End-to-End Optimization of a Diffusion Model on Jetson Orin Nano

Hongseung Yu
Seoul National University
appleyu1@snu.ac.kr

***Abstract*—This paper presents an end-to-end optimization of the diffusion model's U-Net architecture on the Jetson Orin Nano, focusing on performance improvements for efficient image generation. Key optimizations were categorized into data transfer minimization, kernel optimization, intra-GPU communication optimization, and computation-communication overlap.**

Data transfer minimization reduced CPU-GPU exchange to essential operations only, while kernel optimization focused on efficient convolution processing with custom CUDA kernels and fused operations to reduce memory access. Intra-GPU communication was optimized by eliminating unnecessary memory copies, and computation-communication overlap allowed concurrent processing of independent tasks, reducing idle time and hiding latency. These combined strategies achieved an end-to-end generation time of 2.5 seconds, marking a 152.1x speedup over the baseline and an 11.2x speedup over previous implementations.

I. Introduction

The diffusion model, designed for image generation tasks, is based on a U-Net architecture. U-Net is structured to progressively reduce and restore the spatial dimensions through downsampling and upsampling, where convolution operations are repeatedly applied. Notably, these convolutions are largely sequential, and the upsampling stage includes skip connections that reuse feature maps obtained from prior downsampling steps. Following Project1, Project2 aims to accelerate the sampling process by effectively leveraging these unique characteristics of the model.

A key difference between Project2 and Project1 is the ability to utilize the GPU on the Jetson Orin Nano for computations. With a significantly larger number of cores compared to the CPU, the GPU is well-suited for parallel processing. While kernel design on the CPU required efficient vector operations using SIMD, the GPU demands a kernel design leveraging CUDA C++ to make optimal use of parallel computing resources.

However, achieving performance gains through kernel optimization alone is challenging, due to the overhead associated with data transfers between the CPU and GPU. Each data transfer incurs considerable time, often offsetting the benefits of kernel optimization. Additionally, synchronization overhead further limits performance, as synchronizing CPU and GPU operations can lead to wasted GPU computational resources.

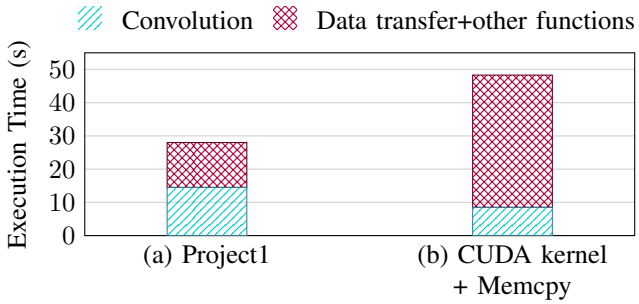


Fig. 1: Total end-to-end generation time and convolution execution time: (a) Project1 and (b) with data copy for weights and activations on each naïve convolution kernel execution.

This is especially evident when data is sent back to the CPU for verification and then returned to the GPU, creating bottlenecks.

Figure 1(b) illustrates the end-to-end generation time when a CUDA kernel is implemented for convolution operations, with weights and biases transferred at each kernel launch, based on the optimized Project1 results shown in Figure 1(a). Although convolution execution time decreased compared to Project1, the total generation time increased significantly due to GPU operations being bottlenecked by data transfers with the CPU.

The primary goal of Project2 is to minimize the overhead caused by data transfer and synchronization between the CPU and GPU. Additionally, by designing custom CUDA kernels optimized for GPU operations and applying various optimization techniques, such as kernel fusion and computation-communication overlap, the end-to-end generation time was reduced to 2.5 seconds—down from the baseline of 380.3 seconds and Project1's 28.0 seconds—achieving a speedup of 152.1x over the baseline and 11.2x over Project1.

II. Optimization Methods

This section provides a brief overview of the optimization methods used in Project2. The optimizations can be broadly categorized into three main areas: data transfer minimization, kernel optimization, and computation-communication overlap.

A. Data Transfer Minimization

Data transfers include inter-processor transfers between the CPU and GPU and intra-processor transfers within the GPU. Reducing CPU-GPU data movement minimizes operational dependency and overhead from back-and-forth transfers. To address this, all operations for each sampling step are confined to the GPU, requiring only essential transfers—receiving and returning the image—thereby lowering data transfer overhead.

Minimizing intra-processor transfers within the GPU involves eliminating unnecessary copy operations, such as data duplication or relocation. Frequently used data, like weights and biases, is stored once for consistent access. Outputs from one module that serve as inputs for the next are declared as global variables, allowing direct reuse across computations and reducing redundant data movement within the GPU.

B. CUDA Kernel Optimization

Moving all computations to the GPU helps reduce the communication overhead observed in Figure 1(b), making CUDA kernel optimization crucial for further reducing end-to-end generation time. Similar to Project1, the im2col + matmul approach was applied to optimize convolution operations effectively.

a) Im2col Operation: Each thread in the im2col operation stores a block of elements in the column matrix, corresponding to `kernel_size x kernel_size`. A 2D thread block configuration was used to allow input elements accessed within the thread block to be shared across threads, enhancing cache locality.

b) Matrix Multiplication: Matrix multiplication was implemented with a kernel utilizing tensor cores in BF16 format, with the bias added at the final computation stage.

c) LayerNorm Kernel: For the LayerNorm kernel, the reduction kernel developed in lab4 was applied to optimize the computation of mean and variance.

d) Kernel Fusion: Kernel fusion was applied to maximize computation whenever feature map data is loaded onto CUDA cores, reducing the need for repeated data movement.

e) Template Variables: Finally, in each optimized kernel, parameters such as `input_width`, `kernel_size`, and `in_channels` were defined as template variables, reducing instruction overhead at compile time.

C. Computation-Communication Overlap

While minimizing CPU-GPU data transfer reduces overhead, eliminating it completely is impractical. Performing tasks independent of the data transfer concurrently helps avoid idle processor states, reducing synchronization delays and partially hiding communication latency.

Asynchronously copying weights and biases to the GPU allows the CPU to preemptively execute the forward function and launch kernels. Even if not all data is copied, computations using the available data can start on the GPU. While the GPU sends the sampled image back to the CPU, the CPU can perform denoising refinement, and during the next data transfer to the GPU, independent operations like `time_mlp` run in parallel.

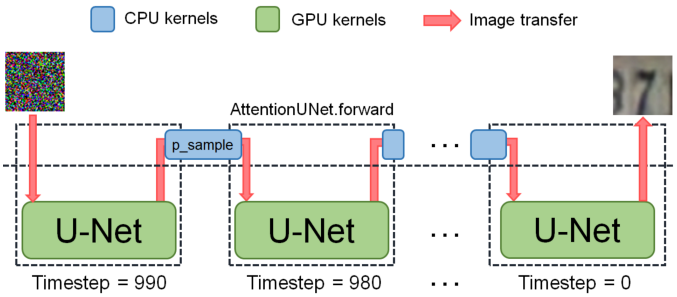


Fig. 2: Overview of processor computations and data transfers during the generation of a sampled image. At each timestep, the sampled image moves from GPU to CPU for denoising refinement, then returns to the GPU to continue the next timestep.

III. Implementation Details

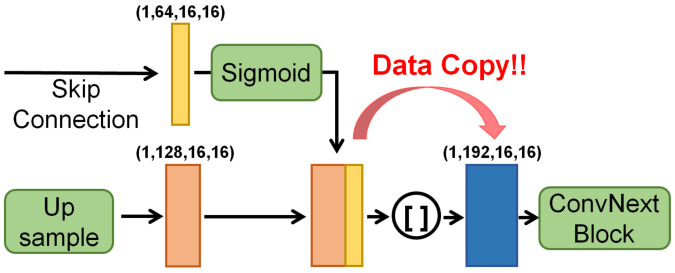
A. CPU-GPU Transfer Minimization

To reduce dependencies between the CPU and GPU, it is ideal to perform computations predominantly on a single type of processor. In Project1, where all computations were performed on the CPU, computation itself became a primary bottleneck. In this project, however, it is preferable to handle all computations on the GPU, which is specialized for parallel processing.

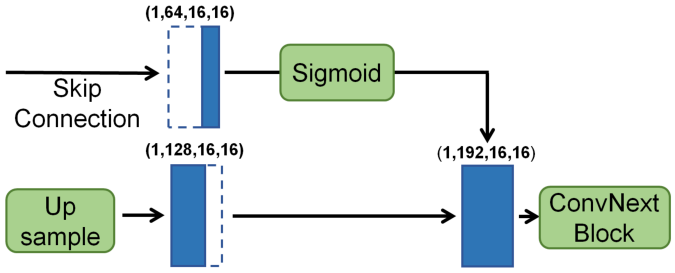
Nevertheless, over the course of 100 timesteps, the entire sampling process cannot be fully offloaded to the GPU, as some parts of the image processing must still be executed on the CPU after each timestep. Thus, as shown in Figure 2, data transfer between the CPU and GPU was minimized to include only the exchange of images at each timestep and the transfer of the initial and final sampled images. Although not depicted in Figure 2, weights and biases were transferred to the GPU before the first timestep.

B. Pre-Allocated Memory for Skip Connection

During the sampling process with the U-Net structure offloaded to the GPU, data copy operations between different memory spaces are often necessary. Figure 3(a) illustrates the concatenation process, where the feature map produced by the upsampling layer and the feature map from the downsampling stage undergo an in-place sigmoid operation before concatenation. If these two feature maps are stored in different memory locations, a new space is required to store the concatenated result, and data must be copied into



(a) Naïve Concatenation



(b) Pre-Allocated Memory

Fig. 3: Comparison of (a) copying data to a new memory space when concatenating feature maps generated from the downsampling process and the upsampled feature maps, versus (b) optimizing by storing the output feature map in pre-allocated memory to eliminate the need for explicit copying.

this space before proceeding with the next `ConvNextBlock` computation.

To prevent this, the feature maps with skip connections from the downsampling stage were allocated specific offsets, allowing the results to be stored directly in locations where they can be immediately used as inputs for `ConvNextBlock` (Figure 3(b)). Since most modules have fixed preceding and succeeding modules, the memory addresses for outputs and inputs can be predefined. These addresses were declared as global variables, accessible by all modules, and stored as member variables within each module. As a result, data transfers within the GPU were entirely eliminated during a single sampling process.

C. Kernel Optimizations

For the high computational demands of convolution, most cases utilized the im2col + matmul approach. However, in the case of `ConvNextBlock`'s first module, `in_conv`, a naïve convolution method was used due to the group size being greater than 1, requiring repeated matmul operations for each group.

The im2col operation stores all required input elements in the same column for computing each convolution output element. Figure 4 illustrates how an 8 x 8 thread block accesses input elements to form the column matrix. To compute a single output element, the number of required input elements is `in_channels x kernel_size x kernel_size`, and each thread is assigned to store `1 x kernel_size x kernel_size` elements in the column matrix. The thread block is configured as a 2D block matching the spatial dimensions of the convolution output matrix.

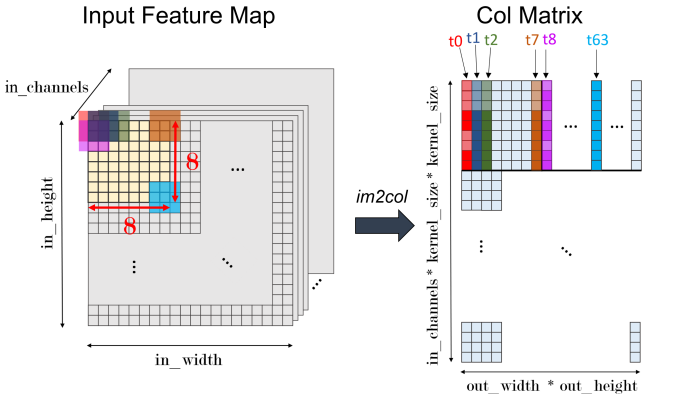


Fig. 4: Process of an 8 x 8 thread block accessing input elements to generate the column matrix, assuming `kernel_size = 3`, `stride = 1`, and `padding = 1`, resulting in equal spatial dimensions for the convolution input and output.

As shown in Figure 4, the 2D thread block configuration allows threads to share a large number of input elements, thereby utilizing cache temporal locality. To optimize spatial locality, the column matrix is stored in row-major order to prevent large strides in memory addresses between adjacent threads, which would occur with column-major storage. The output matrix is stored in BF16 format to prepare for the tensor-core-based matmul operation that follows.

For matrix multiplication, when M, K, and N are multiples of 32, tensor cores with a fragment size of $16 \times 16 \times 16$ were used. In other cases, each thread computed a single element in the output matrix due to generally smaller output sizes. With tensor cores, each warp in a thread block handled a 16×16 output tile, and the number of tiles per block was controlled by adjusting warp count. Since M and N were often small, grid size was expanded to increase parallelism rather than enlarging block size. Bias was added in a separate addition kernel after completing the matmul operation.

LayerNorm involves executing three kernels sequentially, as shown in Figure 5. In Kernel 1, each thread block calculates the sum and sum of squares for the input elements in parallel. Kernel 2 performs a reduction on these results from each block to determine the final mean and variance. Attempting to compute both mean and variance in a single kernel would require atomic additions or inter-block communication, which could hinder parallelization, so the process was split into two kernels. The final kernel then performs in-place normalization, scaling, and shifting using the calculated mean and variance.

Kernel fusion combines lightweight kernels like ReLU, GELU, and addition with preceding kernels to reduce data

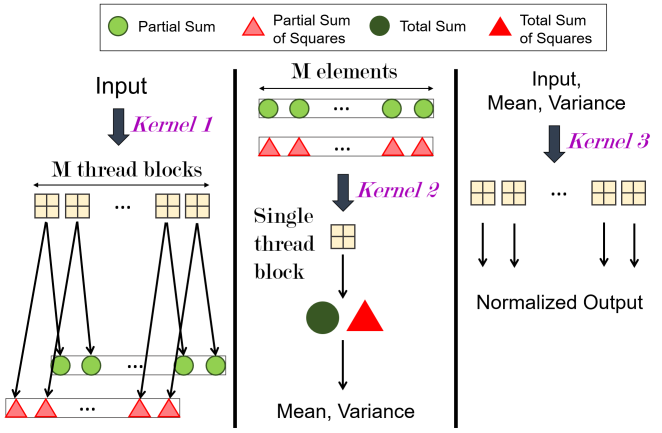


Fig. 5: Structure of LayerNorm, consisting of Kernel 1, which computes partial sums and partial sums of squares for each thread block on the input; Kernel 2, which reduces these results; and Kernel 3, which performs normalization, scaling, and shifting.

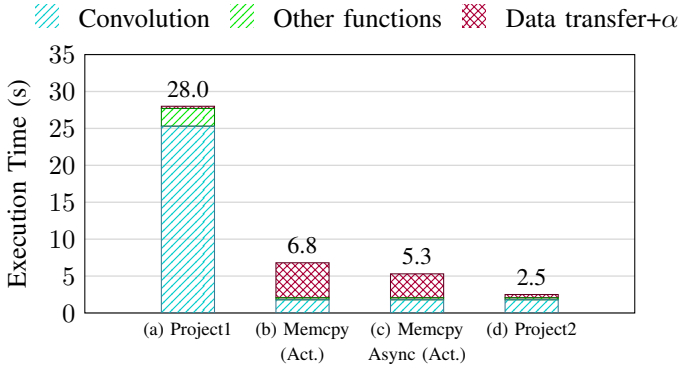


Fig. 6: End-to-end generation time. Each entry represents the stacked execution time for convolution, other functions, and data transfer and else. (a) Project1 (b) Activation Memcpy, (c) Activation MemcpyAsync, (d) Project2.

transfer. For example, in `ConvNextBlock`, the GELU operation after matmul was fused with bias addition, and in `BlockAttention`, the addition of `gate_conv` and `residual_conv` with ReLU was merged into a single kernel. For tensor-core-based matmul, bias addition was fused with GELU, ReLU, and other additions to mitigate separate kernel overhead.

At each timestep, values like input width, kernel size, and in channels are known at compile time. For instance, `in_width` and `in_height` are equal and take values like 8, 16, 32, or 64, while im2col uses fixed configurations for `kernel_size`, `stride`, and `padding`. Linear layer inputs are typically 32 or 128, and matmul kernels are split by K. These constants were set as template variables for compile-time optimization. However, to avoid overhead from excessive kernel variety, such as TLB misses, template variables were selectively used based on performance experiments.

D. Asynchronous Data Copy

Synchronization during CPU-GPU data transfer often leads to significant overhead by wasting GPU computational resources. Asynchronous data transfer allows data copying, CPU computation, and kernel launches to occur independently. To implement this, weights and biases were asynchronously copied to the GPU, enabling the CPU to execute the forward function and launch the next kernel while data is transferred. At the start and end of each timestep, asynchronous image transfer between CPU and GPU prevents idle time, allowing GPU U-Net operations and CPU `p_sample` computations to overlap, as shown in Figure 2(a).

E. Early Start of Computation with Preloaded Weights

Weights and biases needed for sampling must be transferred to the GPU before sampling begins. If the entire transfer must complete before sampling starts, it can delay the overall process due to the large data size. Therefore, weights and biases were copied in sequence, allowing computations to begin as soon as each module's data transfer completed. Since the diffusion model reuses the loaded weights and biases across 100 timesteps, completing the first step still requires all data to be loaded. This approach effectively reduces 100 steps to 99 in practice, and was implemented using CUDA's event synchronization.

IV. Experiments and Discussion

A. Data Transfer and Synchronization Overhead Analysis

Figures 6(a) and 6(d) illustrate the proportion of total time consumed by convolution, other functions, and data copy operations in Project1 and Project2. As shown, convolution occupies the largest share in both projects, accounting for 90.4% in Project1 and 71.6% in Project2 (see Table I). This higher proportion in Project1 reflects the intensive parallel processing demands of convolution operations, for which the GPU in Project2 is a more suitable processor.

To evaluate the overhead caused by data copying between the CPU and GPU, activations were asynchronously transferred to and from the CPU before and after each kernel execution in the final Project2 code. Additionally, synchronous (Figure 6(b)) and asynchronous (Figure 6(c)) copies of activations were compared to observe the impact of synchronization overhead. Results showed that asynchronous copies increased the total runtime by 2.12x, while synchronous copies led to a 2.72x increase. As described in Figure 2, data transfer was limited to image exchanges with the CPU at each timestep, comprising only 17.6% of the total time. However, when copying activations, data transfer occupied over 60% of the total

TABLE I: Execution Time Breakdown by Component (Time in seconds and percentage)

	Project1	**MemCpy (Act.)**	**MemCpyAsync (Act.)**	**Project2**
Convolution	25.3 (90.4%)	1.79 (26.3%)	1.79 (33.8%)	1.79 (71.6%)
Other Functions	2.4 (8.6%)	0.27 (4.0%)	0.27 (5.1%)	0.27 (10.8%)
Data Transfer+α	0.3 (1.0%)	**4.74 (69.7%)**	**3.24 (61.1%)**	**0.44 (17.6%)**
Total Time (s)	28.0 (100%)	6.8 (100%)	5.3 (100%)	2.5 (100%)

TABLE II: Weight/Bias Load to First Iteration: Worst, Best, Avg Times (N = 20)

Time (ms)	**MemCpy (W/B)**	**MemCpyAsync (W/B)**	**Early Start**
Best	282.8	230.6	233.1
Worst	497.7	1092.4	366.6
Average	345.7	424.989	266.4

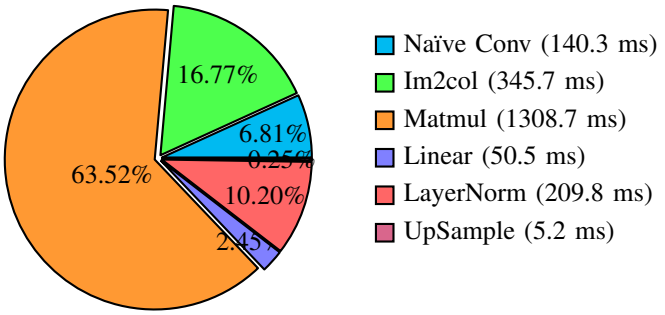


Fig. 7: Kernel Execution Time Distribution for Each Operation (in ms)

runtime—61.1% for asynchronous and 69.7% for synchronous copies.

Meanwhile, weights and biases are transferred only once before the first timestep. Table 2 shows the time taken to receive these weights and biases and execute the first timestep. Due to variation in each run, best, worst, and average times were recorded over 20 executions. For synchronous copying, the best time was 282.8 ms, slower than both asynchronous copying and early start. However, the worst and average times for asynchronous copying were slower than those for synchronous copying, likely due to greater variability across executions.

For early start, the best time was 233.1 ms, slightly slower than the best time for asynchronous copying, indicating no performance improvement. This outcome is reasonable, as completing the first step requires all weights and biases to be fully transferred, and each sampling step has a latency of around 10–20 ms. An interesting observation is that early start showed the smallest variation in weight and bias copy time, resulting in the shortest average time. Although further experimentation and analysis are needed to confirm the exact cause, it was noted that weight and bias copy time constitutes the majority of Data Transfer + α in Table 1's Project2, affecting overall execution time variability. Consequently, early start was implemented in the final Project2 code to ensure stable results.

B. Efficeincy of Copy Operations on Unified Memory

This section discusses the efficiency of using copy operations on the Jetson Orin Nano with unified memory. To leverage the unified memory structure, memory is allocated with `cudaMallocManaged`, allowing both CPU and GPU access. For the GPU to reference memory previously accessed by the CPU, methods such as 1) on-demand paging and 2) prefetching are used [1]. On-demand paging relies on the operating system's page fault mechanism, introducing overhead as fault handling occurs internally when the GPU references the location. Prefetching can be used if the memory location will be accessed by the GPU in the future, reducing latency. Attempts were made to implement prefetching for loading weights and biases, but several issues arose.

Initially, an issue arose when prefetching data from the CPU to the GPU after reading from an `.npy` file, leading to segmentation faults. The U-Net contains multiple modules, and each module's weights were prefetched immediately after loading, causing subsequent reads from other `.npy` files to prefetch unintended locations to the GPU. To address this, the initial memory allocation for weights was modified with `cudaMemAttachHost` during `cudaMallocManaged`, allowing CPU access until explicit prefetching was performed. However, this approach led to a drastic increase in end-to-end generation time (over 50 seconds), likely due to lack of GPU caching at these memory locations. Attempts were made to load all weights and biases on the CPU, then prefetch to the GPU before computation, but repeated errors prevented further testing. Future efforts will focus on resolving these issues to enable performance comparisons.

C. Kernel Execution Time Analysis

To determine the proportion of end-to-end generation time spent on key kernels in Project2, execution times were measured based on timestamps taken before and after each kernel launch within kernel.cu. Figure 7 shows the execution time for each kernel. When summing the times for naïve conv, im2col, and matmul—all related to convolution—these nearly match the convolution time reported for Project2 in Table 1. Among these, matmul accounts for the largest share of time, though im2col also contributes significantly at 16.77%, over one-fourth of the matmul time. This disparity arises because matmul benefits from tensor cores, specialized hardware that enables highly efficient computation.

Another key factor is the optimization achieved by using BF16 for computations. This quantization technique not only reduces data size but also enhances computational efficiency. BF16, which reduces FP32's 23-bit fraction to just 7 bits, allows for faster execution of operations like addition and multiplication due to its simpler logic requirements [2]. Addi-

tionally, by using mixed precision calculations—storing results in FP32 instead of BF16 format—computation speed can be enhanced while maintaining accuracy [3]. However, BF16 still includes an exponent bit, requiring shift operations that may slow down processing. Recent research has aimed to design efficient hardware using fixed-point formats like INT8 or INT4, as well as formats like BFP and MXFP, where multiple elements share a single exponent to further optimize performance [4], [5].

For layernorm, three kernels are executed sequentially, with kernel 2 performing a reduction on a single thread block, which may lead to underutilized computing resources (Figure 5). However, due to the nature of reduction, calculating a single mean and variance inherently limits parallelism. The separation into kernel 1, which maximizes parallelism, and kernel 2, which finalizes the mean and variance calculation, is likely an efficient design choice.

ACKNOWLEDGMENT

In the process of implementing ideas into code, I received assistance from OpenAI's ChatGPT. The prompts and responses used can be viewed through the reference link [6].

REFERENCES

[1] N. Sakharnykh, "Maximizing Unified Memory Performance in CUDA" [Online]. Available: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, *et al.*, "BFloat16: The secret to high performance on Cloud TPUs," in *ISCA*, 2018.

[3] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, *et al.*, "Mixed precision training," in *ICLR*, 2018.

[4] Y.-C. Lo and R.-S. Liu, "Bucket Getter: A bucket-based processing engine for low-bit block floating point (BFP) DNNs," in *MICRO*, 2023.

[5] B. D. Rouhani, *et al.*, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.

[6] OpenAI, "ChatGPT" [Online]. Available: https://chatgpt.com/share/6727bea8-a910-8008-bb9e-5d9a8fa0f1f8