Accelerating Diffusion Model on Embedded CPU

Hongseung Yu
Seoul National University
Seoul, South Korea
appleyu1@snu.ac.kr

***Abstract*—The diffusion model is based on a U-Net structure that includes many convolution operations, and it requires repeated sampling steps to generate a clean image, which demands substantial computing resources. In Project 1, the end-to-end sampling time was reduced in a 6-core CPU environment on the Jetson Orin Nano by utilizing OpenMP and ARM neon SIMD. For these optimizations, the im2col + GeMM approach was applied to convolution operations, and the layernorm function was improved, leading to a significant increase in overall execution speed. In particular, the memory access pattern of im2col was improved using a row-major layout, and SIMD operations were efficiently utilized through a GeMM kernel. In addition, kernel fusion between conv2d and the GELU layer was applied to minimize data transfer and achieve further performance improvement. As a result, the sampling time was reduced from the baseline of 380.3 seconds to 28.0 seconds, achieving a 13.58x speedup.**

***Index Terms*—Diffusion model, Convolution optimization, Kernel fusion, OpenMP, ARM neon SIMD**

I. Introduction

Among various image generation models such as Generative adversarial network (GAN) and Variational autoencoder (VAE), the diffusion model is currently one of the most widely used Neural network models due to its strong generation quality. [1]–[3] The diffusion model can be used for a wide range of image processing tasks, such as generating an image from a text prompt or transforming an existing image into a new one. The model takes complete 2D random noise, an embedding vector, and time information as input, and through the sampling process, gradually changes the RGB values of each pixel to produce an image output. This sampling process is repeated at every time step to remove noise and generate a clean image. In other words, since many sampling iterations are required for end-to-end image generation, the overall execution time becomes very long from a computing perspective.

Project 1 aims to minimize the sampling time of a diffusion model in a CPU environment with a limited number of cores and limited memory bandwidth. To reduce sampling latency, it is important to actively exploit parallelism in the computation. The NVIDIA Jetson orin nano used for experiments and performance measurements has a total of 6 cores, and fully utilizing all cores is the first important factor. In addition, each core has 32 registers and a SIMD ALU for Single instruction multiple data (SIMD) operations. When the same operation must be applied to a large amount of data, SIMD intrinsic functions should be used appropriately.

The final important factor is the time caused by data transfer between levels of the memory hierarchy. When the memory layout of data changes, the form of the optimized computation kernel must also change accordingly. In particular, an implementation that uses all 6 cores and applies SIMD operations for optimization can sometimes become less efficient depending on factors such as data size and layout. The same applies to the process of loading and storing very large data for simple operations. By applying appropriate kernel fusion techniques and implementing optimized kernels based on the characteristics of the data, inefficient data movement in the overall system can be reduced.

In Project 1, optimized kernels were implemented using openMP and arm neon SIMD intrinsic functions based on the above principles, and appropriate layer fusion was applied. As a result, the End-to-End generation time was reduced from the baseline of 380.3s to 28.0s, achieving a 13.58x speedup.

II. Background

A. U-net structure in Diffusion model

The diffusion model is a model trained to reverse the forward process that adds random noise to an image. The forward process that adds noise can be represented as an SDE. Unlike an ODE, however, in an SDE, the stochastic nature of the process makes it impossible to recover the input from the output through a direct inverse operation. However, by adding a score function term, the reverse-time SDE corresponding to the forward-time SDE can be derived mathematically. [4] The score function contains information about the image before

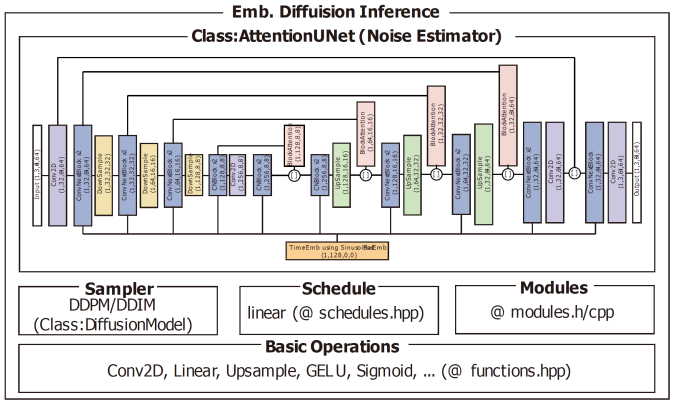

Fig. 1. Overall components of Project 1, the U-Net structure of the Diffusion model, and the modules used in the model. The structure repeatedly applies ConvNextBlocks that receive Time embedding as input between Downsample and Upsample.

noise is added, so there is no direct way to know it during image generation. Instead, this score function information can be learned through a neural network and used in various image generation tasks.

The neural network trained in this way is called a score network, and the U-Net structure is commonly used as a representative architecture. The first half of U-Net reduces the spatial dimension by repeatedly applying downsampling and convolution, while the second half increases the spatial dimension again through convolution and upsampling. Another characteristic is that residual connections exist between activations with the same spatial dimension in the convolution stages of the downsampling and upsampling paths. **Fig. 1** shows the overall structure of Project 1 and the structure of the target model implemented with U-Net. Similar to a typical U-Net structure, the main operations consist of downsample and upsample processes, as well as ConvNextBlock modules that include convolution. The ConvNextBlock in the downsample path is connected through a residual connection to the ConvNextBlock in the upsample path after BlockAttention. In addition, all ConvNextBlocks receive and process time embedding information generated at every time step.

B. GeMM in Convolution layer

To optimize convolution operations, a widely used approach is to transform the input image into a col matrix and multiply it with the weights using GeMM. This approach is called im2col + GeMM. GeMM is a fundamental operation used in many workloads, and because it can divide the workload in parallel through tiling on computing units with a memory hierarchy, optimized kernels and libraries exist for various hardware platforms. [5], [6]

TABLE I
FUNCTION TIME BREAKDOWN IN END-TO-END GENERATION

Function	Conv2D	LayerNorm	End-to-End Generation
Time(s)	361.4	17.2	380.3

Algorithm 1: Column Major Im2Col Operation

```
#pragma omp parallel for
for h = 0 to out_height − 1 do
    for w = 0 to out_width − 1 do
        for c = 0 to in_channels − 1 do
            for kh = 0 to kernel_size − 1 do
                for kw = 0 to kernel_size − 1 do
                    in_h = h × stride + kh − padding;
                    in_w = w × stride + kw − padding;
                    if 0 ≤ in_h < in_height and
                     0 ≤ in_w < in_width then
                        col_matrix(h, w, c, kh, kw) =
                         input(c, in_h, in_w);
```

When GeMM is performed after im2col, if the size of inputA corresponding to the weight is M x K, the size of inputB corresponding to the col matrix is K x N, and the output size is M x N, then M, N, and K can be expressed using #out_channels, kernel_size, out_height, out_width, and #in_channels in the convolution operation as follows.

$$M = \#out\_channels \tag{1}$$

$$K = \#in\_channels * kernel\_size * kernel\_size \tag{2}$$

$$N = out\_height * out\_width \tag{3}$$

III. OPTIMIZATION METHODS

This section classifies the optimization methods applied in Project 1 into three categories according to their methodology. These three categories are organized in the order in which the actual optimizations were performed. **Table I** shows the measured results when conv2d, layernorm, and other functions implemented in function.hpp were commented out in the Baseline code and replaced with tensors filled with zeros. The functions that accounted for the largest portion of the execution time were conv2d and layernorm, and the optimization was mainly focused on these two functions. After optimizing each function, layer fusion and kernel fusion were applied to minimize data transfer.

A. *conv2d function*

As shown in **Fig. 1**, convolution operations repeatedly appear in ConvNextBlock and BlockAttention, making conv2d the highest-priority target for optimization. The Conv2d function went through multiple revisions, which can be divided into three approaches according to the order of optimization. The degree of performance improvement for each approach is presented in Experiments.

Transposed GeMM The existing naïve convolution implementation was completely removed, and column major im2col was first performed, followed by a transposed GeMM operation. As shown in **Algorithm 1**, during im2col, the c, kh, and kw loops are placed inside the h and w loops so that the values belonging to the same column are stored contiguously in memory. In this case, as shown in Equations (2) and (3), input B is stored in a transposed N x K form, which was expected to be advantageous when performing vector multiplication followed by reduction during the GeMM operation.

A Transposed GeMM kernel was implemented to reflect this characteristic, and the basic unit of computation inside the kernel is shown in **Fig. 2**. First, vectors of size 4 are loaded from inputA and inputB using SIMD, and the two vectors are multiplied elementwise using the vmulq_f32 operation. Then, the final sum is computed from the resulting mul_vec using an adder tree structure. In addition, the M x N output is divided using openMP so that each core computes 1/6 of the output. On the target hardware, Jetson Orin nano, the L1 cache line size is 64 bytes, so one cache line can contain 16 float values, each of which is 4 bytes. Considering this, the kernel was designed so that the operation in **Fig. 2** is repeated four times in the inner loop, allowing all values in a loaded cache line to be processed before computing another output tile.

If all Transposed GeMM operations are handled by a single kernel, additional checks are required for parts that are not aligned to SIMD, which reduces overall performance. Therefore, the values of M, N, and K were first classified according to their values and sizes, and an optimized kernel was executed for each M, N, and K configuration. In particular, for operations with large M, N, and K values, the dimensions were often aligned to multiples of 16. For those kernels, higher performance was expected because exception-handling code was not included.

Naïve conv + Transposed GeMM During the process of implementing different optimized kernels for each M, N, and K value, it was observed that the convolution operations could

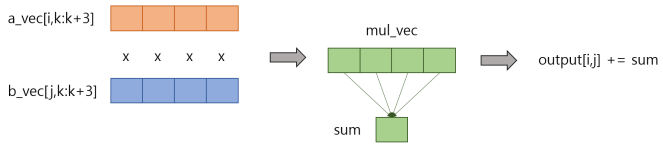

Fig. 2. Visualization of the computation performed in the innermost for loop of the Transposed GeMM kernel

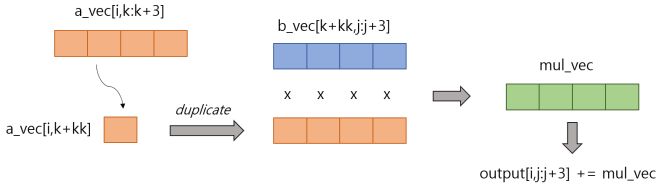

Fig. 3. Visualization of the basic operation in the GeMM kernel. Although mul_vec is not explicitly computed when calculating output[i,j:j+3], it is shown for visual explanation.

be broadly divided into two types. The first type has a group size of 32 or larger, with M = 1 and K = 49. The second type consists of operations where the group size is 1 and M, N, and K are multiples of 4 or 32. The first type corresponds to the process of dividing the first input in ConvNextBlock into groups of size #in_channels and performing convolution, while the second type corresponds to convolution operations used in the other layers. Since ConvNextBlock appears 20 times in total in **Fig. 1**, the first type of operation is small in size but is executed many times throughout the overall process. In a GeMM-based implementation, the entire image is first scanned to create a new col matrix in memory, and the col matrix is then loaded again for the GeMM operation. Therefore, compared to a naïve convolution implementation, the number of memory accesses increases. If the image size is large, kernel optimization can offset this overhead. However, for a small image as in the first type of operation, a naïve implementation that completes the computation with a single access can reduce access overhead and be more efficient in terms of overall latency. Therefore, for small images with group_size ¿= 32 and N ¡ 4096, the implementation was configured to use the original code.

Naïve Conv + GeMM The previous transposed GeMM implementation had two problems that limited performance improvement. The first problem was the reduction operation. In **Fig. 2**, after four elements are multiplied, the result is finally reduced into a single sum. In other words, computing one output element requires K / 4 reduction operations, so even if memory access is efficient, there is still computational inefficiency. The second problem was the inefficient memory

access order when converting the input into a col matrix in the im2col operation. In the loop order shown in **Algorithm 1**, the input values required for a fixed output pixel are stored contiguously, so the loop over in_channels is placed inside the loops over out_height and out_width. Since in_channels is the most major dimension in the input, iterating over in_channel for every output value prevents the use of spatial locality during input access and can cause cache misses.

Considering these two observations, the loop order in im2col was adjusted so that the col matrix is stored in memory in a row-major layout, and the GeMM kernel was newly implemented. **Fig. 3** shows the basic operation of the newly implemented GeMM kernel. Unlike **Fig. 2**, the elements of b_vec in inputB store different output values, so sequential multiply-and-accumulate operations are performed at the element level. In the actual implementation, mul_vec is not explicitly computed as a fused operation; instead, the multiplied value is immediately added to the previous accumulation value. Similarly, to maximize cache line utilization through temporal locality, the computation processes 16 consecutive columns in the K x N inputB at once. When SIMD operations are used for inputA, four elements are loaded along the K dimension. Using the vmlaq_laneq_f32 intrinsic, a new vector is loaded from inputB for four consecutive k values, and to utilize temporal locality for inputA as well, computations are performed over 16 consecutive columns. As in the transposed GeMM implementation, OpenMP divides the output into 1/6 chunks so that each core performs its assigned computation.

B. layernorm function

The second optimized function is layernorm. As shown in **Table I**, it accounts for the largest portion of execution time after convolution. The inefficient part of the baseline implementation was the computation of the mean and var values. To compute the mean, the entire input is summed and then divided by the total number of data elements. For var, each element must be squared, so it takes longer than the mean computation. To optimize this, multithreading using the cores was first applied. OpenMP reduction was used to compute either the sum of the input or the sum of squared input values. In this case, each thread computes a local sum and then the final sum is computed, which limits the memory range accessed by each core and greatly improves computational efficiency. Second, the mean and variance were computed simultaneously. The baseline implementation computes the mean separately and then computes the variance. However, when each element is accessed, both the simple sum and the sum_of_squares can be computed together, allowing both values to be obtained with a single access. Finally, var can be computed using sum_of_squares - mean*mean. In addition, the previous implementation stored duplicated mean and var values for every element, but this was changed so that each is stored as a single float, improving cache efficiency. The newly implemented mean and var computation function that applies these changes is shown in **Algorithm 2**.

Algorithm 2: Compute Mean and Var

Initialize $sum \leftarrow 0.0$, $sum\_sq \leftarrow 0.0$;

#pragma omp parallel for reduction(+:sum, sum_sq)
for $i = 0$ **to** $total\_elems - 1$ **do**
  $value \leftarrow x[i]$;
  $sum \leftarrow sum + value$;
  $sum\_of\_squares \leftarrow sum\_sq + value \times value$;

$mean \leftarrow sum/total\_elems$;
$var \leftarrow (sum\_sq/total\_elems) - (mean \times mean)$;

C. Kernel fusion and Layer fusion

After applying the two optimizations above, each kernel had a short execution time, but the end-to-end generation time was still larger than the sum of the latencies caused by each operation. Even when the computation algorithm is optimized, if the activation or feature map passed between layers or between kernels is large, the time required to read it and write it back to memory remains unchanged and becomes a major overhead. A representative example is the layernorm optimization described earlier. The execution time could be reduced by fusing the mean computation kernel and the variance computation kernel. In the same way, fusion techniques were applied to two different locations in the overall model.

im2col + GeMM Through the optimization of the conv2d function, an im2col kernel and an optimized GeMM kernel were created. Although each kernel was optimized, the two operations are executed sequentially, so data transfer overhead between kernels still remains. If the feature map size is large, a cache miss may occur when the GeMM kernel tries to access a previously transformed part of the col matrix. Therefore, instead of constructing the entire K x N col matrix and then performing GeMM, the implementation was modified to perform im2col only for a 32 x N block, execute GeMM for that block, and then repeat im2col and GeMM. In this way, the behavior of the two kernels is divided into smaller pieces and interleaved. The block size of 32 was selected based on

TABLE II
END-TO-END LATENCY COMPARISON BETWEEN COLUMN MAJOR AND ROW MAJOR IM2COL.

Kernel	Column Major im2col	Row Major im2col
latency(s)	13.7	6.2

TABLE III
EXECUTION LATENCY OF TRANSPOSED GEMM AND GEMM FOR DIFFERENT MATRIX DIMENSIONS.

Input Dimensions	Transposed GeMM	GeMM
M = 64, K = 576, N = 4096	8.5 ms	5.2 ms
M = 128, K = 864, N = 1024	6.0 ms	4.7 ms
M = 256, K = 4608, N = 64	6.8 ms	2.1 ms

the best-performing result among multiples of 16, which is the number of elements that fit in a cache line.

conv2d function + GELU Second, fusion was applied between the conv2d layer and the GELU layer. ConvNextBlock contains many operations, and one of them executes GELU after convolution. In this part, the feature map size between kernels is large, so memory access time acts as a significant overhead. Meanwhile, in the current convolution implementation, after im2col and matmul are completed, bias is added using openMP. If the GELU operation is also performed together when the bias is added, two different functions can be executed with a single access, which was expected to provide significant performance improvement. Therefore, a new function called conv2d_gelu was created, and the implementation was modified so that this function is executed when convolution + gelu must be performed together.

IV. EXPERIMENTS

This section presents experiments on each method described in Optimization methods. First, when performing im2col in the conv2d function, the time occupied by the im2col kernel in end-to-end generation was measured according to the loop order. In addition, the speed difference between the transposed GeMM and GeMM kernels was compared for different M, K, and N values that appear as inputs to matrix multiplication after im2col during the actual sampling process. Finally, techniques such as conv2d optimization, layernorm optimization, and kernel fusion and layer fusion were added one by one, and the resulting changes in end-to-end generation latency were measured. The speedup for each optimization step was also compared and summarized relative to the baseline.

For imcol, the conv2d function was modified to implement column major im2col + transposed GeMM and row major im2col + GeMM, and the kernel execution time was measured using the time difference in end-to-end generation when the im2col operation was excluded. As shown in **Table II**, row major im2col was about 7.5 seconds faster, showing more than a 2x performance difference.

Table III shows the execution time of the transposed GeMM kernel and the GeMM kernel for three different (M, N, K) pairs. The execution time was measured directly by modifying the main.cpp and matmul.cpp functions used in the previous lab. For the transposed GeMM kernel, inputB was transposed before being provided as input, and output correctness was also verified. These three (M, N, K) values were selected from the diffusion model because they represent the largest and most characteristic input and output matrix sizes during end-to-end generation. M = 64, K = 576, N = 4096 represents a case where M is small but N is large. M = 128, K = 864, N = 1024 represents a case where M, N, and K do not differ greatly in size. M = 256, K = 4608, N = 64 represents a case where M and N are small but the accumulation dimension K is large. For M = 64, K = 576, N = 4096 and M = 128, K = 864, N = 1024, the GeMM kernel shows speedups of about 1.63x and 1.47x, respectively. In contrast, for M = 256, K = 4608, N = 64, where the accumulation dimension is large, a 3.28x speedup was observed.

Finally, **Table IV** shows the end-to-end generation time when transposed GeMM, Naïve Conv, GeMM, Layernorm, and Kernel fusion and Layer fusion were applied step by step. The results show that performance improves at every step. Since the Baseline is implemented naïvely without any optimization, simply implementing column wise im2col and optimized transposed GeMM greatly reduces the absolute latency.

V. DISCUSSION

This section analyzes the experimental results in more detail and examines whether the methods anticipated and applied in Optimization methods were reflected numerically through the experiments.

For im2col, **Table II** shows that the column major approach is inefficient when converting a feature map into a col matrix. In addition, as shown in **Algorithm 1**, threads were distributed based on out_height. In this case, every core accesses the full range of in_channel in the feature map, making it impossible to exploit spatial locality and causing cache misses, which was also experimentally verified. In the row major approach, threads are distributed based on in_channel. As a result, the feature map regions accessed by different cores do not overlap, and the access order follows the order in which data is stored in memory. Therefore, spatial locality can be

TABLE IV
PERFORMANCE COMPARISON FOR DIFFERENT OPTIMIZATIONS

	Baseline	Transposed GeMM	Naïve Conv + Transposed GeMM	Naïve Conv + GeMM	Naïve Conv + GeMM + Layernorm	Naïve Conv + GeMM + Layernorm + Kernel & Layer Fusion
Time (s)	380.3	127.6	94.3	60.1	45.6	28.0
Relative speedup from Baseline	1.00x	2.98x	4.03x	6.33x	8.34x	13.58x
Speedup from previous step	1.00x	2.98x	1.35x	1.57x	1.32x	1.63x

maximized and L1 cache misses can be minimized. Moreover, after performing im2col using the inefficient column major approach, transposed GeMM must be performed, but GeMM was also shown to have better performance than transposed GeMM. It was expected that this difference would be caused by reduction overhead, and the experiments clearly confirmed this. In the case of M = 256, K = 4608, N = 64 in **Table III**, K is relatively large at 4608, and the performance gap is larger than in the other two cases.

When K is 4608, the computation includes reduction after 1152 SIMD operations. The concrete reason for this difference lies in how each operation is implemented. In Transposed GeMM, multiplication is performed using vmulq_f32, and then reduction is performed by explicitly calling vaddq_f32, resulting in two SIMD operations. However, GeMM can use the FMA operation, an intrinsic optimized for floating point operations. FMA has been studied as a hardware circuit optimized for AxB+C operations, and many CPU manufacturers provide intrinsics that can execute it as a single operation. [7] In ARM neon, the vmlaq_laneq_f32 operation can be used, which can explain the observed performance difference. In summary, column major im2col + transposed GeMM is less efficient than the row major im2col + GeMM implementation even at the individual kernel level. This result is also reflected in the end-to-end results in **Table IV**. Even after the Naïve Conv + Transposed GeMM implementation reached 94.3 seconds, conv2d still accounted for the major portion of the execution time, and after improving the algorithm, a 1.57x speedup was observed. Considering that im2col achieved about a 2x speedup and GeMM achieved a speedup of about 1.47x to 3.24x at the single-kernel level, the overhead caused by conv2d was significantly reduced.

Table IV contains a total of five improvement steps excluding the Baseline. Among them, the steps that implemented Transposed GeMM and GeMM changed the computation algorithm, while the remaining steps, including Naïve conv, layernorm optimization, and kernel & layer fusion, improved performance by reducing data transfer. The former was performed earlier in the optimization process, and the latter was performed later. This suggests that the baseline was a compute-intensive job, but after the computation method was optimized, the workload characteristics shifted toward a memory-intensive job.

For layernorm, a direct contribution to the performance improvement appears to come from storing the mean and var values, which are the same for all elements, as single floats instead of arrays, and computing them simultaneously to improve the memory access pattern. An interesting observation is that, in the baseline results in **Table I**, layernorm accounted for 17.2 seconds of the total time, but the layernorm optimization alone reduced the time by 14.5 seconds, indicating that the function became more than 6 times faster.

Most of the performance improvement from the final kernel & layer fusion came from the fusion between the conv2d layer and the GELU layer. In the conv2d implementation, bias was added to the entire output as a final step separately from the im2col+GeMM operation. Since the GELU operation was integrated while adding the bias at the end, a clear performance improvement could be achieved. However, if bias addition had been performed inside the GeMM kernel or if bias had been added during accumulation, fusion with the GELU layer would likely not have resulted in a large performance improvement. In another interpretation, the additional latency caused by bias addition was hidden by integrating the existing bias addition bottleneck with the GELU operation.

In addition to the methods applied in Project 1, there are several additional methods worth experimenting with. The first is further improvement of the im2col+GeMM algorithm. In the final implementation, two kernels were implemented, and a fused kernel was constructed by dividing K into units of 32 and interleaving the two kernels. However, from the perspective of reducing memory bandwidth, other approaches can also be attempted. A representative example is the mGeMM operation. mGeMM identifies that the col_matrix generated by

im2col contains many duplicated elements, and it introduces a method that performs the key SIMD operation of GeMM through appropriate register mapping with a single image load, without duplicating data. [8] Since the target optimization environment, Jetson Orin Nano, is also limited by memory bandwidth, applying this type of technique appropriately could further optimize the conv2d implementation. In addition, other methods such as executing small independent operations in parallel or applying kernel fusion based on accurate profiling to optimize data movement could also be explored in future work.

ACKNOWLEDGMENT

Some help from OpenAI's ChatGPT was used in the process of implementing the ideas in code. The prompts and responses used during the process can be found through the shared link in the reference. [9]

REFERENCES

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.

[2] A. Brock, J. Donahue, and K. Simonyan, "Large scale GAN training for high fidelity natural image synthesis," in *Proc. Int. Conf. Learning Representations*, 2019.

[3] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *arXiv preprint arXiv:2006.11239*, 2020.

[4] B. D. O. Anderson, "Reverse-time diffusion equation models," *Stochastic Processes and their Applications*, vol. 12, no. 3, pp. 313–326, 1982.

[5] X. Z., "OpenBLAS: An optimized BLAS library," 2021. [Online]. Available: https://www.openblas.net/

[6] ARM, "ArmNN: Inference engine for CPUs, GPUs and NPUs," 2021. [Online]. Available: https://github.com/ARM-software/armnn

[7] E. Quinnell, E. E. Swartzlander Jr., and C. Lemonds, "Floating-point fused multiply-add architectures," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, pp. 1076–1080, 2007.

[8] J. Park, K. Bin, and K. Lee, "mGEMM: Low-latency convolution with minimal memory overhead optimized for mobile devices," in *Proc. 20th Annual Int. Conf. Mobile Systems, Applications and Services (MobiSys '22)*, pp. 222–234, June 27, 2022.

[9] "ChatGPT shared link," 2024. [Online]. Available: https://chatgpt.com/share/66ff74fd-8d34-8008-9707-c4723c8550e3.