# Part A : Optimized Single Threaded DC and Multithreaded DC

Himanshu Devrani
*Computer Science and Automation*
*Indian Institue of Science*
Sr No : 23106
himanshud@iisc.ac.in

Dev Gandhi
*Computer Science and Automation*
*Indian Institue of Science*
Sr No : 22634
devtejas@iisc.ac.in

***Abstract*—Dilated Convolution is one of the widely used operations in machine learning algorithms. It consist an input matrix and a kernel matrix. The input matrix and kernel matrix are operated using some algorithm which involves multiple products and additions between input matrix and kernel matrix cells to produce output matrix cell. Optimizing implementation of single threaded DC and multithreaded DC can help in efficient implementation of various machine learning algorithm.**
**This report presents the an optimized version of single DC implementation using loop optimization,loop Unrolling and Single Instruction Multiple Data instructions. Along with single threaded optimization we will look on the implementation and scalability of multithreaded DC implementation.**

## I. Introduction

Dilated Convolution involves arithmetic operations between input matrix and kernel matrix to produce resultant output matrix. How fast a Dilated Convolution Algorithm will be, is actually dependent upon three factors :- Platform, Size of Input and kernel matrix and Implementation of DC algorithm. So we will fix our platform, pair of input-kernel matrix and then look upon efficiency of DC Algorithm in term of time taken.

### A. *Hardware Configuration*

We have performed this task in**Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz** .

Some Technical Specification are as follows.

- CPU Core : 4
- Architecture : x86−64
- Thread(s) per core : 2
- L1 d-cache : 128 KiB (4 instances)
- L1 i-cache: 128 KiB (4 instances)
- SIMD instruction support: AVX AVX-2

## II. Single Threaded Dilated Convolution Implementation

We are given a basic reference program for single thread implementation which uses four nested loops.By analysing the reference program with help of perf tool we have identified three major bottlenecks in reference program.

**Bottlenecks in reference program :**

- Frequent Cache misses due to nested loops.
- Redundant costly arithmetic operations like multiplication and division.
- High number of loads from L1-D cache due to some redundant loads.

### A. *Optimization 1 : Using Code Motion*

Code motion, a sophisticated compiler optimization technique, plays a pivotal role in enhancing the performance of computer programs. By strategically relocating statements or expressions outside loops, code motion reduces the frequency at which they are executed, leading to improved program efficiency. This optimization approach effectively minimizes redundant computations, optimizes cache utilization, and boosts overall program execution. Using code motion we can easily takle our first two bottle neck of cache utilization and redundant arithmetic operations.We are using below three strategies:-

- Move loop invariant operations like Multiplications from deepest loop to outer loops.
- Replace Remainder operator with Conditional operator where ever possible.
- Replace Multiplications with Addition operator.

TABLE I
Execution Time Comparison between Reference program and Optimized code 1

| Input | Kernel | Reference (ms) | Optimized(ms) | Reduction(%) |
|---|---|---|---|---|
| 4096 | 9 X 9 | 18999 | 9902 | 47.8 |
| 4096 | 11 X 11 | 28043 | 15261 | 45.5 |
| 4096 | 13 X 13 | 40410 | 20583 | 49 |
| 8192 | 9 X 9 | 75423 | 42914 | 43 |
| 8192 | 11 X 11 | 111083 | 63338 | 42 |
| 8192 | 13 X 13 | 155543 | 86014 | 44.7 |
| 16384 | 9 X 9 | 311895 | 182332 | 41.5 |
| 16384 | 11 X 11 | 467432 | 251982 | 46 |
| 16384 | 13 X 13 | 643903 | 343629 | 46.6 |

We can see in Table 1 our optimized code is getting approximately 40 - 45 % reduction in time.Apart from this using hardware counters we can show apparently there was significant reduction in number of cache misses.We will consider Lakh as unit of measure for cache misses.

TABLE II
CACHE MISS COMPARISON BETWEEN REFERENCE PROGRAM AND OPTIMIZED CODE 1

| Input | Kernel | Reference (lac) | Optimized(lac) |
|---|---|---|---|
| 4096 | 9 X 9 | 54.03 | 16.06 |
| 4096 | 11 X 11 | 57.58 | 36.54 |
| 4096 | 13 X 13 | 116.32 | 19.17 |
| 8192 | 9 X 9 | 342.40 | 70.50 |
| 8192 | 11 X 11 | 332.27 | 90.63 |
| 8192 | 13 X 13 | 223.59 | 76.781 |
| 16384 | 9 X 9 | 996.74 | 410.11 |
| 16384 | 11 X 11 | 987.94 | 555.16 |
| 16384 | 13 X 13 | 1315.58 | 364.70 |

Above we can see there is significant reduction in cache misses which results in speed up in execution.Hence we have shown how Code Motion can be beneficial in cache miss reduction.

## B. *Optimization 2: Loop Unrolling and SIMD instructions*

In first optimization we have successfully tackled two of our three bottlenecks for DC. Now the remaining bottleneck is L1 d cache loads.This can be done by two ways :-

**SIMD Instruction :** We have used SIMD instruction from *AVX-2* code included in intrinsic.h header for c++.In the deepest loop for kernel_j we are doing 8 multiplications simultaneously instead of one multiplication and this is the power of SIMD instructions. It will not only help in reducing time taken by multiplication operation but it will also help in reducing branch instruction because each time we will be incrementing our deepest loop by 8 instead of 1.

**Loop unrolling :** We have loop unrolling for *outer_j* loop.In this approach we will DC for 8 contiguous cells of output matrix.Instead of running whole kernel matrix loop for only one cell in one go what we will be doing is we will perform all calculation inside kernel matrix loop for all contiguous 8 elements of output matrix together. It will suddenly help in reduction of L1-d cache miss because we are now utilizing a kernel cell more efficiently before it evacuates from cache due to demand for other cells.
Best thing is these optimization can be done above the optimization done earlier in part A. Now we will look in to performance gain by optimized code 2 implemented above part A optimizations, over reference code.

TABLE III
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND OPTIMIZED CODE 2

| Input | Kernel | Reference (ms) | Optimized(ms) | Reduction(%) |
|---|---|---|---|---|
| 4096 | 9 X 9 | 18999 | 7365 | 61.2 |
| 4096 | 11 X 11 | 28043 | 10877 | 61.2 |
| 4096 | 13 X 13 | 40410 | 15675 | 61 |
| 8192 | 9 X 9 | 75423 | 28888 | 61.6 |
| 8192 | 11 X 11 | 111083 | 42408 | 61.8 |
| 8192 | 13 X 13 | 155543 | 58441 | 62.4 |
| 16384 | 9 X 9 | 311895 | 127707 | 59 |
| 16384 | 11 X 11 | 467432 | 189739 | 60 |
| 16384 | 13 X 13 | 643903 | 248032 | 61.4 |

We can say by applying both of optimization to our single threaded implementation we have achieved an overall $60-65\%$ reduction in time.Apart from reduction in execution time we can show that we have successfully tackled third bottleneck also by reducing number of load misses in L1 data cache.Again for purpose of calculations we will unit "Lakh as unit" for Load misses.

TABLE IV
L1-D CACHE LOAD MISSES COMPARISON BETWEEN REFERENCE PROGRAM AND OPTIMIZED CODE 2

| Input | Kernel | Reference (lac) | Optimized(lac) |
|---|---|---|---|
| 4096 | 9 X 9 | 3872.34 | 513.33 |
| 4096 | 11 X 11 | 6829.58 | 663.83 |
| 4096 | 13 X 13 | 9288.42 | 904.58 |
| 8192 | 9 X 9 | 13252.57 | 2007.10 |
| 8192 | 11 X 11 | 23862.71 | 2506.42 |
| 8192 | 13 X 13 | 34615.29 | 3537.98 |
| 16384 | 9 X 9 | 49913.79 | 8065.96 |
| 16384 | 11 X 11 | 75430.79 | 10245.98 |
| 16384 | 13 X 13 | 109548.86 | 14190.99 |

From above table we have seen how Using Loop unrolling and SIMD instruction can help in reducing L1-d cache misses will which will eventually result in better execution time.

## III. MULTITHREADED DIALATED CONVOLUTION IMPLEMENTATION

With bottleneck in clock frequency and pipeline level parallelism, performance of single threaded algorithms gets effected. To overcome these limitations we can use multi-threading.Multi Threading requires no independence between multiple threaded if there will be parallelism then we need to exclusively implement synchronization between threads.
To implement Dilated convolution we are dividing no of output row between threads such that there will be no dependencies.Ideally if no .of threads are t then execution time should reduce by factor of $\frac{1}{t}$

## A. *Scalability of Threads*

We will take optimized single thread code as a base code for our multithread program.We will test our thread implementation for different number of threads.Below are the results for DC implementation for no of threads taken as 2,4,8,16.Our goal is to see the effect of scaling no of threads on execution time and finding limit till which it can provide scalability in speed up.

TABLE V
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND SINGLETHREAD CODE

| Input | Kernel | Reference (ms) | SingleThread(ms) | Speed Up%) |
|---|---|---|---|---|
| 4096 | 9 X 9 | 18999 | 7365 | 2.5 |
| 4096 | 11 X 11 | 28043 | 10877 | 2.5 |
| 4096 | 13 X 13 | 40410 | 15675 | 2.57 |
| 8192 | 9 X 9 | 75423 | 28888 | 2.6 |
| 8192 | 11 X 11 | 111083 | 42408 | 2.61 |
| 8192 | 13 X 13 | 155543 | 58441 | 2.66 |
| 16384 | 9 X 9 | 311895 | 127707 | 2.44 |
| 16384 | 11 X 11 | 467432 | 189739 | 2.46 |
| 16384 | 13 X 13 | 643903 | 248032 | 2.59 |

Above table shows that single threaded program after optimization can achieve speedup of approximately 2-2.5x.Here we are using single thread implementation as a base implementation for multi thread program.Now we will see impact of increasing number of threads.

TABLE VI
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND MULTITHREADED CODE USING 2 THREADS

| Input | Kernel | Reference (ms) | Multithreadms) | SpeedUp |
|---|---|---|---|---|
| 4096 | 9 X 9 | 16690 | 4029 | 4.14 |
| 4096 | 11 X 11 | 24068 | 5495 | 4.37 |
| 4096 | 13 X 13 | 32914 | 7441 | 4.42 |
| 8192 | 9 X 9 | 67420 | 15100 | 4.46 |
| 8192 | 11 X 11 | 98086 | 21669 | 4.52 |
| 8192 | 13 X 13 | 141486 | 30477 | 4.64 |
| 16384 | 9 X 9 | 268785 | 70873 | 3.79 |
| 16384 | 11 X 11 | 396257 | 92228 | 4.29 |
| 16384 | 13 X 13 | 538841 | 119964 | 4.49 |

Here from above table it is clear increasing number of thread to 2 threads speed up also increased to 4x.

TABLE VII
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND MULTITHREADED CODE USING 4 THREADS

| Input | Kernel | Reference (ms) | Multithreadms) | SpeedUp |
|---|---|---|---|---|
| 4096 | 9 X 9 | 15864 | 1899 | 8.35 |
| 4096 | 11 X 11 | 23881 | 2806 | 8.51 |
| 4096 | 13 X 13 | 32887 | 3755 | 8.75 |
| 8192 | 9 X 9 | 64599 | 7630 | 8.46 |
| 8192 | 11 X 11 | 97454 | 11074 | 8.8 |
| 8192 | 13 X 13 | 133274 | 15153 | 8.79 |
| 16384 | 9 X 9 | 285092 | 35793 | 7.96 |
| 16384 | 11 X 11 | 395989 | 45704 | 8.66 |
| 16384 | 13 X 13 | 544778 | 62136 | 8.76 |

For 4 threads speed up goes to 8x approximately whih shows speed up is propotional to number of threads.

TABLE VIII
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND MULTITHREADED CODE USING 8 THREADS

| Input | Kernel | Reference (ms) | Multithreadms) | SpeedUp |
|---|---|---|---|---|
| 4096 | 9 X 9 | 15864 | 1756 | 9.03 |
| 4096 | 11 X 11 | 23881 | 2568 | 9.29 |
| 4096 | 13 X 13 | 32887 | 3295 | 9.98 |
| 8192 | 9 X 9 | 64599 | 7718 | 8.46 |
| 8192 | 11 X 11 | 97454 | 10386 | 9.38 |
| 8192 | 13 X 13 | 133274 | 14659 | 9.09 |
| 16384 | 9 X 9 | 285092 | 32315 | 8.8 |
| 16384 | 11 X 11 | 395989 | 41914 | 9.44 |
| 16384 | 13 X 13 | 544778 | 59570 | 9.14 |

TABLE IX
EXECUTION TIME COMPARISON BETWEEN REFERENCE PROGRAM AND MULTITHREADED CODE USING 16 THREADS

| Input | Kernel | Reference (ms) | Multithreadms) | SpeedUp |
|---|---|---|---|---|
| 4096 | 9 X 9 | 15864 | 1858 | 8.53 |
| 4096 | 11 X 11 | 23881 | 3180 | 7.5 |
| 4096 | 13 X 13 | 32887 | 3497 | 9.4 |
| 8192 | 9 X 9 | 64599 | 7230 | 8.93 |
| 8192 | 11 X 11 | 97454 | 11184 | 8.7 |
| 8192 | 13 X 13 | 133274 | 13808 | 9.6 |
| 16384 | 9 X 9 | 285092 | 31913 | 8.9 |
| 16384 | 11 X 11 | 395989 | 40551 | 9.7 |
| 16384 | 13 X 13 | 544778 | 56801 | 9.5 |

By above tables noting speed up for different number of threads we have observed that the speed up was scaling proportionally to no of threads till no of threads reaches no of cores in the system (remember we have mentioned in hardware budget that we have 4 cores).

As the no of threads surpasses no of cores in the system it doesn't speed up with doubling rate. Although it has not increased with doubling rate but have provided a slightly better speed (from 8x to 9x). It is because there can be two threads per core hence if one thread stalls cpu can schedule other thread to fill up the stalls hence give better execution time.

As the no of threads crosses (no.of core * no.of thread per core) i.e 8, we have seen a fluctuation in execution time.For some iteration it is give slight improvement, for some it is giving slight degradation in performance with respect to 8 threads.
By this experiment we have concluded no f threads must be equal to the no of core X no of threads per core, this way of initializing no of threads in multi threaded program will give us best result.

## IV. Conclusion

In this report we have observed how a poorly returned code can be optimized by effectively using hardware counters to find bottleneck in the program.We have also understood how general principal like Code motions,loop unrolling and SIMD Instruction can be helpful in reducing execution time by 60 % in case of single threaded program.

Then we understood some limitations of single thread program hence we moved to multi thread program ,where we checked scalability of threads for a particular hardware configuration.
We have seen for a particular hardware budget the speed up will be directly proportional to the number of thread till threads are lesser or equal to no of CPU cores in that machine.Once no of threads will exceed no of cpu the there will not be very much difference in execution time.At end of this experiment we got reduction of approximately 90% for our Multi thread program with 8 threads in Hardware platform having quad core CPU.

## References

[1] https://en.wikipedia.org/wiki/Intrinsic_function
[2] https://man7.org/linux/man-pages/man7/pthreads.7.html
[3] https://en.wikipedia.org/wiki/Pthreads
[4] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
[5] https://www.brendangregg.com/perf.html