HPCA Programming Assignment 2023-2024

Optimizing Performance of Dilated Convolution(PartA)

Machine Specification

For the single threaded and the multi-threaded executions, we have used a system with Intel x86 64-bit ISA with 4 cores and 2 threads per core.L1d cache size : 128 KB (4 instances),L1i cache size : 128 KB (4 instances),L2 cache size : 1 MB (4 instances),LL Cache size : 6 MB (1 instance).

For the running of large matrix 16834 * 16834(for Single thread,Multithread) We used a system with specifications Intel x86 64-bit ISA with 4 cores and 2 threads per core.L1d cache size : 192 KB (4 instances),L1i cache size : 128 KB (4 instances),L2 cache size : 2 MB (4 instances),LL Cache size : 6 MB (1 instance).

Input: Input Matrix of dimensions: Input_Row x Input_Column.

Kernel Matrix of dimensions: Kernel_Row x Kernel_Column.

Output: An Output Matrix of dimensions: (Input_Row-Kernel_Row+1) x (Input_Column – Kernel_Column +1)

Speed up = Runtime of unoptimized program/Runtime of optimized program

Our solution enhances the efficiency of a single-threaded implementation of Dilated Convolution by introducing a series of optimizations. Subsequently, we extend these optimizations to leverage multithreading capabilities. In Dilated Convolution, the arithmetic operations are inherently independent, with similar operations executed on each index value of the output. This intrinsic independence presents a vast potential for exploring parallelism and exploiting data parallelism across the computation, leading to significant improvements in performance

1. Single Thread Implementation:

1.1.Optimizing Dilated Convolution Code Motion, and Loop Unrolling :

- We also have minimized the number of unnecessary arithmetic computations to save CPU time.
- **Code Motion:**
  Code motion, also known as loop invariant code motion, involves moving code outside a loop if it remains constant across loop iterations. In the provided code, the expression int a = output_i * output_col; is moved outside the innermost loop. This calculation does not depend on the inner loop variables and can be computed once for each iteration of the outer loop. Moving this calculation outside the inner loop reduces redundant computations and can lead to performance improvements.
- **Loop Unrolling:**
  Loop unrolling is a technique that involves replicating loop bodies to reduce loop overhead and potentially improve instruction-level parallelism. The innermost loop, responsible for convolving elements with the kernel, is unrolled with a factor of 4. Instead of processing one element at a time, the loop processes four elements

simultaneously. This unrolling reduces loop control overhead and enhances the potential for instruction pipelining and parallel execution, leading to improved computational efficiency.

Speedup achieved on different matrix sizes:

Matrix size	Kernel size	Reference Time(ms)	Code motion + loop unrolling execution time(ms)	Speed Up
4096*4096	5*5	4087.95	1712.71	2.39
4096*4096	7*7	8245.46	3002.48	2.74
4096*4096	9*9	13281	5642.82	2.35
4096*4096	11*11	19662.3	8664.24	2.27
4096*4096	13*13	27634.5	11926.4	2.31
8192*8192	5*5	17757.7	7570.12	2.34
8192*8192	7*7	30862.9	12603.4	2.45
8192*8192	9*9	54491.7	23197.4	2.34
8192*8192	11*11	81232.4	34281.3	2.36
8192*8192	13*13	113442	44435.1	2.55
16384*16384	5*5	36828.1	21712.9	1.69
16384*16384	7*7	71833	37288.8	1.93
16384*16384	9*9	112958	61223.2	1.84
16384*16384	11*11	163031	86324	1.89
16384*16384	13*13	42338.4	127038	1.82

- There is a significant speedup achieved by this specific optimization.

1.2. SIMD(Singular Intruction multiple data) instructions :

- We can use SIMD instructions code to exploit more data parallelism.
- An AVX 256 bit vector is used in our code which performs 8 16-bit integer multiplications and integer additions in parallel.

- **Memory Access and Accumulation with SIMD**:Memory access and accumulation are optimized using SIMD instructions. The input values are loaded using _mm256_loadu_si256, and the convolution operation (_mm256_mullo_epi32) is performed in a vectorized manner.
- **Contiguous Processing and Alignment:**The results are then accumulated using _mm256_add_epi32, and the updated values are stored back in memory with _mm256_storeu_si256.
  The loop processing is aligned to handle contiguous blocks of data, as evident from the alignment of the __m256i vectors and the adjustment of the loop indices (output_j and output_i).

The optimization using SIMD instructions accelerates the convolution operation by leveraging the parallel processing capabilities of AVX2. SIMD allows multiple arithmetic operations to be performed simultaneously on data elements, enhancing the efficiency of the dilated convolution and potentially providing significant performance gains, especially when dealing with large datasets.

Speedup achieved on different matrix sizes:

Matrix size	**Kernel size**	**Reference Time(ms)**	**SIMD Execution**	**Speed Up**
4096*4096	5*5	4087.95	735.24	5.89
4096*4096	7*7	8245.46	1592.75	4.98
4096*4096	9*9	13281	2649.07	5.01
4096*4096	11*11	19662.3	3691.25	5.38
4096*4096	13*13	27634.5	4136.86	6.68
8192*8192	5*5	17757.7	4235.22	4.19
8192*8192	7*7	30862.9	5488.74	5.62
8192*8192	9*9	54491.7	9468.26	5.76
8192*8192	11*11	81232.4	14807.5	5.49
8192*8192	13*13	113442	16048.8	7.07
16384*16384	5*5	36828.1	7599.83	4.84
16384*16384	7*7	71833	13563	5.29
16384*16384	9*9	112958	24751	4.5
16384*16384	11*11	163031	30772.1	5.29

16384*16384	13*13	231187	42338.4	5.46

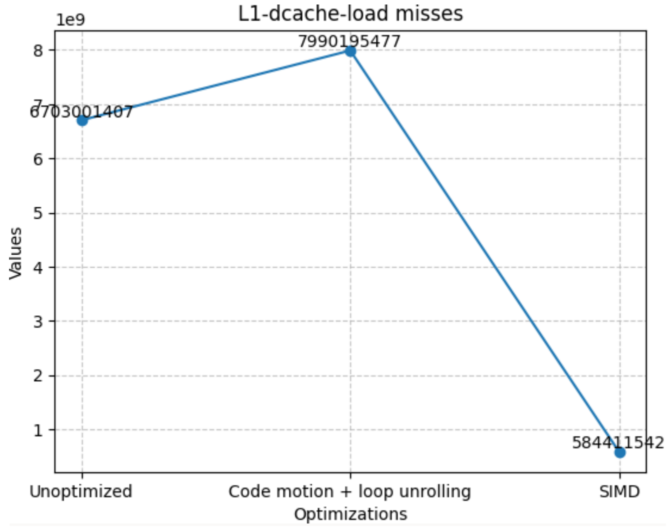
- We were able to achieve a greater speedup compared to the previous optimization.

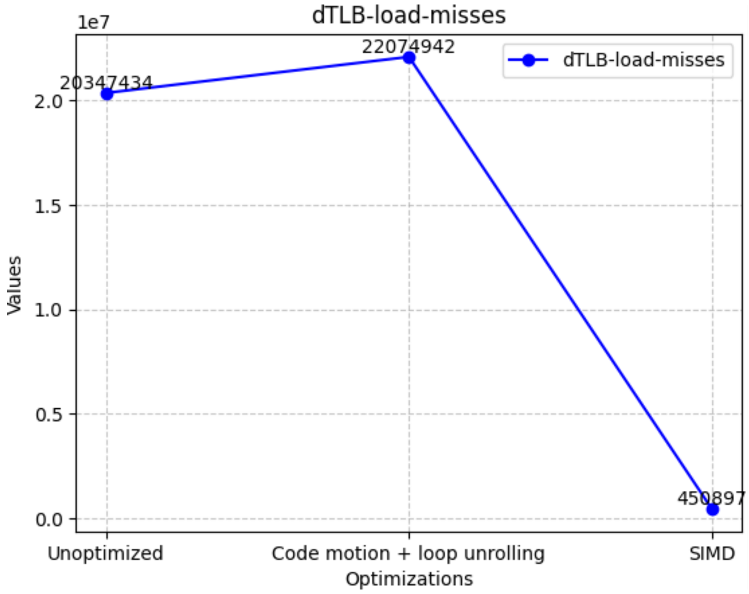
Observations using perf tool and hardware performance Counters:

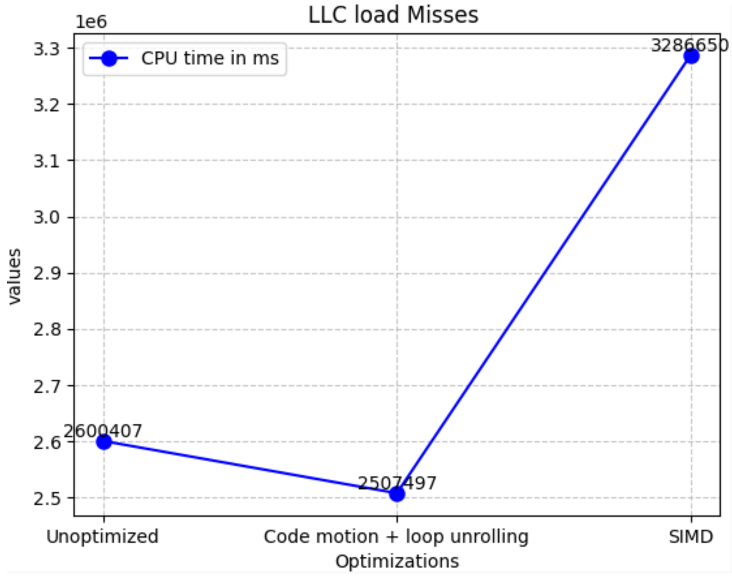
Here we Considered input size of 8192*8192 and kernel size of 13*13

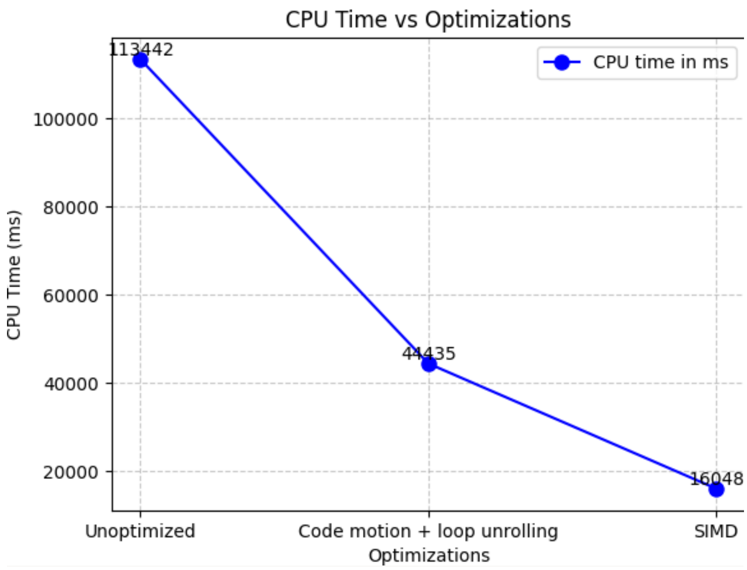
The results from the experiment for after applying each optimization have been tabulated below followed by the set of observations for each experiment.

Performance Events	Unoptimized	Code motion + loop unrolling	SIMD instructions
L1-dcache-load-misses	6703001407	7990195477	5884411542
L1-icache-load-misses	20347434	22074942	18272799
dTLB-load-misses	136741	108594	450897
LLC -load-misses	2600407	2507497	3286650
dTLB-store-misses	2954438	2756544	3524384
iTLB-load-misses	96101	96839	107088
branch-misses	32377256	98411522	34492213









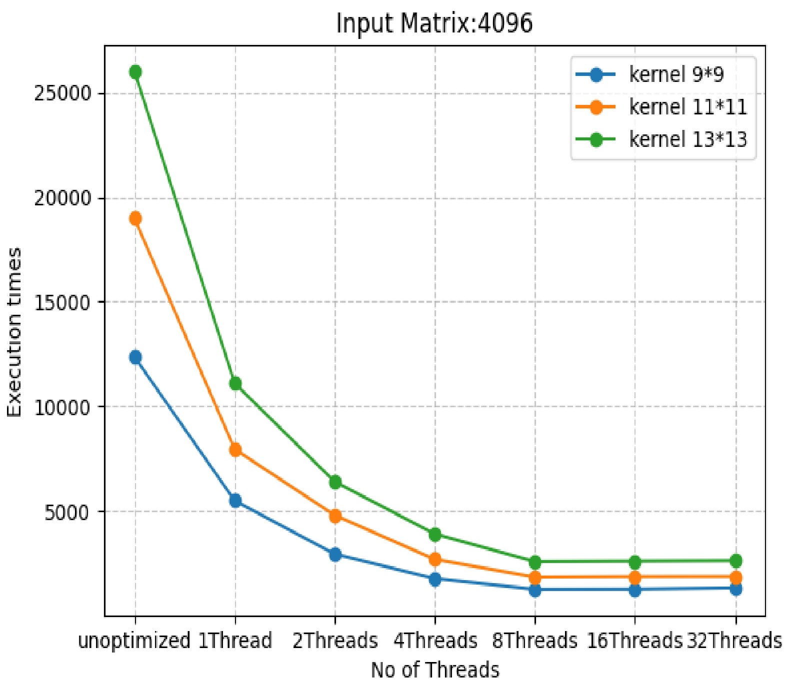
From the above results, we make the following observations

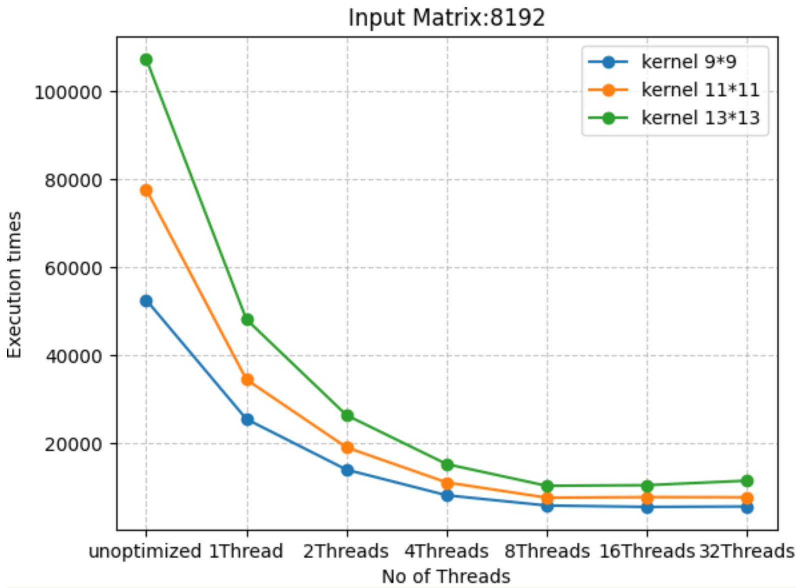
- For a vector operation of loading the data it costs only one load instruction. Load instruction will fetch the consecutive 8 elements in the memory starting from the given memory location. It can be inferred that huge decrease in the cache misses led to the decrease in execution time for the vector intrinsic code.
- L1-dcache misses -load misses as we can see that for loop unrolling the misses increase because that for vector intrinsic code it will load multiple rows at a time.Since matrix size is 8192 in above .that means row size of 32KB.it will take advantage of locality so the L1-dcache load misses are less for it
- For the unoptimized code due to sequential access the DTLB load misses are less than loop unrolling.DTLB misses increase and worsen when we do loop unrolling. Also once we unroll the loop, we further increase the load operations each time by 4 fold in our program and hence this leads to more DTLB misses .
- Considering the CPU times for each optimization we see that each subsequent optimization reduces the runtime for the code. As expected loop unrolling improves performance. The most significant improvements however are seen when we use processor (SIMD)intrinsic vectorization instruction which decrease the CPU runtime of program by more than 4 times.

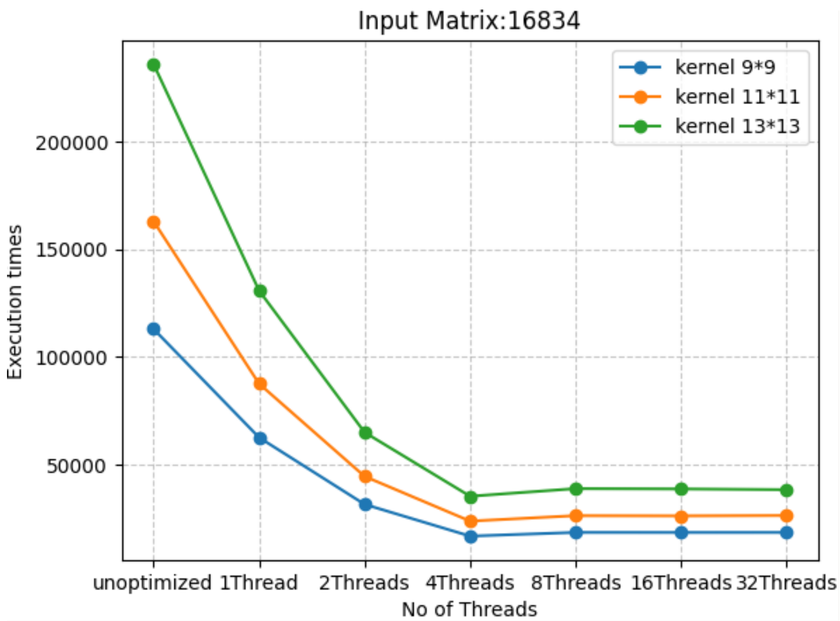
2. Multhreaded Implemtation:

- The provided code implements a multithreaded approach to parallelize the computation of a dilated convolution operation.
- The code determines the number of threads (Number_of_Threads) to be created.
- Same ideas as mentioned in the single threaded code are implemented in multithreaded code as individual versions.

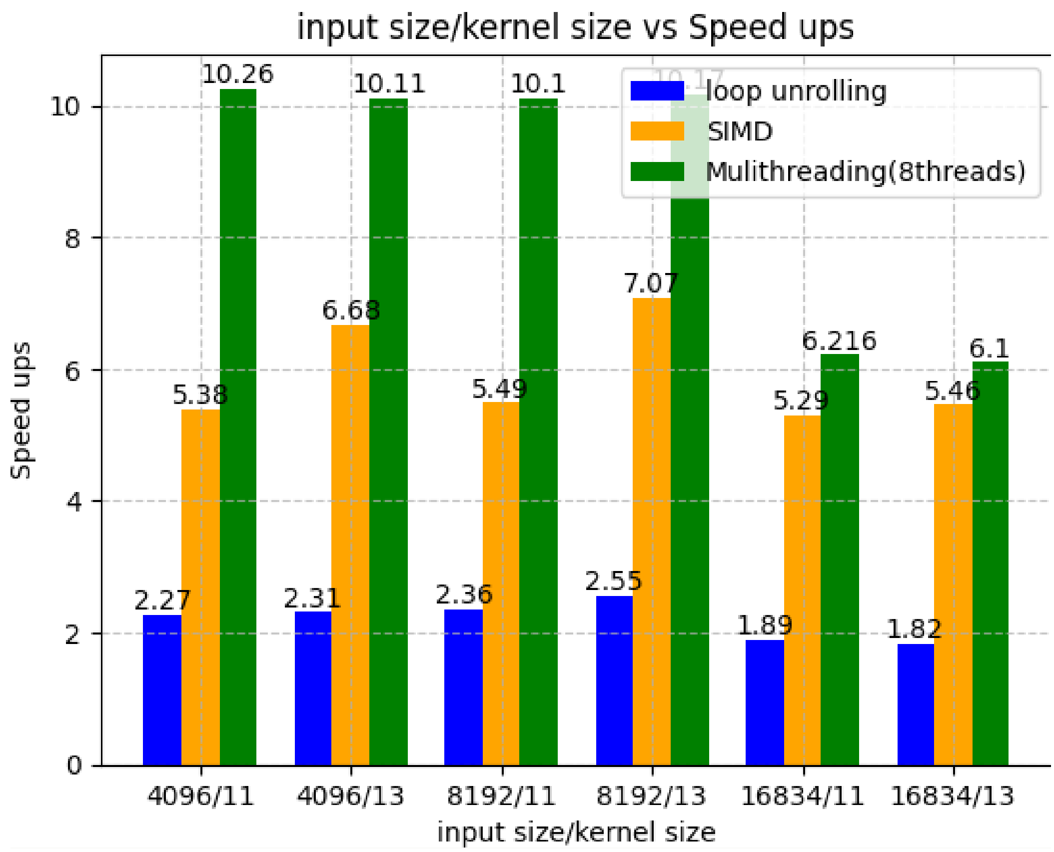
Input matrix size	Kernel Matrix size	Unoptimized code execution time	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
4096	9*9	12371.7	5507.14	2959.07	1787.04	1273.74	1280.92	1346.81
	11*11	18977.5	7968.09	4810.87	2705.97	1857.18	1879.57	1885.49
	13*13	26008.7	11117.7	6411.65	3918.27	2597.92	2618.95	2639.18
8192	9*9	52662.6	25574.5	14072.8	8254.42	5928.25	5616.84	5707.21
	11*11	77786.1	34580	19145.6	11183.8	7709.3	7832.03	7816.06
	13*13	107427	48248.5	26457.2	15340.2	10407.1	10554.6	11579.1
16834	9*9	112911	62520.9	31478.5	16714.9	18434.1	18421.2	18428.6
	11*11	162957	87510.9	44466.3	23679.7	26211.8	26121	26299.6
	13*13	235956	130737	64918.8	35233.3	38772.8	38637.8	38283.4







- As we can see that for 4K and 8K size matrix after 8 threads there is slight increasing of time because as mentioned in the specifications the machine of 4 cores of 2 threads each where it will best for 8 threads
- Similarly the same reason for slight improving after 4 threads in 16k matrix because of machine specifications.
- Increasing time because of more context switchings happen within the core.



- The reason of 16k slight decrease is it was run on different machine than 4k and 8k due that specifications it has changed but improvement is there for optimizations in each.
- Same as in the single thread implementation, it also out performs in the vector intrinsic implementation

Bottlenecks

We observe that there remain few bottlenecks in the program which limit the scalability of the code.

- There remains a limit to the speedup which AVX instructions can provide since these instructions can only do a certain number of parallel computations. There remains a large unexploited potential in terms of the thread-level performance of this program. An issue with the use of these vector intrinsic instructions can be their inflexibility since many instructions require the operands to be aligned in memory.
- While multithreading can speed up the code, it also can incur context switching and thread creation overheads which can limit its performance. Also it becomes very critical to divide the work among thread evenly since the run time of program depends on the slowest threads