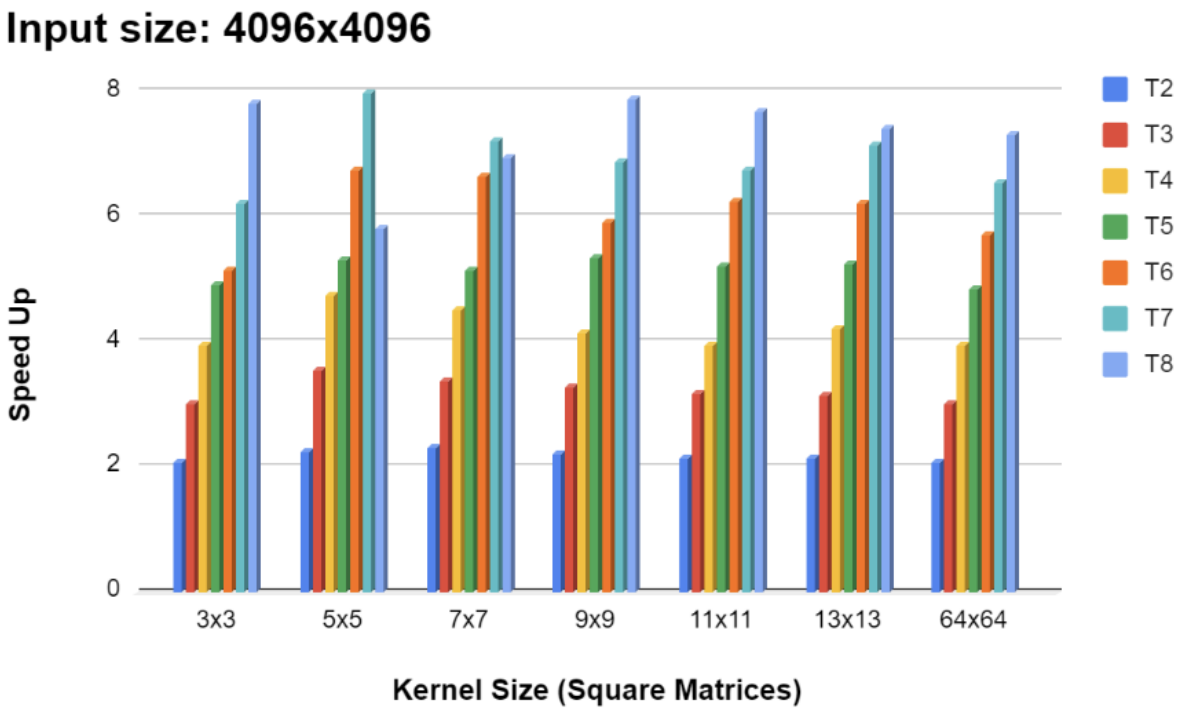

Figure 10: Speed up Analysis with Input matrix size 8192 x 8192

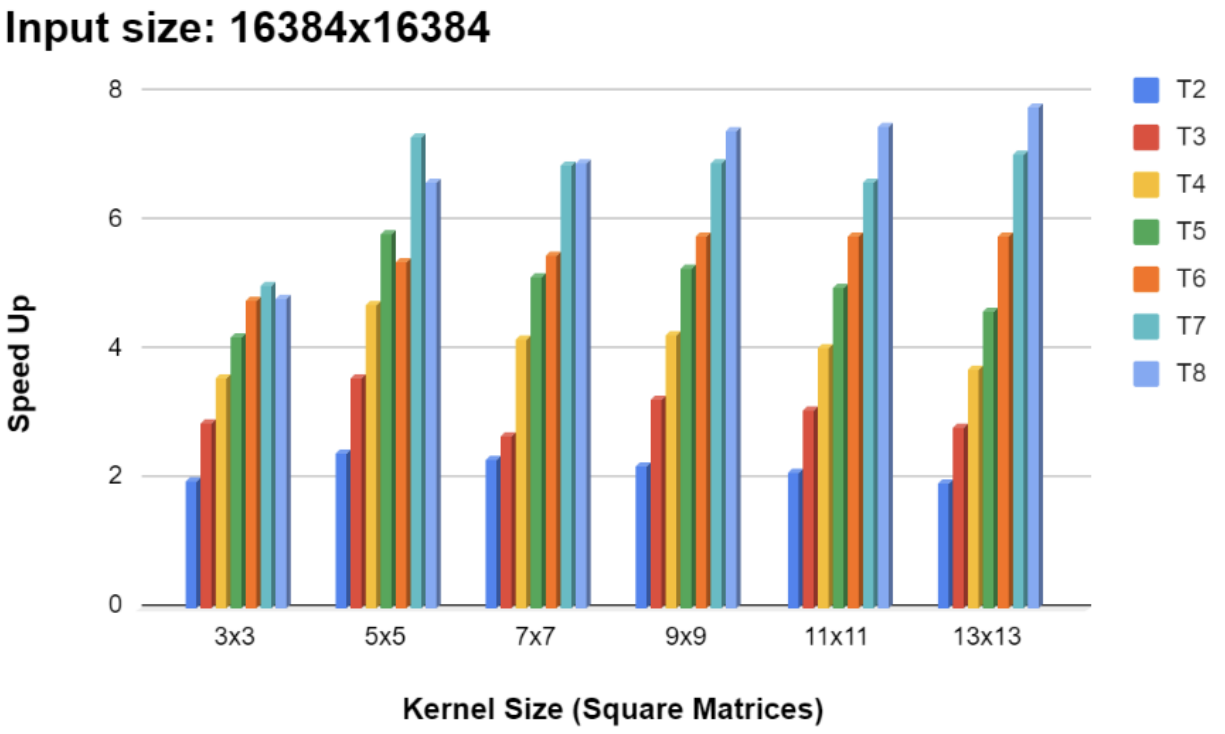

Figure 11: Speed up Analysis with Input matrix size 16384 x 16384

- Multithreading demonstrates varying speedup for a 4096x4096 input, with larger kernels, especially 64x64, consistently showing improved performance, suggesting parallelization benefits for

computationally intensive tasks.

- As input size increases to 8192x8192, the advantages of multithreading become more pronounced across various kernel sizes, emphasizing scalability, especially for larger kernels.The trend persists for a 16384 x 16384 input, indicating that larger kernels consistently exhibit better speedup. However, linear speedup may not always occur due to potential resource limitations or contention.

- As input size increases to 8192x8192, the advantages of multithreading become more pronounced across various kernel sizes, emphasizing scalability, especially for larger kernels.The trend persists for a 16384 x 16384 input, indicating that larger kernels consistently exhibit better speedup. However, linear speedup may not always occur due to potential resource limitations or contention.

- The influence of kernel size is crucial, with larger kernels facilitating better workload distribution among threads, leading to enhanced parallel processing efficiency.

- The optimal number of threads may vary, emphasizing the need to tune the parallelization strategy based on specific input and kernel characteristics.

- Smaller input sizes may not consistently benefit from multithreading, indicating that advantages are more prominent for larger and computationally demanding scenarios.

- Multithreading is particularly effective for convolution tasks involving larger input sizes and kernels.

- The observed trends underscore the importance of a nuanced approach to multithreading, considering both input and kernel characteristics to maximize performance gains.

Perf Analysis

We have run our reference code,single threaded optimised code and multithreaded version of it with 8 threads and used PERF for various kinds of Miss analysis. We have performed operation for various input sizes and kernel sizes. Below shown result are rounded average of different kernel sizes with input matrix of 4096x4096 size. We have mainly analyzed Branch misses,L1-I cache misses and d-TLB misses.

The multithreaded implementation with 8 threads demonstrates superior performance compared to the reference and single-threaded optimized versions. It notably reduces dTLB load misses and branch mispredictions, indicating efficient data translation and better instruction flow. However, the single-threaded optimized code excels in minimizing L1-I cache misses, showcasing effective utilization of the instruction cache. Overall, the multithreaded approach appears favorable for parallelization benefits, but considerations for specific application requirements and hardware characteristics should guide the choice between single-threaded and multithreaded optimizations.

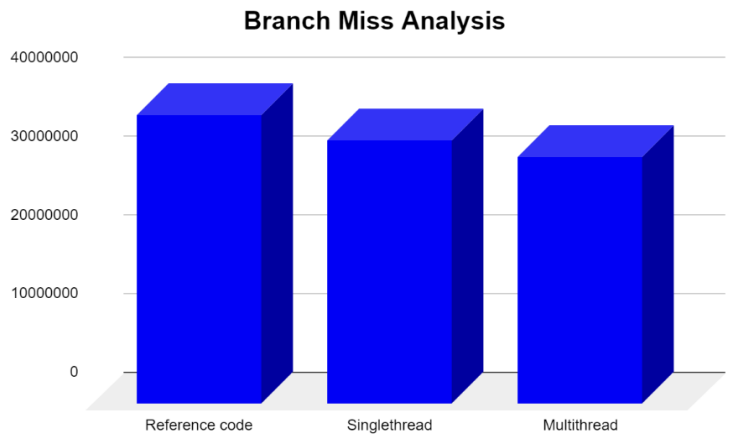

Figure 12: Branch misses analysis

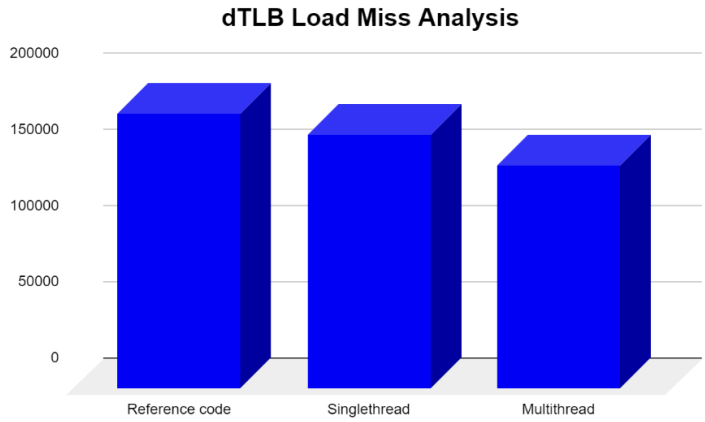

Figure 13: dTLB Load miss analysis

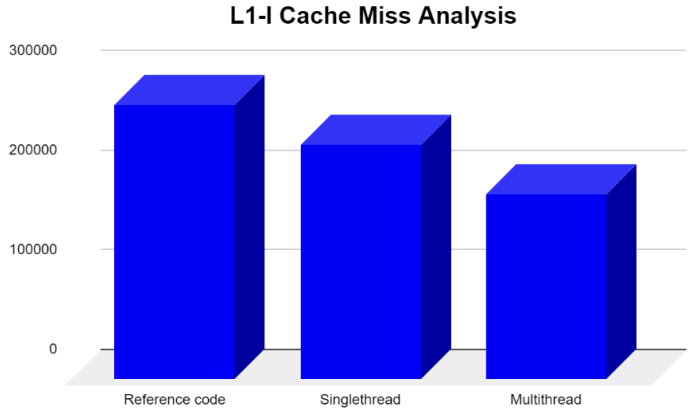

Figure 14: L1-I Cache miss analysis