Optimizing Performance Of Dilated Convolution

Anishka Ratnawat
Computer Science and Automation
IISc, Bangalore, India
anishkar@iisc.ac.in

Snigdha Shekhar
Computer Science and Automation
IISc, Bangalore, India
snigdhas@iisc.ac.in

Abstract

Parallel processing aims to concurrently execute multiple parts of a task when two or more CPU cores are used to handle separate parts of the task. Unlike a single core processor where a maximum of only one thread can be executed at any given time, parallel processing gives a significant performance enhancement due to concurrent execution.

In this exercise, we have to optimise the performance of dilated convolution. This type of convlution operation is a variant of convolutional operations used in deep learning, particularly in the context of convolutional neural networks (CNNs). The term "dilated" refers to the introduction of gaps, or dilations, between the values of the filter kernel during the convolution operation.

Our aim is to analyse the performance statistics of the program that performs dilated convolution using hardware performance counters using the *perf* utility in Linux and apply concepts from thread level parallelism in order to optimise the code by identifying the bottlenecks in the program, effectively reducing run time by splitting the work across all cores in the CPU.

I. Introduction

A. Thread Level Parallelism

Multicore processors consist of more than one independent processing unit (also called a core). While single-core CPUs can only execute instructions sequentially one after another, multicore CPUs provide increased speed and responsiveness compared to single-core processors as they can execute in parallel as many threads as there are CPU cores. Thread-Level Parallelism exploits the potential of splitting a program into independent parts and running these parts concurrently as "threads". These threads may either run independent programs together or separate blocks inside the same program.

TLP provides each thread with its own copy of the Program Counter (PC), register files, etc. Multithreading improves the throughput of computers that are running multiple programs, and the execution time of multi-threaded programs is decreased.

B. Dilated Convolution

For the purpose of this exercise, we were asked to optimize the code for dilated convolution. In a regular convolution operation, a kernel or filter of a fixed size slides over the input feature map, and we multiply the values in the kernel with the corresponding values in the input feature that the kernel maps to in order to produce a single output value. In the case of dilated convolution, the dilation rate effectively increases the receptive field of the kernel. This is due to the fact that we introduce gaps between the values without increasing the size of the kernel or the number of parameters. This can be useful in situations where a larger receptive field is needed, but increasing the size of the kernal would lead to an increase in the number of parameters and computational complexity.

The dilation rate d is a hyperparamater that determines the size of these gaps. In other words, $(d-1)$ pixels are skipped in the kernel, based on the value of this hyperparameter. On keeping the value of $d = 2$, we skip 1 pixel while mapping the kernal onto the input, thus covering more information in each step.

Mathematically, the dilated convolution operation can be expressed as

$$(I_{*d}k) = \sum_{(\ s+dt=p)} I(s) * k(t)$$

Here, I is the input data, k is the filter/kernel, $*$ denotes the convolution operation, and d is the dilation factor.

II. Methodology

A. Analysing performance bottleneck using ***perf***

1) System Configuration: The system on which we ran our code had the following configuration.

Generation	11th Generation Intel® Core™ i5
Cores	12
Base Frequency	2.70 GHz
Cache	12 MB

TABLE I: Processor Configuration

2) Bottlenecks: We ran the unoptimized version of the code snippet below and observed that the following performance monitoring counters were significantly hindering the overall performance of the program on matrices of sizes 2048 and 4096 respectively.

```
for(int i = 0; i < output_row * output_col; ++i)
        output[i] = 0;

    for(int output_i = 0; output_i< output_row; output_i++)
    {
        for(int output_j = 0; output_j< output_col; output_j++)
        {
            for(int kernel_i = 0; kernel_i< kernel_row; kernel_i++)
            {
                for(int kernel_j = 0; kernel_j< kernel_col; kernel_j++)
                {
                    int input_i = (output_i + 2*kernel_i) % input_row;
                    int input_j = (output_j + 2*kernel_j) % input_col;
                    output[output_i * output_col + output_j] += input[input_i*input_col +input_j]
                                                                * kernel[kernel_i*kernel_col +kernel_j];
                }
            }
        }
    }
```

Performance Counter	Value
instructions	88,509,524,420
cache-misses	1,954,172
LLC-store-misses	1,740
dTLB-store-misses	106,975

Performance Counter	Value
instructions	356,416,551,970
cache-misses	8,274,066
LLC-store-misses	554,005
dTLB-store-misses	240,696

TABLE II: Counters for 2048x2048 and 4096x4096

This falls in line with the reasoning that the cache and TLB misses are high, and therefore the memory-bound bottlenecks contribute the most towards the increased runtime. This happens due to the repeated reload of the values from the input and kernel as well as recalculating the indices repeatedly. Following this analysis, we attempted to optimize the code from various angles.

B. Approach 1: Changing the data layout

In this approach, we attempted to change the layout of the input matrix by bringing the consecutively accessed columns together. For example, let us assume that the input matrix was 4x4. We transformed the matrix as follows:

00	01	02	03
10	11	12	12
20	21	22	23
30	31	32	33

=>

00	02	01	03
20	22	21	23
10	12	11	13
30	32	31	33

TABLE III: Transformation applied in Approach 1

The above transformation depicts the process of bringing the consecutively accessed elements (after skipping 1 row /1 column in an iteration) together to minimize data cache misses.

However, as it turns out, this method does not give a significant performance enhancement measured in terms of speedup. The maximum speedup achieved by this method was approximately 1.2x or the performance increase was only 16%. Hence, we move on to the next approach, namely, vectorization.

C. Approach 2: Vectorization

Vectorization is a Single Instruction Multiple Data (SIMD) technique used to optimize code by transforming scalar operations into vector operations, where multiple data elements are processed simultaneously using a single instruction.

We made use of the *__m256i* AVX intrinsics, which is an extension to the x86 instruction set architecture designed to improve performance for floating-point and integer-intensive calculations. We calculated the partial sum in the following manner by using the vectorized *_mm256_add_epi32* and *_mm256_mullo_epi32* instructions to perform 8 integer operations per vector simultaneously.

```
partial_sum = _mm256_add_epi32(sum, _mm256_mullo_epi32(inputVal, kernelVal));
```

However, by using this method as well, the maximum speedup achieved was still not quite high.

D. Approach 3: Optimising code structure

Our final code consists of the following changes.

1. *Loop unrolling*: Loop unrolling involves replicating the body of a loop multiple times within the loop itself, effectively increasing the size of each iteration, which reduces loop overhead. We have performed loop unrolling eight times in order to decrease the runtime of the program.

2. *Reducing ALU operations* : We have tried to save time on ALU operations which occur due to repeated calculations of the input matrix. We are initialising the indices *input_i* and *input_j* by *output_i* and *output_j* respectively and also incrementing them by the dilation factor (2 in this case) whenever *kernel_i* and *kernel_j* are incremented. The code snippet below illustrates the same.

```
for (int output_i = 0; output_i < output_row; ++output_i) {
    int output_row_skip = output_i * output_col;

    for (int output_j = 0; output_j < output_col; output_j+=8) {
        partial_sum = 0; partial_sum_1 = 0; partial_sum_2 = 0;
        partial_sum_3 = 0; partial_sum_4 = 0; partial_sum_5 = 0;
        partial_sum_6 = 0; partial_sum_7 = 0;

        input_i = output_i;

        for (int kernel_i = 0; kernel_i < kernel_row; ++kernel_i) {
            int kernel_row_skip = kernel_i * kernel_col;

            int input_row_skip = input_i * input_col;
            input_j = output_j;

            for (int kernel_j = 0; kernel_j < kernel_col; ++kernel_j) {

                partial_sum += input[input_row_skip + input_j] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_1 += input[input_row_skip + (input_j+1)%input_col] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_2 += input[input_row_skip + (input_j+2)%input_col] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_3 += input[input_row_skip + (input_j+3)%input_col] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_4 += input[input_row_skip + (input_j+4)%input_col] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_5 += input[input_row_skip + (input_j+5)%input_col] *
                kernel[kernel_row_skip + kernel_j];

                partial_sum_6 += input[input_row_skip + (input_j+6)%input_col] *
                kernel[kernel_row_skip + kernel_j];
```

```
                partial_sum_7 += input[input_row_skip + (input_j+7)%input_col] *
                kernel[kernel_row_skip + kernel_j];


                input_j = input_j + dilation;
                if(input_j >= input_col)
                    input_j = input_j % input_col;
            }

            input_i = input_i + dilation;
            if(input_i >= input_row)
                    input_i = input_i % input_row;
        }

        output[output_row_skip + output_j] = partial_sum;
        if(output_j + 1 < output_col)
            output[output_row_skip + output_j + 1] = partial_sum_1;
        if(output_j + 2 < output_col)
            output[output_row_skip + output_j + 2] = partial_sum_2;
        if(output_j + 3 < output_col)
            output[output_row_skip + output_j + 3] = partial_sum_3;
        if(output_j + 4 < output_col)
            output[output_row_skip + output_j + 4] = partial_sum_4;
        if(output_j + 5 < output_col)
            output[output_row_skip + output_j + 5] = partial_sum_5;
        if(output_j + 6 < output_col)
            output[output_row_skip + output_j + 6] = partial_sum_6;
        if(output_j + 7 < output_col)
            output[output_row_skip + output_j + 7] = partial_sum_7;
    }
}
```

We observed the values for performance monitoring counters for our optimized code for table sizes 2048 and 4096 respectively, listed in the tables below.

Performance Counter	Value
instructions	66,409,385,056
cache-misses	1,757,703
LLC-store-misses	542
dTLB-store-misses	38,081

Performance Counter	Value
instructions	266,598,594,250
cache-misses	7,341,072
LLC-store-misses	6,042
dTLB-store-misses	206,146

TABLE IV: Counters for 2048x2048 and 4096x4096 for optimized code

E. Further optimisation: Multithreading

1. pthread library: The pthreads (POSIX threads) library is a set of C programming language threads (also called lightweight processes or tasks) that provide a way for a program to fork or create multiple concurrent execution flows.

a. Detecting the number of cores automatically

```
int num_threads = 8;
const auto cores = std::thread::hardware_concurrency();
```

b. Declaring thread as a struct

```
struct ThreadData {
    int *input;int *kernel;
    long long unsigned int *output;
    int input_row, input_col, output_row, output_col, kernel_row, kernel_col;
    int start_i, end_i; int dilation;
};
```

c. Creating and initializing threads

```
void multiThread( int input_row, int input_col,int *input,
                int kernel_row, int kernel_col, int *kernel,
                int output_row, int output_col,
```

```
                long long unsigned int *output ) {

    if(cores !=0)
        num_threads = cores;
    pthread_t threads[num_threads];
    struct ThreadData threadData[num_threads];

    int rows_per_thread = output_row / num_threads;

    for (int i = 0; i < num_threads; ++i) {
        threadData[i].input = input;
        threadData[i].kernel = kernel;
        threadData[i].output = output;
        threadData[i].input_row = input_row;
        threadData[i].input_col = input_col;
        threadData[i].output_row = output_row;
        threadData[i].output_col = output_col;
        threadData[i].kernel_row = kernel_row;
        threadData[i].kernel_col = kernel_col;
        threadData[i].start_i = i * rows_per_thread;
        if ( i == num_threads - 1 )
            threadData[i].end_i = output_row;
        else
            threadData[i].end_i = ( i + 1 ) * rows_per_thread;
        threadData[i].dilation = 2;
        int partial_sum = pthread_create(&threads[i], nullptr, threadDilatedConvolution, &threadData[i]);
        if (partial_sum != 0) {
            std::cerr << "Error" <<i<< ":" << partial_sum << std::endl;
            exit(EXIT_FAILURE);
        }

    }

    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], nullptr);
    }
}
```

The individual work done by each thread makes use of the same code as that in the single-threaded optimized version.

III. Observations

We ran our code for matrices of sizes 2K, 4K, 8K and 16K for various kernel sizes and computed the speedup and performance improvement % as displayed in the figure below.

Input Matrix	Kernel Matrix	Reference Time (ms)	Single Thread (ms)	Multi-threaded (ms)	Speedup - Single Thread	Speedup - Multi Thread	Performance % - Single Thread	Performance % - Multi Thread
2048	13	2,776.08	1103.73	323.611	2.52	8.58	60.24	88.34
2048	64	60,975.00	23420.4	5961.49	2.60	10.23	61.59	90.22
4096	13	16,509.20	6252.26	1646.54	2.64	10.03	62.13	90.03
4096	64	253,579.00	96412.3	23635	2.63	10.73	61.98	90.68
8192	13	71,939.60	25618.4	6566.96	2.81	10.95	64.39	90.87
8192	64	1,013,150.00	397518	113757	2.55	8.91	60.76	88.77
16384	13	291,947.00	102947	26183.3	2.84	11.15	64.74	91.03
				Average	2.65	10.08	62.26	89.99

Fig. 1: Speedup and performance improvement % of various matrix sizes

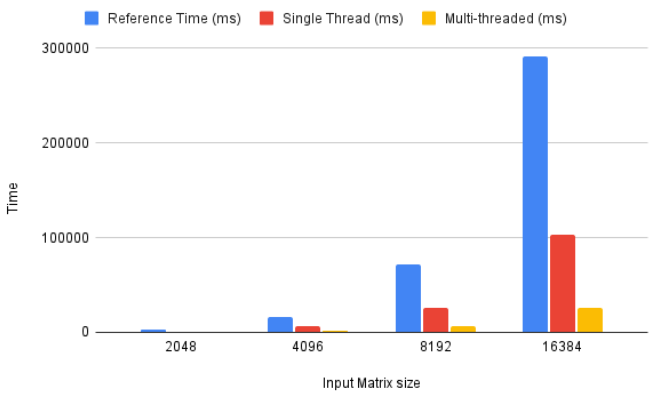

Fig. 2: Reference vs Single thread vs Multi thread

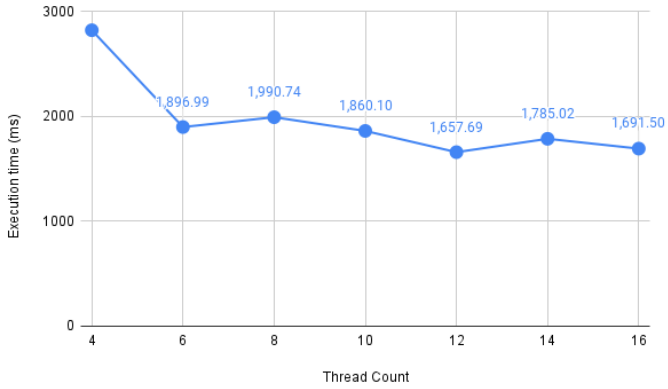

Fig. 3: Execution time vs Thread Count for input size 4096 and kernel size 13

A. *Single Threading*

We first compare the hardware counters obtained by running the unoptimized code with the counter values after running the optimized code after our proposed changes. Hence, we can observe that the values of the events corresponding to the number of instructions, LLC store misses (long latency cache misses), and dTLB store misses reduce significantly. This happens as we have reduced the number of instructions by reducing ALU operations as described in the methodology section. This also reduces long-latency cache misses as well as time spent translating the virtual addresses of these variables into the physical addresses.

Further from Fig. 1, we can deduce that the average speedup of our code for various input matrix sizes turns out to be approximately 2.65x. This means that the performance of our code shoots up by approximately 62.26%. We also see that the speedup (and performance) generally tends to increase with an increase in the input matrix size.

We can also observe from Fig. 2 that as we increase the matrix size, the optimization that we have performed continues to play a significant role in reducing the runtime. This is due to the fact that the runtime decreases by a larger margin as the matrix size is increased.

B. *Multi Threading*

From fig 1, we can see a significant improvement in the runtime, speedup, and overall performance. The average speedup that we could achieve by utilizing all the cores (12 for our processor) was approximately 10.08, and the average performance enhancement was up by 90%. We also see that our multithreaded code gives us increasingly better performance as we increase the matrix sizes. This is due to the fact that the benefit of using multiple threads can be realized better when the input size is larger, as more of these SIMD operations are done in parallel as compared to the unoptimized version.

We also varied the thread counts to establish a relationship between the number of threads used and the runtime of the program. From fig 3, we can observe a decrease in runtime as the number of threads increases. This aligns with our understanding that the work will be divided amongst more cores; hence, more instructions are executed in parallel, and the runtime comes down. However, we also observe a minimum runtime of 12 cores, and increasing any number of cores beyond that does not necessarily improve the runtime. Again, this follows from the concept that maximum performance will occur when all the cores are utilized. Once we cross this threshold, the number of threads being scheduled on the cores is limited, as only one thread can be scheduled on one core at any given point in time. Hence, we do not see a significant increase in performance after 12 threads. Hence, our code for multithreading scales well by increasing the thread count until all cores are utilized.

IV. Conclusion

In this exercise, we have used the concepts of multi-threading and optimized the code using loop unrolling and reducing ALU operations. The speedup and performance enhancement obtained by using these concepts help us understand how parallel execution on multicore processors saves time. Machine learning algorithms often involve computationally intensive operations, and parallelizing these operations across multiple cores can lead to improved performance and faster training or inference times.