Optimization of 2D Convolutions

Aditi Tripathi, Jenya Singh, Sreenidhi Saranayan

I. Introduction

Convolution algorithms present a key component and a significant step in many modern real-world applications like signal processing, image processing, computer vision, and deep learning. Despite their high arithmetic complexity, these algorithms are widely used because of their great importance for extracting signal properties and features.

Convolution algorithms are compute-hungry, and thus become the bottleneck in achieving high-performance applications. Since they form the crux of most applications, a slight speed-up in these algorithms can result in a significant boost in performance of the entire application that they are part of. In order to achieve peak performance of these modern applications, we propose a vectorization-based implementation of 2D Convolution that exploits *data level parallelism (DLP)* while efficiently using the available cache. For this, we try to make the best use of the features of the architecture and the accompanying instruction set.

In this project, we focus on 2D Convolution in the domain of Image Processing wherein convolution is performed on the image with a small matrix (typically of size 3 x 3, 4 x 4, or 5 x 5) of weights, called the *filter* for the purpose of altering the properties of the image - for example, blurring, sharpening, softening, and embossing. We review two implementations for convolving a 2D image with a filter -
(1) *Vectorization using SIMD*, and (2) *Optimized SIMD-kernel implementation using independent operations*. To further improve performance of the implementation, we use compiler optimization techniques on loops, particularly loop unrolling. We then measure the OpenMP-based parallel performance of the base code and compare this against the kernel implementation performance.
To measure the boost in the convolution algorithm, we compare our incrementally optimized implementations against a naive implementation of 2D convolution that uses scalar operations within a 4-level deep nested loop.

II. Algorithm and Kernels

Convolution is a transformation technique of combining two signals mathematically to form a third signal. *2D Convolution* refers to the combination of two signals in both the x and y dimensions resulting in a 2D output signal. In image processing, the two input signals are the image itself and the filter of weighted elements, where different weights correspond to alteration of different image properties. The convolution algorithm to calculate one element (pixel) of the output image involves three steps - (1) Sliding the filter over the image at a certain position, (2) Performing element-wise multiplication between the overlapped image and filter elements, and (3) Summing-up the calculated products. These three steps are performed iteratively on different filter-image overlapped positions until the filter passes over all the image pixels. Thus, dot-products form the basis of the algorithm.

The kernel identified as part of the algorithm is the 2D convolution operation itself, where, if A is an image of size x x y and K is the chosen filter of size i x j, then the convolved image, B is calculated as:

$$B_{y,x} = \Sigma_i \Sigma_j A_{y+i,x+j} K_{i,j} \quad (1)$$

The independent operations that make up the kernel are the convolutions performed between the image and the kernel at different positions. The dependent instructions that make up these independent operations are the add instructions that sum-up the products of the overlapped filter-image elements. In our implementation, we use specialized *Fused Multiply-Add (FMA)* units that perform both addition and multiplication simultaneously. These FMA instructions are compute-heavy and cause a bottleneck in the performance of the algorithm. The limited number of available registers (for load-store operations) also cause a bottleneck in achieving peak performance.

III. Kernel Design

For the purpose of this study, we use matrices of varied sizes - from 8 x 8 to 1024 x 1024 consisting of double-precision floating point elements, where each element represents the pixels of images of corresponding sizes, and a separable 4 x 4 filter. We run our architecture-specific implementation on an *Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz* machine.

Basic Kernel Implementation

This implementation solely depends on the use of SIMD instructions to improve performance of the convolution algorithm.

The first step in designing the kernel involves the survey of the instruction set and the instruction set extension of the *Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz* machine. After careful analysis, we chose the *AVX-instruction-set-based* FMADD SIMD intrinsic for the core dot-product computation owing to the following reasons - (1) it is faster as it performs both addition and multiplication simultaneously using a single unit, and (2) it provides extra accuracy as it rounds-off intermediate results using extra bits. We also use the *AVX-instruction-set-based* ADD, HADD and PERMUTE SIMD instructions to ensure the output matrix (i.e., the altered image) elements are stored in the right order.

In this implementation, we use serialized structure of loading, performing convolution, and storing the convolved output pixels, while using SIMD instructions to exploit vectorization benefits.

We choose the size of the kernel based on the number of registers available in the *Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz* machine - 16. This is because the limited number of registers in any machine is a major bottleneck to achieving theoretical peak performance. For each iteration, we divide the available 16 registers in a way that incorporates all three matrices involved in convolution - 8 registers for loading 8 rows of input image vectors in a row-major fashion, 4 registers for loading the filter vectors, and 4 registers for storing the convoluted output vectors. Thus, the **size of the kernel** is 8 x 4, using which we calculate 4 output (altered image) pixel values in a row-major order.

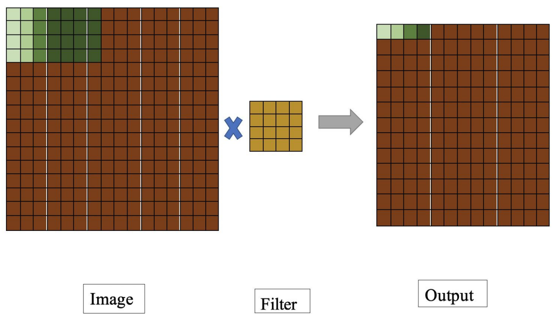


Fig. 3.1. Pictorial Representation of the elements loaded from the input image and the kernel, and the corresponding convolved output image values computed in the Basic Kernel Implementation

The theoretical peak is computed based on the architecture. In order to compute the theoretical peak of our implementation on the *Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz* machine that uses *Haswell Architecture*, we analyze the following given specifications for the *Haswell Architecture AVX-SIMD intrinsics* used in the implementation -

SIMD Instruction	Latency	Throughput
_mm256_fmadd_pd	5	2
_mm256_hadd_pd	6	0.5
_mm256_add_pd	3	1

Table. 3.1. Latency and Throughput (in Haswell Architecture) of the Different SIMD Instructions used in the Kernel Implementations

Based on these values, we get - 5 x 2 = 10 independent FMA instructions, 3 x 1 = 3 independent ADD instructions, and 6 x 0.5 = 3 independent HADD instructions per cycle. Since we use SIMD-vectorization, we get - 5 x 2 x 4 = 40 independent FMA instructions, 3 x 1 x 4 = 12 independent ADD instructions, and 6 x 0.5 x 4 = 12 independent HADD computations per cycle.

Thus, the **theoretical peak performance** of the implementation is 40+12+12 = 64 FLOPS/cycle.

Optimized Kernel Implementation

This implementation aims to further optimize the basic kernel implementation by exploiting the presence of independent operations in the convolution algorithm. The implementation uses the same *SIMD-AVX* instructions as the basic kernel implementation - FMADD, HADD, ADD and PERMUTE. However, instead of loading 8 input image row vectors in a row-major fashion in each iteration, we load 8 input image row vectors in a column-major fashion, and we perform dot-product (using FMADD) of these image row vectors with different row vectors of the filter. Thus, we are able to compute 7 different output (altered image) pixels in each iteration. Exploiting *data level parallelism (DLP)* gives us a further boost in performance.

We choose the size of this kernel by taking into account both the number of independent computations performed in an iteration, and the limitation on the number of registers in the *Haswell Architecture*. For each iteration, we divide the available 16 registers in a way that incorporates all three matrices involved in convolution - 4 registers for loading 8 rows of input image vectors in a column-major fashion (the registers are reused), 4 registers for loading the filter vectors, and 8 registers for storing the convoluted output vectors. Thus, the **size of the kernel** is 8 x 4, using which we calculate 8 output (4 complete, and 4 temporary) (altered image) pixel values in neither a row-major nor a column-major fashion (to efficiently use the available cache).

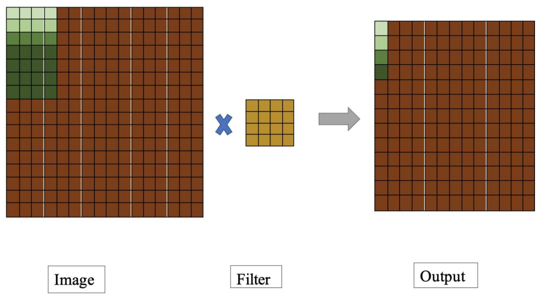


Fig. 3.2. Pictorial Representation of the elements loaded from the input image and the kernel, and the corresponding convolved output image values computed in the Optimized Kernel Implementation

Since we use the same SIMD instructions as the basic kernel implementation, the **theoretical peak performance** of this implementation = 64 FLOPS/cycle.

IV. Memory Optimization

The two kernel implementations are oblivious to the amount of cache memory available. However, the **basic kernel implementation**, in each iteration, aims to take advantage of the data already sitting in cache. This reduces the number of cache misses at each level of cache, and thus, also reduces the number of cache evictions, thus giving a slight increase in performance as compared to the base code. However, across iterations, since it does not take into account the amount of cache available, cache is not used to its full potential. Whereas, the **optimized kernel implementation** favors reduction of data movement due to excessive loads over efficient utilization of cache in each iteration. Due to this, more number of convolved output elements are computed with fewer loads of input. However, since we traverse in both dimensions - in the column-dimension within a single iteration, and in the row-dimension across iterations, thus exploiting spatial and temporal locality of the elements, we significantly improve the efficiency of cache utilization as compared to the base code.

V. Parallel Implementation

The *basic kernel implementation* and the *optimized kernel implementation* are vectorized using SIMD whilst using independent operations in each iteration. Since these SIMD implementations are optimized to a significant extent by taking advantage of *data level parallelism (DLP),* parallelizing them using OpenMP degrades the performance, instead of the expected improvement in performance.

However, in order to analyze the impact of using *thread-level parallelism (TLP)* in the convolution algorithm, we use OpenMP to parallelise the naive base code implementation. After analyzing the performance of (1) parallel for, (2) parallel region, (3) task, (4) parallel for with static scheduling (using chunk-size as 1), and (5) parallel for with dynamic scheduling (using chunk-size as 1) on the base code, we see the following -

OpenMP Parallelism Scheme	**Performance Impact (as compared to the Base Code)**
Parallel For	Degradation
Parallel Region	Improvement
Tasks	Degradation
Parallel For with Static Scheduling (using Chunk-size as 1)	Degradation
Parallel For with Dynamic Scheduling (using Chunk-size as 1)	Degradation

Table. 5.1. Table of Performance Impact for Different OpenMP Parallelism Schemes

Thus, we choose the *OpenMP Parallel Region Scheme* for parallelizing the base code since only this scheme gives an improvement in performance.

As for the number of threads, since we use input images of various sizes, the number of threads that give an improvement in performance vary for different sizes of the input image. The choice of number of threads for the different image sizes based on heuristics are -

Input Image Size	Number of Threads Chosen (based on performance improvement over optimised SIMD)
8 x 8	1
16 x 16	1
32 x 32	4
64 x 64	8
128 x 128	14
256 x 256	28
512 x 512	28
1024 x 1024	35
2048 x 2048	35
4096 x 4096	35

Table. 5.2. Table of Performance Impact for Different OpenMP Parallelism Schemes

VI. Experimental Setup

Our codebase has the following files:

1. base.c (Base Code)
2. Main.c (Basic Kernel Implementation)
3. Main_optimized.c (Optimized Kernel Implementation)
4. Conv_mpi.c (OpenMP-based Parallelized Implementation)
5. Makefile (Compiles and Runs all experiments)
6. README.txt (Instructs how to run the codebase)

The codebase is submitted as a zip folder named as '*project_2DConv_aditit_jenyas_ssaranya.zip*'. Unzip and run the codebase using the instructions from *README.txt* in the *ECE005* machine that uses *Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz.*

VII. Results & Discussion

For measuring performance, we plot the latency (in cycles) of the 3 optimized implementations - *Basic Kernel*, *Optimized Kernel*, and *OpenMP-based Parallized Implementation*, and compare it with the latency of the *Base Code* -

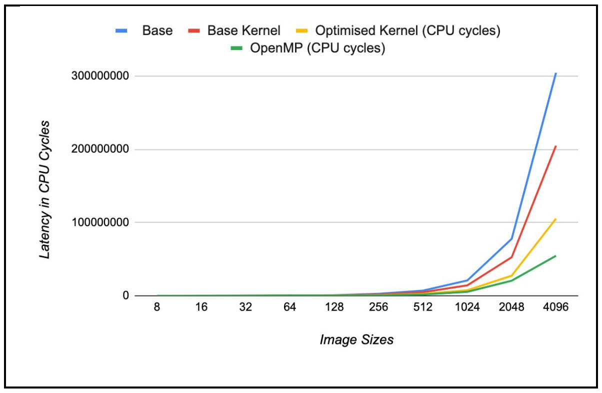


Fig. 7.1. Performance (Latency) Graph for Different Implementations (Base, Basic Kernel, Optimized Kernel, and OpenMP)

The lower the latency, the faster the implementation, and thus, better would be the corresponding implementation.

Performance of Basic Kernel Implementation

Analyzing the graph in Fig. 7.1., we see that the *Basic Kernel* slightly boosts the performance. The peak performance of the Basic Kernel is less than the ideal theoretical peak of 64 FLOPS/cycle. This can be attributed to two reasons - (1) we do not use independent operations, and (2) the bottleneck caused by the number of registers available. The **peak performance** of this implementation is thus - 1 independent FMA instruction, 3 independent HADD instructions, and 3 independent ADD instructions = 1*4 + 3*4 + 3*4 = 28 FLOPS/cycle.

Performance of Optimized Kernel Implementation

Analyzing the graph in Fig 7.1., we see that the *Optimized Kernel* boosts the performance further (as compared to the *Basic Kernel Implementation*). The peak performance of the Optimized Kernel is less than the ideal theoretical peak of 64 FLOPS/cycle. This can be attributed to two reasons - (1) less number of independent operations than ideal, and (2) the bottleneck caused by the number of registers available. The **peak performance** of this implementation is thus - 7 independent FMA instructions, 3 independent HADD instructions, and 3 independent ADD instructions = 8*4 + 3*4 + 3*4 = 56 FLOPS/cycle. Thus, we see that the peak performance of this implementation approaches the ideal theoretical peak, and offers a significant boost in performance as compared to the *Basic Kernel Implementation*.

Performance of OpenMP-based Parallized Implementation

Analyzing the graph in Fig 7.1., we conclude that SIMD optimization works well for both small and large size images. *OpenMP* performs better than *SIMD* by 30% - 50% for large size images but worse for image sizes less than 128 x 128 , which is expected as the thread scheduling overhead becomes a bottleneck with small data move
ment.

Conclusion about Overall Performance

In terms of performance, *OpenMP* is the winning model, except for very small images where thread scheduling overhead becomes excessive. The only drawback is the choice of the number of threads based on heuristics, extensive experiments and perhaps types of image compressions.

Image sizes	**Optimised SIMD Kernel (CPU cycles)**	**OpenMP (CPU cycles)**	**% Change in performance**
8 x 8	351.8	5524.08	-1470%
16 x 16	1964.1	2.05e4	-942%
32 x 32	10217.8	3.7e4	-268%
64 x 64	49475.2	5.6e5	-1026%
128 x 128	207597.8	2.07e5	0.02%
256 x 256	8.7e5	5.4e5	38%
512 x 512	2.6e6	1.7e6	36%
1024 x 1024	7.4e6	5.4e6	27%
2048 x 2048	2.7e7	2.05e7	25%
4096 x 4096	1.05e8	5.45e7	48%

Table. 7.1. Table comparing Performance of Optimized SIMD Kernel vs OpenMP-based

Parallel Performance with Different Number of Threads

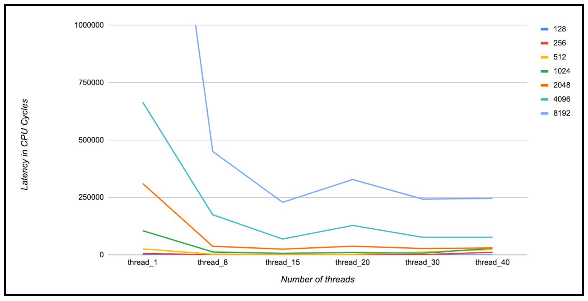


Fig. 7.2. Performance (latency) for different image sizes (128 x 128 to 8192 x 8192) and for different number of threads

VIII. Future Work

(1) Using a different combination of SIMD instructions (for example, FMADD and BROADCAST) may give better performance. Thus, by experimenting with different combinations of SIMD instructions and analyzing the latency and throughput of each, we should carefully select the SIMD-instructions combination that would have the highest theoretical peak.
(2) Make the implementations cache-aware to use the available cache to its maximum potential for the given algorithm. This would lead to a significant decrease in the number of cache misses and cache evictions, and thus, would give a significant improvement in performance.
(3) The current implementation of OpenMP requires more experimentation in terms of considering even larger image sizes and different combinations of scheduling schemes - *dynamic, guided* and *static* using different chunk-sizes along with other parallel primitives.

References

[1] S. Smith, Digital Signal Processing: A Practical Guide for Engineers and Scientists: A Practical Guide for Engineers and Scientists. Newnes, 2013.
[2] Intel Intrinsics Guide - https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX&expand=444,1789
[3] 2D Convolution in Image Processing https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/
[4] 2D Image Convolution using Three Parallel Programming Models on the Xeon Phi - https://arxiv.org/pdf/1711.09791.pdf

Appendix

I. Theoretical Peak Computation: Pictorial Representation

The theoretical peak computation of an implementation involves determination of the number of independent computations performed by the corresponding functional units in ‘latency’ cycles.
In both the kernel implementations, we use three main *AVX-SIMD* instructions - FMADD, HADD, and ADD. Determination of the number of independent computations for each SIMD unit can be understood pictorially with the help of the following pipeline diagrams -

(1) **_mm256_fmadd_pd**

In Haswell Architecture -
Latency = 5
Throughput = 2

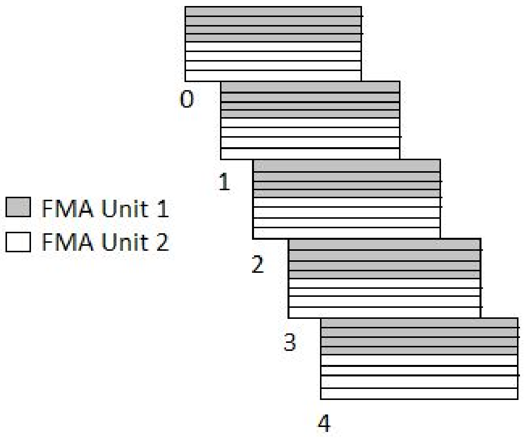


Fig. A.1.1. Pictorial Representation of number of independent computations for FMADD

In 5 cycles, the FMA units in the machine can perform 5*2*4 = 40 independent FMA operations.

(2) **_mm256_hadd_pd**

In Haswell Architecture -
Latency = 6
Throughput = 0.5

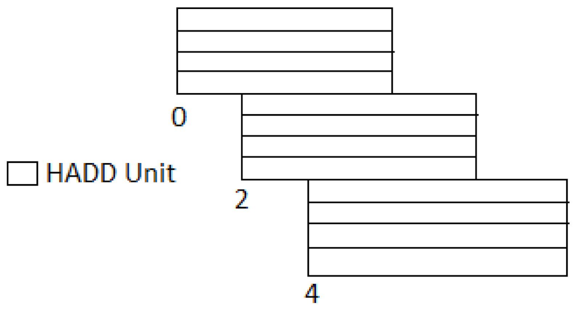


Fig. A.1.2. Pictorial Representation of number of independent computations for HADD

In 6 cycles, the HADD unit in the machine can perform 6*0.5*4 = 12 independent HADD operations.

(3) **_mm256_add_pd**

In Haswell Architecture -
Latency = 3
Throughput = 1

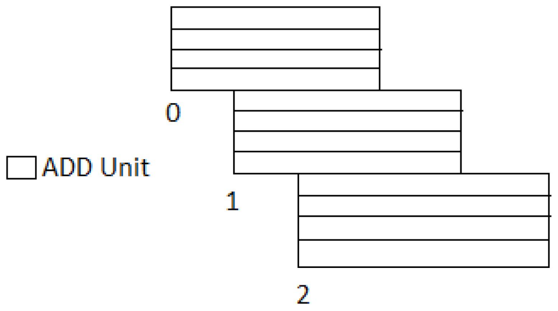


Fig. A.1.3. Pictorial Representation of number of independent computations for ADD

In 3 cycles, the HADD unit in the machine can perform 3*1*4 = 12 independent HADD operations.

II. Latency of all Implementations and the Percentage Speed-up Compared to the Base Code

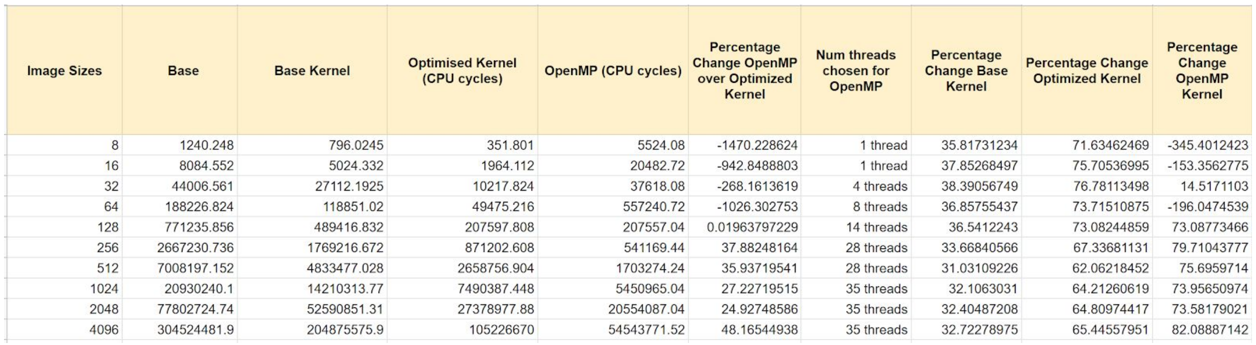
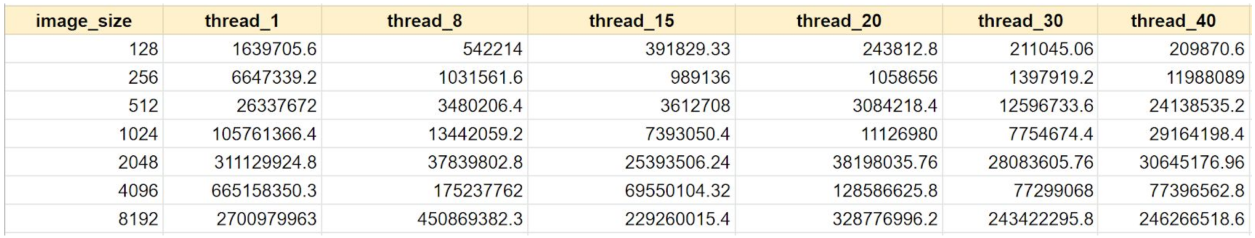


Image Sizes	Base	Base Kernel	Optimised Kernel (CPU cycles)	OpenMP (CPU cycles)	Percentage Change OpenMP over Optimized Kernel	Num threads chosen for OpenMP	Percentage Change Base Kernel	Percentage Change Optimized Kernel	Percentage Change OpenMP Kernel
8	1240.248	796.0245	351.801	5524.08	-1470.228624	1 thread	35.81731234	71.63462469	-345.4012423
16	8084.552	5024.332	1964.112	20482.72	-942.8488803	1 thread	37.85268497	75.70536995	-153.3562775
32	44006.561	27112.1925	10217.824	37618.08	-268.1613619	4 threads	38.39056749	76.78113498	14.5171103
64	188226.824	118851.02	49475.216	557240.72	-1026.302753	8 threads	36.85755437	73.71510875	-196.0474539
128	771235.856	489416.832	207597.808	207557.04	0.01963797229	14 threads	36.5412243	73.08244859	73.08773466
256	2667230.736	1769216.672	871202.608	541169.44	37.88248164	28 threads	33.66840566	67.33681131	79.71043777
512	7008197.152	4833477.028	2658756.904	1703274.24	35.93719541	28 threads	31.03109226	62.06218452	75.6959714
1024	20930240.1	14210313.77	7490387.448	5450965.04	27.22719515	35 threads	32.1063031	64.21260619	73.95650974
2048	77802724.74	52590851.31	27378977.88	20554087.04	24.92748586	35 threads	32.40487208	64.80974417	73.58179021
4096	304524481.9	204875575.9	105226670	54543771.52	48.16544938	35 threads	32.72278975	65.44557951	82.08887142

Fig. A.2.1. Table of Latencies for all Implementations, and the corresponding Percentage Speed-up of these Implementations as compared to the Base Code

III. Latency of all OpenMP-based Parallized Implementation using Parallel Regions with Different Number of Threads



image_size	thread_1	thread_8	thread_15	thread_20	thread_30	thread_40
128	1639705.6	542214	391829.33	243812.8	211045.06	209870.6
256	6647339.2	1031561.6	989136	1058656	1397919.2	11988089
512	26337672	3480206.4	3612708	3084218.4	12596733.6	24138535.2
1024	105761366.4	13442059.2	7393050.4	11126980	7754674.4	29164198.4
2048	311129924.8	37839802.8	25393506.24	38198035.76	28083605.76	30645176.96
4096	665158350.3	175237762	69550104.32	128586625.8	77299068	77396562.8
8192	2700979963	450869382.3	229260015.4	328776996.2	243422295.8	246266518.6

Fig. A.3.1. Table of Latencies for the OpenMP-based Parallelized Implementation with different number of threads