DIFFUSION ON EMBEDDING SYSTEM*

REPORT ON CPU_BASED ACCELERATION

Li XiaoLong
Student ID: 2024-81484
Email: lixiaolong@snu.ac.kr

***Abstract*—This report investigates the acceleration of embedding systems using CPU-based diffusion methods. Through experiments and analysis, we demonstrate the trade-offs between speed and memory requirement and offer insights into optimizing the performance of such systems. This work provides both theoretical understanding and practical implementations aimed at improving the efficiency of diffusion model forward process.**

I. Introduction

Efficient convolution operations are critical for deep learning models, especially in tasks involving extensive computations such as image processing and segmentation. In this report, we focus on optimizing the `conv2d` function, a fundamental operation in convolutional neural networks (CNNs). Our approach leverages several advanced techniques to accelerate performance, including the transformation of input data using the `img2col` method and optimizing matrix multiplications via Single Instruction, Multiple Data (SIMD) and multi-threading.

We implemented three distinct `conv2d` versions to evaluate performance:

Original Convolution Operation: This first implementation follows the standard template code for convolution, where the operation is conducted in a straightforward manner without optimizations. This serves as our baseline for comparison.

Optimized Convolution Using img2col: The second implementation utilizes the `img2col` function to reshape the input data into a suitable format for matrix multiplication, combined with a matrix multiplication implementation that leverages multi-threading and SIMD for acceleration. This approach significantly enhances performance by reducing computational overhead and increasing data locality.

Advanced Convolution with Transposed Input: The third implementation builds on the second by first transposing the second input matrix (the convolution kernel) before performing the convolution operation. This version employs multi-threading, SIMD, and loop unrolling techniques to further boost performance. By optimizing the data access patterns and minimizing cache misses, this implementation aims to achieve the highest possible efficiency.

To ensure optimal performance, we tested these three `conv2d` implementations using structural parameters from the convolution layers of the AttentionUnet inference process. Based on these results, we designed an intelligent convolution function that dynamically selects the most efficient implementation based on the size of the input data, kernel dimensions, and other relevant factors. This adaptive approach ensures that the `conv2d` operation is always executed with maximum efficiency, tailored to the specific characteristics of the input data.

Here is the final convolution function I use:

Algorithm 1 Convolution 2D Function

Function conv2d(input, weight, bias, stride, padding, groups, dilation)
batch_size $\leftarrow$ input.shape[0]
in_channels $\leftarrow$ input.shape[1]
out_channels $\leftarrow$ weight.shape[0]
if groups > 1 **then**
  return conv2d_original(input, weight, bias, stride, padding, groups)
end if
if out_channels ≥ 64 **then**
  return conv2d_optimized_new_matmul(input, weight, bias, stride, padding, groups, dilation)
else
  return conv2d_optimized(input, weight, bias, stride, padding, groups, dilation)
end if
{For finer control, more parameters can be added}

II. Experiments

In this section, we evaluate the performance of three different convolutional implementations, using various convolution layers encountered during the inference process of the Attention U-Net model. The test cases involve a variety of input shapes and weight (filter) shapes, which represent typical scenarios in deep learning models. We designed a benchmark consisting of 30 test cases to compare the original convolution implementation, an optimized version, and a newly designed matrix multiplication-based (MatMul) convolution approach.

Table I(in Appendix) provides a detailed comparison of these implementations across different test cases. The key metric used to assess performance is the execution time (in milliseconds), and we observe significant differences between the three approaches. For example, the optimized implementation often reduces the execution time significantly compared to the original. However, the new MatMul implementation outperforms both in some cases, particularly when handling

larger kernels or more complex input/output shapes, such as in test case 1 where it reduced the time from 22 ms to 7 ms.

However, there are cases where the new implementation doesn't always yield the best performance, particularly for smaller input shapes, as seen in test case 2. This variability highlights the importance of dynamically selecting the most appropriate convolution method based on the input size, kernel size, and channel dimensions, leading to the design of a "smart" convolution function, capable of adapting its implementation based on these factors to achieve the best performance.

In summary, these experiments underscore the trade-offs between different convolution methods, with the potential for significant performance improvements when the optimal approach is selected.Based on the observation we adopt a new conv2d function to smartly select the best implement of conv2d according to the input for better performance.

III. Analysis and Discussion

The experiments conducted on the three convolution implementations revealed several key insights into their performance and efficiency. Overall, the results aligned with our expectations, demonstrating that optimized implementations can significantly reduce execution time compared to the original convolution operation.

For instance, in Test Case 1, the optimized convolution using the `img2col` method showed a reduction in execution time from 22 ms to 10 ms, while the new MatMul implementation further improved this to 7 ms. This result highlights the effectiveness of data reshaping and optimized matrix multiplications in enhancing performance.

However, there were unexpected results in Test Case 2, where the optimized implementation took 80 ms, compared to the original's 9 ms, and the new MatMul implementation took 117 ms. This discrepancy can be attributed to the overhead introduced by reshaping the input data and the complexities involved in optimizing matrix multiplications when dealing with smaller inputs. Such scenarios suggest that while optimizations generally yield performance gains, they may not be universally applicable and could introduce inefficiencies for certain configurations.

Additionally, it was observed that larger kernel sizes or more complex input shapes tend to favor the new MatMul implementation. In Test Case 3, for example, the optimized version achieved 22 ms compared to the new implementation's 19 ms, indicating that for specific scenarios, the choice of implementation should be context-dependent.

These findings emphasize the importance of adaptive techniques in convolution operations. By dynamically selecting the most efficient implementation based on input characteristics, we can significantly improve performance across a wide range of scenarios. The developed "smart" convolution function addresses this need, ensuring that the most suitable approach is chosen based on real-time analysis of input parameters.

Overall, the trade-offs between different convolution methods underscore the necessity for flexibility in implementation strategies, allowing systems to adapt to varying demands and maximize efficiency in embedding systems.

IV. Conclusion

To conclude, this report presents a comprehensive investigation into the optimization of the `conv2d` function in embedding systems through CPU-based diffusion methods. By exploring various implementations, we established a clear understanding of how to balance execution time and resource requirements effectively. The results demonstrate that leveraging advanced techniques such as `img2col`, multi-threading, and SIMD can lead to substantial performance enhancements.

Moving forward, the insights gained from this study can inform future research and practical applications in embedding systems, highlighting the importance of context-aware optimization strategies for convolution operations.

References

[1] A.M. Hemeida, S.A. Hassan, Salem Alkhalaf, M.M.M. Mahmoud, M.A. Saber, Ayman M. Bahaa Eldin, Tomonobu Senjyu, and Abdullah H. Alayed, "Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor," *Ain Shams Engineering Journal*, vol. 11, no. 4, pp. 1179-1190, 2020.

APPENDIX

TABLE I: Performance Comparison of Different Convolution Implementations

Test Case	Input Shape	Weight Shape	Original (ms)	Optimized (ms)	Optimized New MatMul (ms)
1	(1, 3, 64, 64)	(32, 3, 7, 7)	22	10	7
2	(1, 32, 64, 64)	(32, 1, 7, 7)	9	80	117
3	(1, 32, 64, 64)	(64, 32, 3, 3)	92	22	19
4	(1, 64, 64, 64)	(32, 64, 3, 3)	103	29	30
5	(1, 32, 64, 64)	(32, 32, 3, 3)	13	4	3
6	(1, 32, 32, 32)	(32, 1, 7, 7)	2	17	22
7	(1, 32, 32, 32)	(64, 32, 3, 3)	23	5	4
8	(1, 64, 32, 32)	(32, 64, 3, 3)	25	6	6
9	(1, 64, 32, 32)	(64, 64, 3, 3)	12	3	2
10	(1, 64, 16, 16)	(64, 1, 7, 7)	1	15	17
11	(1, 64, 16, 16)	(128, 64, 3, 3)	22	5	4
12	(1, 128, 16, 16)	(64, 128, 3, 3)	23	5	4
13	(1, 128, 8, 8)	(128, 1, 7, 7)	0	14	14
14	(1, 128, 8, 8)	(256, 128, 3, 3)	22	5	4
15	(1, 256, 8, 8)	(128, 256, 3, 3)	23	6	4
16	(1, 256, 8, 8)	(256, 1, 7, 7)	1	59	50
17	(1, 256, 8, 8)	(512, 256, 3, 3)	90	22	16
18	(1, 512, 8, 8)	(256, 512, 3, 3)	90	28	16
19	(1, 512, 8, 8)	(256, 256, 1, 1)	10	2	1
20	(1, 256, 8, 8)	(1, 256, 1, 1)	0	0	0
21	(1, 384, 8, 8)	(384, 1, 7, 7)	1	131	117
22	(1, 384, 8, 8)	(512, 384, 3, 3)	135	42	24
23	(1, 512, 8, 8)	(256, 512, 3, 3)	151	48	17
24	(1, 256, 16, 16)	(128, 256, 3, 3)	92	26	16
25	(1, 128, 16, 16)	(1, 128, 1, 1)	0	0	0
26	(1, 96, 32, 32)	(96, 1, 7, 7)	6	138	153
27	(1, 96, 32, 32)	(128, 96, 3, 3)	138	31	24
28	(1, 128, 32, 32)	(64, 128, 3, 3)	93	24	20
29	(1, 64, 64, 64)	(32, 64, 3, 3)	102	27	29
30	(1, 32, 64, 64)	(3, 32, 1, 1)	1	0	0