**Parallel OpenMP**

**Convolution layer:**

The convolution is being parallelized using OpenMP by dividing the outer loop over the filters into multiple chunks and processing each chunk in parallel using multiple threads. The #pragma omp parallel directive creates a team of threads, and the #pragma omp for schedule(static) directive divides the loop iterations equally among the threads, with a static scheduling policy.

Inside the parallel region, each thread iterates over a subset of the filters. For each filter, it loops over the output tensor's spatial dimensions and maps each output element to a corresponding region in the input tensor. Then, it applies the filter to that region using nested loops over the filter's spatial dimensions and the input tensor's depth dimension. Finally, it accumulates the products of the filter values and input values to produce the output value for the current output element.

By parallelizing the outer loop over the filters, OpenMP allows multiple filters to be processed simultaneously by different threads, which can significantly speed up the convolution operation.

The OpenMP parallelization in the given code is an example of data partitioning, also known as data parallelism. Data partitioning is a technique in parallel computing where a large data set is divided into smaller subsets, and each subset is processed in parallel by different threads or processes.

Overall, this approach is an example of data parallelism because the input data is partitioned into smaller subsets, and each subset is processed independently in parallel by different threads.

**Fully Connected Layer:**

In the fully connected layer , OpenMP parallelizes the outer loop over the output tensor elements using the #pragma omp parallel for directive. This directive divides the iterations of the loop into chunks and assigns each chunk to a thread to execute in parallel. The chunk size is determined automatically by OpenMP at runtime based on the number of iterations and the number of threads in the team.

Additionally, OpenMP parallelizes the inner loop over the input tensor elements using the #pragma omp parallel for reduction(+:inputv) directive. This directive indicates that the inputv variable should be treated as a reduction variable, meaning that each thread maintains a private copy of inputv and combines its value with the values of the other threads using an operator (in this case, addition) to produce a final result.

By parallelizing both the outer and inner loops, OpenMP enables the computation for each output tensor element to be performed in parallel across multiple threads, with each thread computing a subset of the input tensor elements and contributing to the final result through the reduction operation. This can help improve the performance of the fully connected layer by taking advantage of parallel hardware and reducing the overall execution time.

OpenMP was used to parallelize the activate() function using data partitioning.

The activate() function contains two nested loops, where the outer loop iterates over the output dimensions, and the inner loop performs the matrix multiplication of the inputs and weights to compute the input values for each output neuron.

To parallelize the outer loop using data partitioning, OpenMP's pragma omp parallel for directive was used. This divides the iterations of the outer loop into multiple parallel tasks that can be executed concurrently by multiple threads. Each thread is assigned a subset of the iterations to process.

To parallelize the inner loop using data partitioning, another OpenMP directive was used: pragma omp parallel for reduction(+:inputv). This directive indicates that the variable inputv should be a reduction variable, and it specifies that the operation to be used for the reduction is addition (+). The reduction operation ensures that each thread maintains a private copy of the variable inputv, and the final result is obtained by summing up the values computed by all the threads.

By using data partitioning in this way, the computations required for each output neuron can be processed independently, and the workload is evenly divided among the threads. This helps to achieve parallelism and can lead to significant speedup in the execution time of the code.