U-Net Acceleration on Zynq-7000 FPGA with High-Level Synthesis

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

***Abstract*—This paper presents an end-to-end optimization of the U-Net architecture in a diffusion model, implemented on the Xilinx Zynq-7000 FPGA, with a focus on accelerating image generation tasks. The project emphasizes hardware-specific optimization, efficient kernel execution, and reduction of communication overhead between processing elements.**

Key optimizations were categorized into HLS-based PL (programmable logic) acceleration, CPU-side optimization, and PS-PL communication minimization. The PL design focused on efficient matrix multiplication using tiling strategies tailored to FPGA resource constraints, while CPU-side optimization minimized memory access bottlenecks and reduced redundant data transfers. Communication overhead was mitigated by refining shared memory operations and optimizing data flow between the PS and PL to improve efficiency.

These strategies achieved a total execution time of 4.26 seconds, surpassing the performance target of 7.5 seconds and representing a 108.4x speedup over the baseline.

I. Introduction

The diffusion model is designed for image generation tasks and is based on a U-Net architecture, which repeatedly reduces and restores spatial dimensions through downsampling and upsampling processes. This U-Net structure features sequential operations and complex memory usage patterns due to skip connections. In Project 3, following Project 1 and Project 2, the primary goal is to accelerate the U-Net sampling process.

In Project 3, unlike the previous projects, hardware acceleration is implemented using High-Level Synthesis (HLS) tools on Xilinx's Zynq-7000 FPGA. The FPGA features programmable logic (PL), enabling custom hardware designs tailored to various applications. HLS allows for the straightforward conversion of C++ code into hardware implementations, offering a significant advantage.

Additionally, the processing system (PS) includes an ARM Cortex-A9 dual-core CPU, and the PS and PL are interconnected via an AXI interface. This setup enables the declaration of shared memory in DRAM for efficient data exchange between the two components [1]. Compute-intensive operations, such as convolution, are executed on the PL, while other operations are handled by the PS. Optimizing the computations within each domain is expected to result in overall performance improvements.

Efficient use of the PL requires optimizing the utilization of BRAM, which serves as a cache, and DSP resources for arithmetic operations. Even with high computing capabilities, frequent or inefficient DRAM access patterns can lead to significant DSP resource idle time. Conversely, insufficient computing resources can increase overall latency, making it crucial to balance these resources effectively. Similarly, on the PS side, efficient use of the dual-core CPU and optimizing memory access patterns are essential for achieving better performance.

Optimizing the PS and PL individually is indeed critical for overall performance improvement. However, as seen in Project 2, where minimizing data transfer and synchronization overhead between the GPU and CPU was crucial, reducing communication overhead between the two domains will be the key to enhancing performance in this project as well.

Taking these factors into account, Project 3 focused on optimizing individual functions and kernels through efficient hardware design and CPU optimization while minimizing data transfer and synchronization overhead between the PS and PL. As a result, the execution time was reduced from the baseline of 455.4 seconds to 4.26 seconds, achieving a 108.4x speedup.

II. Methods

The optimization methods applied in Project 3 can be categorized into three main areas: HLS design for the PL, CPU optimization for the PS, and improving the connection between the PS and PL.

- For the PL, hardware was designed to efficiently perform matrix multiplication (matmul), the most computationally intensive operation in the U-Net structure's convolution layers.
- On the PS side, most operations, aside from convolution, were memory-intensive, so the focus was on optimizing memory access patterns.
- Lastly, various experiments were conducted on shared memory between the PS and PL to identify optimal methods for reducing communication overhead. Details of these experiments are covered in Section IV-C.

A. PL Optimization

Convolution operations can be executed using either the direct convolution method or a combination of image-to-column (im2col) and matrix multiplication (matmul). To leverage previously applied matmul optimization techniques, the im2col + matmul approach was selected. When executed

TABLE I: Baseline Execution Time Breakdown

Kernel	Time (s)	Percentage (%)
Matmul	444.742	97.29
Im2col	1.720	0.38
Layernorm	0.901	0.20
CpyToShm	5.121	1.12
CpyFromShm	1.222	0.27
Others	3.384	0.74
Total	**455.372**	**100.00**

on the CPU, the im2col operation accounted for only 1.7 seconds, while naïve matmul took 444.7 seconds (Table I). This indicates that the primary bottleneck in convolution lies in the matmul operation. To address this, the matmul computation was offloaded to the FPGA to efficiently utilize its limited resources.

In the U-Net structure, the matmul operation involves multiplying a weight matrix of size M x K with an input matrix of size K x N, followed by a row-wise addition of a bias vector of size M. The dimensions M, K, and N are defined as follows:

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

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

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

$$\begin{aligned} M &= out\_channels \\ K &= in\_channels \cdot kernel\_size^2 \\ N &= out\_height \cdot out\_width \end{aligned}$$

$$\begin{aligned} \mathbf{q} &: seq\_len \times d_{\text{head}} \\ \mathbf{k}^\top &: d_{\text{head}} \times seq\_len \end{aligned}$$

$$\begin{aligned} M &: seq\_len \text{ of the } \mathbf{k} \\ K &: d_{\text{head}} \\ N &: seq\_len \text{ of the } \mathbf{q} \end{aligned}$$

- $N_H \geq A \gg K$: Maximizes the efficiency of the HMMU
- $N_L \ll B \ll K$: Maximizes the efficiency of the LMMU

The width and height of the images and feature maps are always equal, with possible values of 64, 32, 16, or 8. In the U-Net architecture, as the spatial dimensions decrease during downsampling, the number of channels—and thus K—increases. Conversely, during upsampling, as N increases, K decreases.

Considering these characteristics, the matmul operations can be categorized into two cases: when N is 4096 or 1024 (high-res case) and when N is 256 or 64 (low-res case). If separate bitstreams are created for these two cases, the N values in the U-Net structure—4096, 1024, 256, 64, 256, 1024, and 4096—can be handled with only two reprogramming steps. While using distinct bitstreams for each combination of M, K, and N would optimize individual operations, it is more practical to limit the design to one or two bitstreams to minimize delays caused by reprogramming.

Using a single bitstream optimized for various M, K, and N values eliminates the additional delay caused by bitstream reprogramming entirely. Moreover, performing the same type of operation on different hardware depending on input size would be highly inefficient from a reusability perspective. A single bitstream is thus a more practical and efficient choice.

Section III introduces the design approaches of Version 1, which uses two types of bitstreams for matmul operations, and Version 2, which relies on a single bitstream for all computations. The advantages and disadvantages of each approach are analyzed. In Section IV-A, experimental results are presented to compare their performance and determine which design is more efficient.

FP16 was chosen as the additional quantization format. While 16-bit fixed-point calculations would require FP32 for accurate results, this approach does not utilize DSP resources and instead relies on LUTs, reducing computational efficiency. To achieve correct accumulation with a single fixed-point format, at least 24 bits are necessary, which still requires two DSP resources per FMA operation and consumes significant LUT and BRAM resources. In contrast, FP16 can perform two FMA operations using a single DSP in four cycles, while also requiring fewer LUTs and BRAM. This makes FP16 the most suitable choice.

B. PS Optimization

All operations, except for the matmul in convolution, are executed on the CPU within the PS, making optimization in this area crucial for overall performance. However, the ZYNQ-7000 features a dual-core CPU with only one physical thread per core, which limits the potential for parallelization in kernel optimization. Aside from matmul, most functions (e.g., activation functions and upsampling) are bottlenecked by memory access. In the U-Net structure, the presence of skip connections further complicates memory access patterns, making memory access optimization a primary focus.

First, data copying was minimized when kernels were executed consecutively. Since most modules have predefined preceding and succeeding modules, the memory addresses for outputs and inputs were predefined. These addresses were declared as global variables, allowing all modules to reference them, and were stored as member variables in each module. As a result, no data transfer occurred within the GPU during a single sampling process. For feature maps with skip connections in the downsampling process, offsets were calculated so that the results could be directly stored in the appropriate location and immediately used as inputs for the `ConvNextBlock`, eliminating the need for explicit concatenation operations.

Second, kernels were categorized based on whether in-memory operations—where outputs and inputs share the same memory location—were feasible. This allowed for memory

reuse, reducing the total memory footprint and improving the cache hit rate. For instance, operations such as activation functions (RELU, GELU), layer normalization, and elementwise multiplication/addition have outputs and inputs of the same size, with each output value depending solely on the corresponding input value at the same location. These characteristics make in-memory operations viable for such kernels.

C. PS and PL Communication Optimization

As in Project 2, where minimizing overhead from inter-processor communication was as important as optimizing each processor, reducing communication overhead between the PS and PL is also critical in Project 3. First, the U-Net constructor was designed to program the bitstream and pre-store weights in shared memory, ensuring that these tasks were excluded from the execution time of the forward function. During the forward function, weights did not need to be copied to shared memory for each run; instead, only their locations were passed to the FPGA via MMIO, significantly reducing runtime overhead. Time measurements showed that bitstream programming took approximately 0.3 to 0.4 seconds, and copying 60MB of weights required 1 second. By eliminating these steps during runtime, overall execution time was reduced. However, it should be noted that any reprogramming of the bitstream during execution is still included in the time measurements.

While storing weights in shared memory and passing their locations to the FPGA proved highly efficient, the same approach may not work as effectively for feature maps. For weights, the PS writes once and the PL only reads, but feature maps involve repeated read and write operations by both the PS and PL.

Take the `ConvNextBlock` as an example, specifically the (convolution1-GELU-layernorm-convolution2) sequence. To minimize data transfers, the output of convolution1 (shared_output) stored in shared memory must pass through in-memory GELU and layernorm operations before being used as input for convolution2. If shared_output resides in non-cacheable memory, GELU and layernorm operations cannot take advantage of spatial locality, leading to increased DRAM access. On the other hand, if shared_output is in cacheable memory, the PS must flush the cache for the PL to read the data, and the PS must invalidate the cache when the PL writes data. This means that, regardless of whether the shared memory is cacheable, additional overhead is inevitable.

To address this, computations on the PS were performed using memory accessible only by the CPU, avoiding shared memory. Data transfers between the PL and PS were handled through explicit copying for each interaction. An exception was made for the im2col operation performed before the matmul computation. The output of the im2col operation, which serves as the input for matmul, is typically large. Reloading and storing this data back on the CPU after the im2col operation would be inefficient. Therefore, the im2col output was directly stored in shared memory, allowing the matmul operation to proceed immediately without additional data transfer steps.

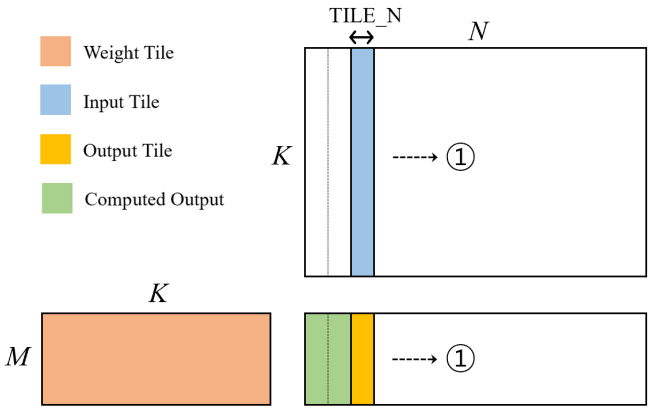

Fig. 1: Matmul flow for the high-resolution case in Version 1. The weights are fully loaded into BRAM, and input tiles of size $K \times$ TILE_N are sequentially loaded from DRAM. The resulting $M \times$ TILE_N output tiles are directly written back to DRAM.

III. HLS Design for Matmul

This section introduces three matmul algorithm designs: the high-res case in Version 1, the low-res case in Version 1, and Version 2, which is applicable to both cases.

A. Version 1: High-Res Case

The high-res case occurs when the spatial dimension, represented by N, takes on values of 1024 or 4096. In this scenario, both `in_channels` and `out_channels` are relatively small. According to equations (1) and (2), M can reach a maximum of 128, while K can go up to 1152. Consequently, M is relatively small, whereas N is significantly larger, resulting in a final output matrix of size M x N that is highly unbalanced, with a much greater width than height.

Figure 1 illustrates the matmul operation flow in the high-res case. Since M and K are relatively small, the entire weight matrix can be stored in BRAM. Input tiles of size K x TILE_N are loaded into BRAM to perform the matmul operation. As the entire weight matrix is preloaded into BRAM, all computations required for a given input tile can be executed once the tile is loaded. After computation, the output tile of size M x TILE_N is immediately written back to DRAM (Figure 1①). The sizes of the weight BRAM and input BRAM are determined based on the maximum values of M and K for efficient utilization across all operations.

This approach ensures that both the weight and input data are loaded into BRAM only once, minimizing DRAM access and improving performance. However, the process has limitations. While the entire weight matrix is loaded initially,

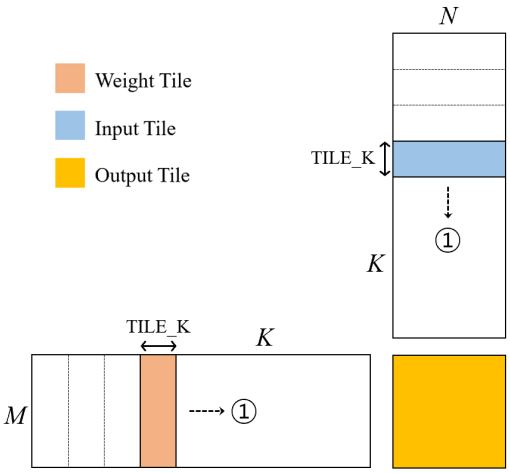

Fig. 2: Matmul flow for the low-resolution case in Version 1. Partial sums of the output matrix are accumulated in BRAM. Tiling is performed along the accumulation dimension K, with weight tiles of size $M \times$ TILE_K and input tiles of size TILE_K $\times N$ loaded from DRAM. Output tiles are stored in BRAM for efficient computation.

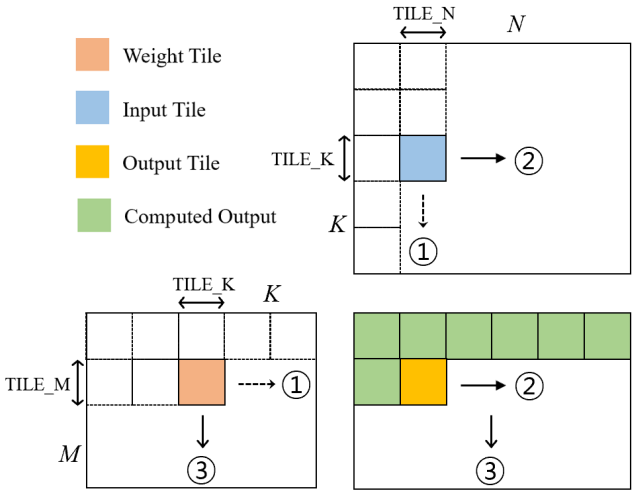

Fig. 3: Enhanced matmul flow in Version 2. Tiling is performed along all dimensions M, K, and N for efficient BRAM utilization. Inner loops iterate over K ①, followed by N ②, and M ③, ensuring optimized computation and parallel loading of weights and inputs to minimize DRAM access latency.

input tiles are loaded sequentially afterward, preventing parallel loading from different bundles. Even with the weights preloaded, the process of loading input tiles can become a computational bottleneck.

B. Version 1: Low-Res Case

The low-res case applies when the spatial dimension, represented by N, is 64 or 256. In this scenario, `in_channels` and `out_channels` are relatively larger compared to the high-res case. According to equation (2), the square of `kernel_size` is multiplied with `in_channels`, resulting in M reaching a maximum of 512 and K up to 4608.

Figure 2 illustrates the matmul operation for the low-res case. In this case, the accumulation dimension K is significantly larger than the output matrix dimensions M and N, making it impossible to store the entire weight or input data in BRAM. Instead, the partial sums of the output are stored in BRAM, and matmul is performed using tiling along the K dimension (Figure 2①).

In this approach, the output tile effectively represents the entire output matrix. Matmul is performed by loading a weight tile of size M x TILE_K and an input tile of size TILE_K x N into BRAM. Similar to the high-res case, both the weight and input data require only a single DRAM access. However, unlike the high-res case, this approach allows parallel loading from two bundles, helping to alleviate the DRAM access bottleneck.

In Version 1, the BRAM size is determined based on the maximum values of M, K, and N for both the high-res and low-res cases. While this approach can accommodate operations with smaller values of M, K, and N, it results in underutilization of BRAM space for most operations. Version 2 addresses this limitation by optimizing the allocation of BRAM to maximize its usage across any combination of M, K, and N inputs.

C. Version 2: Enhanced Design

Version 2 addresses the limitations of Version 1, including the inability to utilize bundles in parallel and the inefficient use of BRAM space. It was designed to operate effectively across all cases. Unlike the previous cases, where tiling was applied only to N and K, Version 2 applies tiling to M, K, and N simultaneously (Figure 3).

The innermost loop processes K, accumulating values for the output tile until the final output is computed (Figure 3①). The next loop iterates over N (Figure 3②), while the outermost loop iterates over M, shifting the output tile to compute the entire output matrix (Figure 3③).

In the previous approach, the BRAM size was determined based on the maximum values of M, K, and N. In Version 2, however, BRAM usage is proportional to TILE_M, TILE_K, and TILE_N, allowing for larger tile sizes and a reduction in the number of iterations. Although Version 2 requires repeatedly storing inputs from DRAM into BRAM, the parallel loading of weights and inputs enables latency hiding, ensuring no additional delay.

Additionally, weights are stored in DRAM in a transposed format. This approach is intended to balance the latency when loading weights and inputs from DRAM to BRAM. The DRAM-to-BRAM latency depends on the size of the data loaded in bursts, which is only feasible for contiguous data in memory [2]. For weights, the burst size is proportional

TABLE II: Execution Time Breakdown for Different Matmul Versions

	Bitstream Reprogram	MMIO	Matmul (high-res)	Matmul (low-res)	Total
Version 1	721ms	0.024ms	1357ms	503ms	2581ms
Version 2	0ms	0ms	**873ms**	**450ms**	1323ms

to TILE_K, while for inputs, it is proportional to TILE_N. If TILE_M, TILE_K, and TILE_N are all equal, there is no issue. However, if they differ, the load balance may become uneven.

For example, consider a case where TILE_M = TILE_N = 128 and TILE_K = 16. If the weights are not stored in a transposed format, the maximum burst length for weights would be 16, while for inputs it would be 128. Despite the tiles having the same size, the weight load would take significantly longer, increasing the overall computation time. However, by storing the weights in a transposed format, both tiles can achieve a maximum burst length of 128, aligning the load balance and ensuring that both loads complete at nearly the same time. In Section IV-B, this hypothesis will be tested, and the impact of TILE_M, TILE_K, and TILE_N sizes on performance will be analyzed.

IV. Experiments and Discussion

A. Version 1 vs. Version 2

Table II compares the matmul execution times between Version 1 and Version 2. In Version 2, a preprogrammed single bitstream is used, eliminating any delay from reprogramming. In contrast, Version 1 requires two bitstream switches during execution, each taking approximately 0.3 to 0.4 seconds, resulting in a total delay of 721 ms. Despite this overhead and the use of different bitstreams for high-res and low-res cases, Version 2 consistently outperformed Version 1.

The performance gap is particularly significant in the high-res case. This is because Version 1's high-res case fails to utilize parallel bundle resources, providing no latency hiding. While the computational workload is the same, the inability to hide DRAM access latency causes the execution to be over 1.5 times slower.

For the low-res case, the gap is smaller, but Version 2 still demonstrates better performance. This can be attributed to Version 1 allocating BRAM based on the maximum output matrix size, which leads to smaller weight and input tiles. Additionally, the lack of transposed weights in Version 1 causes imbalanced DRAM access.

These experiments highlight that minimizing DRAM access and considering load balancing for latency hiding significantly impact performance, underscoring their importance for efficient matmul execution.

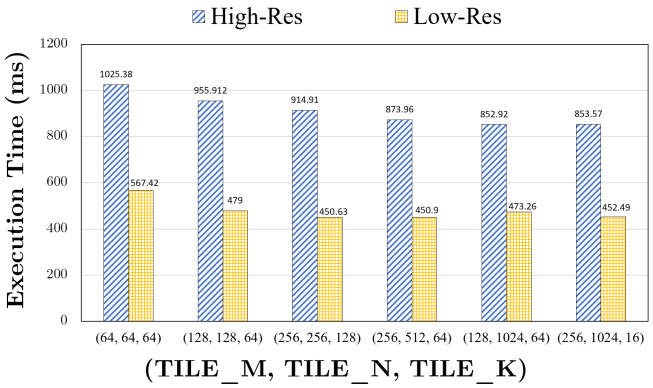

Fig. 4: Execution time for different tile sizes in Version 2. The x-axis represents the tile size, and the y-axis represents the execution time in milliseconds.

B. Tiling Size Analysis

In Version 2, experiments were conducted using various tile sizes to analyze the impact of TILE_M, TILE_K, and TILE_N on performance. Figure 4 illustrates the matmul execution times for different tile sizes. Across all scenarios, the high-res case showed longer execution times compared to the low-res case, and larger tile sizes consistently resulted in reduced execution times.

Although increasing the tile size reduces the number of iterations, the total computation workload remains unchanged, so the execution time should theoretically stay constant. However, the reduction in execution time with larger tile sizes can be attributed to an increase in the max burst length, which lowers DRAM access latency. Consequently, in the high-res case, where the maximum value of N is 4096, performance improves as TILE_N increases.

Unlike TILE_M and TILE_N, the size of TILE_K had little impact on performance. This is because TILE_K does not influence burst length or the total computation workload; it merely determines the granularity of the computations. In other words, setting TILE_K to a smaller value allows TILE_M and TILE_N to increase proportionally, reducing the number of iterations and consequently improving performance.

Based on these observations, the optimal tile sizes were determined to be TILE_M = 256, TILE_N = 1024, and TILE_K = 16. With this configuration, the high-res case took approximately 850 ms, and the low-res case took about 450 ms, reducing the total matmul execution time to around 1.3 seconds.

C. Data Copy Overhead

As previously noted, repeated read/write operations on the same feature map by the PS and PL can result in overhead, regardless of whether shared memory is cacheable. To confirm this, I compared the total execution times of kernels in the `ConvNextBlock` sequence (convolution1-GELU-layernorm-convolution2). In one scenario, the output of

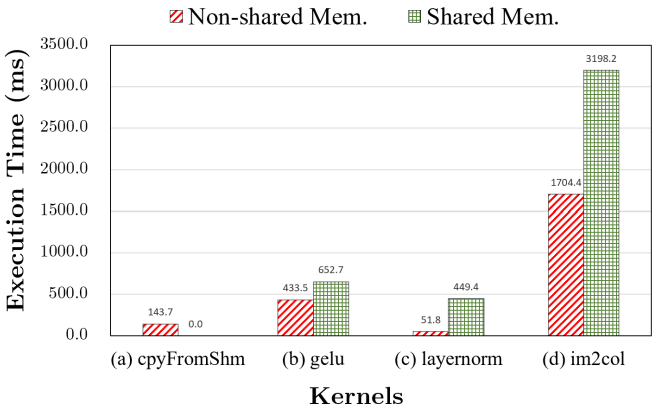

Fig. 5: Comparison of execution time for different memory access patterns. The x-axis represents the memory access pattern, and the y-axis represents the execution time in milliseconds.

convolution1 was stored in shared memory and used directly for the im2col operation in convolution2. In the other, the output was stored in non-shared memory accessible only to the PS before being passed to the im2col operation in convolution2.

The experimental results are shown in Figure 5. When non-shared memory was used, approximately 143 ms were required to load and store the output of convolution1. Directly utilizing shared memory for operations such as GELU, layernorm, and im2col without copying resulted in significant slowdowns: GELU was 1.51x slower, layernorm was 8.67x slower, and im2col was 1.87x slower. These results indicate that shared memory behaves as noncacheable memory, leading to substantial performance degradation.

In conclusion, even when accounting for the overhead of data copying, performing computations in non-shared memory is roughly twice as fast. This underscores the importance of separating memory access for the PS and PL to optimize performance.

D. Execution Time Breakdown

Finally, the execution times of each operation in the final design were measured and analyzed. Matmul accounted for only 30% of the total execution time, while im2col consumed the largest portion at 40%, followed by GELU. (Table III) When the im2col output was allocated in non-shared memory, it took approximately 0.25 seconds. However, storing the output in shared memory prevented the use of locality, leading to a latency increase of about 7x, resulting in a total of 1.68 seconds.

Storing the im2col output in non-shared memory and then copying it to shared memory might seem like a faster approach. While this method does reduce the im2col computation time to 0.25 seconds, the copy operation to shared memory adds approximately 1.7 seconds. As a result, directly storing the output in shared memory remains slightly faster overall.

TABLE III: Final Execution Time Breakdown

Kernel	Time (s)	Percentage (%)
Matmul(high-res)	0.853	20.04
Matmul(low-res)	0.452	10.62
Im2col	1.684	39.54
GroupedConv	0.318	7.46
Gelu	0.432	10.15
Layernorm	0.091	2.13
CpyToShm	0.002	0.04
CpyFromShm	0.370	8.67
Other	0.058	1.35
Total	**4.260**	**100.00**

V. Conclusion

In this Project 3, the final execution time was reduced to 4.26 seconds, surpassing the performance target of 7.5 seconds. The matmul operation accounted for only 30% of the total execution time, demonstrating successful optimization. However, there remain several areas for further improvement.

One of the most significant areas for improvement lies in leveraging computation-communication overlap during the matmul operation. With HLS, it is possible to design a dataflow-based system using streams. As shown in Figure 4, the variations in execution time with different tile sizes were solely due to differences in burst length, while the number of DRAM accesses contributing to communication latency remained unchanged. If data loading and computation were executed simultaneously, performance improvements would surpass the gains achieved by parallel utilization of two bundles, and the performance differences between tile sizes would be even more pronounced.

Additional improvements could involve utilizing the widened bits of the AXI interface or integrating and optimizing the im2col operation, which accounts for 40% of the total execution time. Since this project involves implementing new hardware, there is also the possibility of exploring alternative convolution implementations that deviate from the conventional memory-hierarchy-optimized im2col and matmul approach. For instance, one could consider implementing tile-wise direct convolution or using Winograd convolution for further optimization [3], [4].

Through the three sequential projects, I gained valuable experience in accelerating DNN models across CPU, GPU, and FPGA platforms, deepening my understanding of the unique characteristics of each processor. I believe the insights and skills acquired during these projects will be immensely beneficial for future research endeavors.

Acknowledgment

In the process of implementing ideas into code, I received assistance from OpenAI's ChatGPT. The prompts and re-

sponses used can be viewed through the reference link [5].

REFERENCES

[1] ”Vitis High-Level Synthesis User Guide.” [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Using-Manual-Burst

[2] ”Zynq 7000 SoCs.” [Online]. Available: https://www.amd.com/ko/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

[3] F. Indirli, A. Erdem, C. Silvano, ”A Tile-based Fused-layer CNN Accelerator for FPGAs,” in *ICECS*, 2020.

[4] A. Ahmad, M. A. Pasha, ”FFConv: An FPGA-based Accelerator for Fast Convolution Layers in Convolutional Neural Networks,” in *TECS*, 2020.

[5] OpenAI, ”ChatGPT” [Online]. Available: https://chatgpt.com/share/674c7af3-ccc8-8008-860c-9f75b4c521cd