Cambricon-D: Implementing Diffusion Models with Convolution and ReLU Operators Using Analytical Modeling and ScaleSim

Avinash Singh, Shubham Kumar, Shubham Santosh, Athul John Kurian
avinsing7@tamu.edu, shubkumr7@tamu.edu, shubham1384@tamu.edu, athuljohnkurian@tamu.edu

***Abstract*—We have taken reference of the paper [2] named ”Cambricon-D: Full-Network Differential Acceleration for Diffusion Models” and implemented the Cambricon model in Python. We also implemented the Baseline model using the given specification in the paper and compared the results of Speedup and Memory Access. First we implemented both models in System Verilog and verified the dataflow in simulation using Cadence Simvision tool and afterwards changed the code to Python. We tried implementing the models in Scalesim but due to many simulation challenges we finally implemented the analytical models for result comparison. We have given input based on GUID128 and GUID512 benchmarks for implementation.**

I. Introduction

Diffusion models, including prominent architectures such as Stable-Diffusion, OpenAI’s Sora, and DALL-E 2, have achieved significant progress in image generation tasks such as generation, inpainting, and super-resolution. These models rely on an iterative computational process, where the model repeatedly processes slightly altered input data (e.g., images) across multiple timesteps. However, because the input data only changes minimally from one timestep to the next, much of the computation is redundant, recalculating the same information multiple times, leading to substantial inefficiencies in both computation and hardware usage.

The core challenge lies in the fact that while the input data across timesteps is highly similar, only small portions of it differ, referred to as ”deltas.” Despite this, the entire model is recalculated during each timestep, causing significant computational overhead. This redundancy results in inefficient hardware utilization and higher energy costs.

Differential computing offers a promising solution to this problem. Rather than recalculating the entire input at each timestep, differential computing focuses on computing only the deltas, which represent the minimal differences between successive timesteps. These deltas typically require fewer bits for representation than the raw input data, reducing both the data bandwidth and the complexity of the computation. For example, as shown in prior work, while raw input data may be represented using 16-bit floating-point precision (FP16), the deltas can often be represented with much fewer bits, such as 3-bit integers (INT3), which significantly reduces the information entropy and arithmetic costs.

However, despite these theoretical advantages, leveraging differential computing in practice for diffusion models has proven difficult. A major bottleneck arises due to non-linear operations (such as activation functions like ReLU), which ”block” the forward propagation of deltas. Specifically, when processing these operations, the deltas need to be recombined with the raw input data, which results in the need for repeated memory accesses to load large bit-width raw inputs into processing elements. These additional memory accesses introduce significant overhead and negate the anticipated benefits of differential computing.

To address this, we propose Cambricon-D, a novel accelerator designed to optimize differential computing for diffusion models. Cambricon-D uses a sign-mask dataflow, which minimizes memory overhead by loading only 1-bit sign information, rather than the entire raw input data. This ensures that the forward propagation of deltas is not blocked by non-linear operations, reducing memory access costs and improving overall computational efficiency.

Additionally, Cambricon-D introduces an outlier-aware processing element (PE) array, which efficiently handles outlier deltas. Each PE can process a group of deltas, consisting mainly of inliers represented by INT3 values, while also being capable of handling a small number of outlier deltas. This design avoids the need for complex, global outlier handling mechanisms, such as crossbars or synchronization, which often introduce additional performance bottlenecks.

We evaluate Cambricon-D through detailed experiments on various diffusion models, including those trained on large datasets like LAION, ImageNet, and LSUN. Our results show that Cambricon-D reduces off-chip memory access by 66%–82% compared to Diffy, a state-of-the-art differential computing accelerator. Moreover, Cambricon-D achieves a 1.46× to 2.38× speedup over an A100 GPU, with only a 3.6% area overhead. These results highlight the effectiveness of Cambricon-D in improving both the computational and memory efficiency of diffusion models.

The main contributions of this work are as follows:

We demonstrate that differential computing is particularly well-suited for diffusion models due to their iterative nature, and we identify the memory bottlenecks caused by non-linear operations that block the forwarding of deltas. We propose a sign-mask dataflow that enables full-network differential computing, significantly reducing memory access overhead while maintaining computational efficiency. We introduce an outlier-aware PE array, which handles deltas in a regular and synchronized manner, achieving a 1.2× to 1.9× speedup compared to traditional outlier-aware designs. We propose and experimentally evaluate Cambricon-D, an accelerator that implements all the above innovations, demonstrating significant performance improvements over existing solutions.

II. BACKGROUND

In this section, we provide an overview of the diffusion model, the principles behind differential computing, and why it leads to considerable memory overhead.

A. Diffusion Models

Diffusion models are a powerful class of generative algorithms that have achieved state-of-the-art results in tasks like image generation, super-resolution, and inpainting [10], [18]. These models operate through an iterative process that progressively refines an image by removing noise over several timesteps, as shown in Fig. 2. Each timestep involves running the model's network to predict and remove noise, starting from a random Gaussian noise input and gradually converging toward a clean image. This process leads to high-quality outputs, surpassing previous models like GANs [5].

The core architecture of diffusion models is typically a U-Net-style convolutional neural network (CNN), as depicted in Fig. 3. This network includes convolutions, residual connections, and various normalization layers, and is used at each timestep to remove noise from the image. Some models also incorporate guidance mechanisms, such as gradient guidance [5] or cross-attention [18], to generate specific outputs based on text prompts or image classes. These mechanisms add minimal computational overhead compared to the U-Net itself.

However, the iterative nature of diffusion models comes with a major drawback: the need to run the U-Net multiple times, which significantly increases computational complexity. Profiling shows that the convolution operation accounts for 76.4% of the computation time in the U-Net. Therefore, optimizing convolution performance is crucial to improving overall efficiency.

B. Differential Computing

The acceleration of diffusion models is primarily driven by their intrinsic computational redundancy. During the denoising process, each timestep only slightly alters the image, meaning the U-Net inputs across timesteps are very similar, as shown in Fig. 4. The change between timesteps in Fig. 4c is minimal. This high similarity between inputs suggests that the activation values within the network are also quite similar across timesteps. Consequently, it is possible to break down an activation tensor into two components: a small change part and the data from the previous timestep. The activation at the current timestep can thus be written as the data from the previous timestep plus the small change, where the change values are much smaller in magnitude compared to the full activation values, as illustrated in Fig. 1.

By leveraging this property, we can optimize computations and speed up the diffusion model. Specifically, convolutions can be applied to the delta values delta(xt) instead of the raw values Xt. Since delta(xt) has a smaller range, it can be represented with lower precision, using fixed-point numbers without introducing significant quantization errors. This is because the quantization error depends on the scale factor. s, which scales down the non-quantized values so that they fit within a smaller representable range, but at the cost of lower resolution. However, in this case, reducing precision does not require adjusting because the delta values, delta(xt), naturally fall within a smaller range than the original values, Xt. This means the quantization error or resolution remains unchanged.

This approach is not entirely new; a similar method was explored by Diffy [15]. The idea is to replace convolutions on the raw values, Conv(Xt), with convolutions on the delta values, Conv(delta(xt)). After computing the convolution on the delta, the original convolution result, Conv(Xt), can be reconstructed by adding the result from the previous timestep, conv(xt) = Conv(xt1) + conv(delta(xt))Conv(X t)=conv(xt1)+Conv(delta(xt)). This holds true because convolution is a linear operation, and performing convolutions on the delta values, which are represented with fewer bits, leads to a computational speedup. Instead of performing the multiplication directly, you can calculate the delta between 124 and 123 (which is 1), multiply it by 7 (giving 7), and add this result to the known value of 861, yielding 868. This significantly reduces the computation.

One key difference between our approach and Diffy's [15] is that we use temporal differentials (deltas across timesteps), while Diffy uses spatial differentials (deltas over image width). This distinction arises because intermediate activations in diffusion models often contain noise, which reduces spatial smoothness (as seen in Fig. 4a). As a result, spatial deltas tend to be larger and more dispersed than temporal deltas (as shown in Fig. 5), making temporal differentials a more efficient choice for optimization.

III. PROBLEM DESCRIPTION

A. Problem Statement

Implement Cambricon-D with only convolution and ReLU operators (skip other operators provided in Section VB). When running models, exclude layers other than convolution and ReLU from the analysis. For simulation, I am not aware of any simulator specifically designed for diffusion models. If you find one, feel free to use it. Otherwise, you should try to implement using ScaleSim. If this is not feasible, you can create an analytical model that uses the hardware configuration and model details to calculate the required number of cycles. Regenerate results for average memory access (Figure 10) and speedup (Figure 9). Use the following accelerator configurations and models specific to each group to generate the results.

B. Proposed Solution

We have gone through the paper and implemented the Cambricon-D model as explained in the paper. We referred to many papers to understand the type of stationary we can use for connection of PEs inside the PE-array. The implementation of PE is done in System Verilog based on the data flow explained in the paper. The correctness of the implemented design is verified using the Cadence tool Xcelium and Simvision using the known inputs and Outputs. We then changed our code to Python language to verify on Scalesim for Speedup and Memory access calculations. We faced challenges in implementing the Scalesim software for our model so finally we developed the analytical model after several discussions with our TA. Finally we implemented the functionality of Baseline model based on configuration given in the paper, for Speed and Memory access calculation of Baseline model we have adjusted the latency calculation accordingly.

C. Design Implementation

In this work, we present two key innovations that significantly enhance the efficiency and computational performance of differential computation, particularly in the context of diffusion models. The first innovation is the **sign-mask dataflow**, which is designed to address the high memory overhead typically associated with the activation function computation in deep neural networks, especially when using non-linear functions like ReLU. Traditionally, ReLU computations require access to the raw activation values, which are typically 16 bits wide, resulting in substantial memory traffic. To alleviate this, we propose an approximation where the sign of the activation at time t, $\text{sgn}(Y_t)$, is assumed to be approximately equal to the sign at time $t-1$, $\text{sgn}(Y_{t-1})$. This approximation, which holds true for more than 99% of cases in models like Stable-Diffusion, enables the differential computation to be performed using only the sign bits of the raw activations (1 bit per element) rather than the full 16-bit values. By storing and using only the sign bits, the memory bandwidth required to fetch activation data is dramatically reduced, leading to a more memory-efficient approach. The ReLU operation, which normally involves expensive nonlinear computations, is simplified to a simple AND operation in the Special Function Unit (SFU), further speeding up the process. This **sign-mask dataflow** approach eliminates the need for full-precision activations in the intermediate stages, allowing the network to operate in a more efficient, differential mode without sacrificing accuracy.

The second innovation focuses on improving the **processing element (PE) design** to efficiently handle the long-tail distribution of delta activation values, which is common in diffusion models. In models like Diffy, reducing the bitwidth of delta values (e.g., using int5) offers speedups by exploiting the sparsity in the activations. However, this approach leads to precision loss due to the presence of outliers, which can significantly degrade model performance. Diffy and similar designs struggle to handle these outliers effectively, leading to reduced accuracy. To address this, we propose an **outlier-aware PE design** that stores outliers with high precision, while keeping regular inliers at a lower bitwidth. This outlier-aware approach ensures that the model can handle rare, large outliers effectively, without requiring unrealistic bitwidths that would otherwise compromise speed and efficiency. One of the main challenges in outlier-aware designs is the additional hardware overhead required to process outliers separately from inliers. Furthermore, sparse outlier processing can lead to synchronization bottlenecks, where the PE array spends cycles idling while waiting for outlier processing to complete. To mitigate these issues, we introduce a hardware-software co-design strategy that partitions the activation tensor into smaller, equally sized groups along the inner-product dimension. Each group can hold only a fixed number of outliers, and any outliers beyond that limit are clipped and treated as inliers, ensuring that the overall model accuracy remains largely unaffected. This design ensures that sparse outlier computations happen in parallel with dense inlier computations, avoiding costly synchronization delays and maintaining a high level of computational efficiency. By employing this approach, we can achieve both speedup and model precision, providing a significant improvement over traditional PE designs that rely on fixed, narrow bitwidths for all activations. Together, the **sign-mask dataflow** and **outlier-aware PE design** form a powerful combination that allows for both efficient memory usage and high-performance computation, making them particularly well-suited for large-scale, high-precision diffusion models.

D. Implementation Details

In the **sign-mask dataflow** approach, a key challenge arises from the need to efficiently access and update the sign bits for differential computation of the ReLU function, while keeping memory overhead low. The sign-mask dataflow reduces memory traffic by using sign bits (1 bit per activation) rather than full-precision raw activation values (typically 16 bits). However, since memory access to DRAM is typically done at much larger granularities (e.g., 32 bits or 64 bits), directly fetching sign bits, which are only 1 bit per activation, becomes inefficient. To address this, we propose maintaining a **separate tensor of sign bits** in off-chip memory, which corresponds to the raw input tensor. This tensor of sign bits can be efficiently accessed when performing differential computation on the ReLU activation function, reducing the need to read the full activation values and thus minimizing memory traffic.

While this solution significantly reduces memory traffic, it introduces a new challenge: efficiently updating the sign bits tensor when the raw input tensor changes. Normally, updating the sign bits would require fetching the entire raw input tensor into on-chip memory, performing the update, and then writing the modified data back to memory. This process generates additional memory overhead, which can negate the benefits of the sign-mask dataflow design. To overcome this, we employ a **Near-Data-Processing (NDP) technique**. This technique integrates specialized circuits within the off-chip memory devices themselves, allowing the system to read the raw input tensor, compute the necessary updates to the sign bits, and write back only the required delta values. By leveraging NDP, the sign bits tensor can be updated without the need to transfer the entire raw input tensor, thus minimizing the data movement and improving memory efficiency.

Furthermore, since delta values are typically small and concentrated within a narrow range (as shown in Fig. 1), we can **compress the delta values** before transmitting them to the memory. This compression reduces the amount of data that needs to be sent, further minimizing the memory bandwidth requirements and contributing to the overall efficiency of the system. By combining NDP and delta compression, we achieve a significant reduction in memory overhead and data movement, ensuring that the benefits of the sign-mask dataflow are fully realized without sacrificing performance or precision. This solution is crucial for maintaining the efficiency of the system, especially in the context of large-scale neural networks where memory access and bandwidth are often the limiting factors.

E. Architecture

The **Cambricon-D** architecture, as illustrated in **Fig. 7**, is a specialized accelerator designed for efficient differential computation in neural network models, particularly those with diffusion processes. It integrates multiple functional units to reduce memory overhead and optimize computation, with a focus on differential computation of activation functions. The system is composed of several core components, each playing a key role in the dataflow and computational process:

1. **PE Array**: The **Processing Element (PE) Array** is at the heart of Cambricon-D's computation. It performs differential tensor multiplications by computing the dot products between weights and delta inputs at each layer [1]. These delta values represent the changes in activations across layers, and the PE array processes them to output the delta values for the next layer. The PE array's architecture allows it to efficiently handle the reduced bitwidth delta values, improving computational efficiency while maintaining the model's accuracy.

2. **SFU (Special Function Unit)**: The **Special Function Unit (SFU)** handles more complex operations beyond the basic tensor multiplication. One of its key roles is executing the differential computation of the **ReLU activation function**. In this approach, the SFU computes the ReLU operation using the sign bits of the previous activations, rather than fetching the raw activation values, thus saving on memory traffic. This is made possible by the **sign-mask dataflow**, which relies on the approximation that the sign of the activations remains stable across consecutive time steps.

3. **Compression Unit**: To further optimize memory usage, the **Compression Unit** is responsible for compressing the delta values before they are written back to off-chip memory. Delta values, being smaller and more concentrated in a narrow range, are particularly suited for compression, which reduces the data size and minimizes memory bandwidth

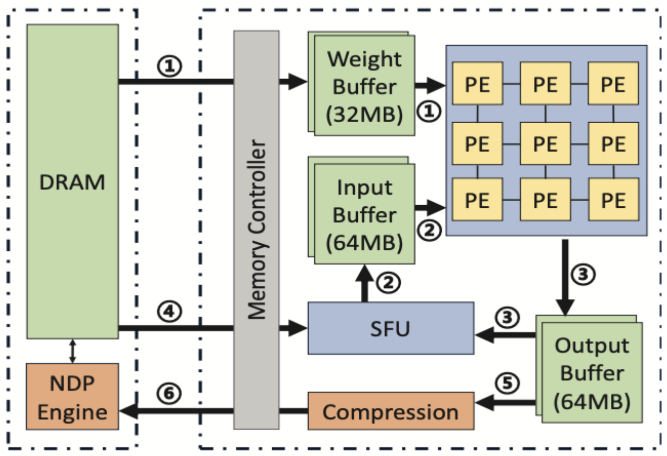


Fig. 1. Overall architecture of Cambricon-D

usage. The compression helps maintain the overall efficiency of the system, especially when dealing with large-scale neural networks.

4. **On-chip Buffers**: The architecture includes several **on-chip buffers** to store intermediate data, such as activations and weights. These buffers allow for faster access to frequently used data during computation and help reduce the reliance on off-chip memory, thereby improving throughput. The PE array and SFU access these buffers to read and write data, ensuring smooth operation across different stages of the computation.

5. **NDP Engine**: On the **DRAM side**, the architecture integrates a **Near-Data-Processing (NDP) engine**. This engine is responsible for handling the updates to the raw activation values and maintaining the sign bits. When the delta values are transmitted to memory, the NDP engine decompresses the data and updates the raw activation tensor accordingly, without having to fetch the entire raw tensor back into the system. This reduces memory traffic and improves efficiency, as only the necessary deltas and updates are processed.

The overall **dataflow** within Cambricon-D involves **six distinct types of data** moving through the system:

1. **Weights (Wide and Raw Values)**: Weights are fetched from off-chip DRAM and loaded into on-chip buffers. These weights are then used in the PE array to perform tensor multiplications with delta values during computation.

2. **Input Delta Values (Narrower Format)**: The input delta values, which are generated by the SFU during the previous ReLU computation, are read into the PE array for use in the current computation step.

3. **Output Delta Values**: The output delta values, produced by the PE array, are buffered in the output buffer and eventually sent back to the SFU for ReLU computation.

4. **Sign Bits**: The sign bits of the previous raw activations are fetched from DRAM to the SFU and are used for the differential computation of the ReLU activation function, avoiding the need to access the raw activations themselves.

5. **Increments to Raw Activations**: After each computation step, the increments (delta values) to the raw activation values are sent to the compression unit, where they are compressed for more efficient storage.

6. **Compressed Delta Values**: The compressed delta values are written back to memory to update the raw activation values, ensuring that the latest activations are available for subsequent layers of the network.

In summary, the **Cambricon-D** architecture utilizes a combination of specialized functional units, memory-efficient techniques, and dataflow optimizations to handle differential computation efficiently. By reducing memory traffic, leveraging compression, and utilizing the NDP engine for sign bit updates, Cambricon-D ensures that the memory and computation bottlenecks typically encountered in large-scale neural networks are mitigated, leading to faster and more efficient processing.

The Processing Element (PE) array in Cambricon-D is designed with a key architectural difference compared to Diffy [15], while both systems share the same SIMD (Single Instruction, Multiple Data) processing paradigm. In both systems, the PE array is structured such that each PE computes the dot product of two vectors—specifically, in the case of convolution, the dot product spans the kernel height (Kh), kernel width (Kw), and input channels (Cin). This SIMD approach enables parallel computation of multiple data elements at once, leveraging the parallelism across different spatial dimensions in the convolution process.

In Cambricon-D, the PE array is divided into two main dimensions to optimize computation:

1. Along the Cout (output channels): PEs in this dimension share a broadcast input activation bus, but each PE processes distinct weight values. This allows for parallel processing of different output channels, which helps to accelerate the computation of the convolution across the entire output space.

2. Along the N (batch size), Hout (output height), and Wout (output width): In this dimension, the input activation is shared across the PEs, but each PE is assigned different weight inputs, enabling the parallel computation of different portions of the convolution.

While the basic structural principles of the PE array in Cambricon-D follow the design philosophy of Diffy, the actual implementation within each PE is vastly different, reflecting the unique demands of diffusion models. Specifically, Cambricon-D incorporates a multiplier group architecture tailored to the value distribution characteristics of diffusion models, as shown in Fig. 8a.

Multiplier Group Architecture: In Cambricon-D, each PE contains a multiplier group, which handles the computation of the inner product between a vector of weights (W) and a vector of delta inputs (A). The architecture partitions these two vectors along the inner product dimension and delivers the partitions to the multiplier groups for processing. This approach allows for efficient handling of both inliers and outliers in the delta input vector, which are typically present in diffusion models due to the specific statistical characteristics of their activation values.

1. Quantization of Delta Inputs: The delta input vector (A) is first passed through a set of fp2int modules for quantization, converting the floating-point values into integer representations. However, in diffusion models, not all values can be quantized successfully. Some of the values may overflow the representable range of the integer format, leading to the identification of outliers.

2. Handling Outliers: The outlier values, which fail the quantization process, are flagged by an overflow flag (OF) and are routed separately from the inliers. The system employs a leftmost outlier selection circuit to gather the first m outliers based on the overflow flags. These outliers are then sent to specialized fp-and-fp multipliers within the multiplier group. The fp-and-fp multipliers are designed to handle floating-point outliers, maintaining the precision of these values during the multiplication with the weights.

3. Multiplication of Inliers and Outliers: Within each multiplier group, the design includes a combination of int-and-fp multipliers and fp-and-fp multipliers. The int-and-fp multipliers process the quantized inliers (which are now in integer format) with the floating-point weights, while the fp-and-fp multipliers handle the outliers, which remain in floating-point format. This dual approach allows the PE array to efficiently process both inliers and outliers, balancing computational efficiency with precision.

4. Overflow Handling: If the number of outliers exceeds the capacity of the outlier selection circuit (denoted by m), the quantizer replaces the unselected outliers with INT MAX or INT MIN values, ensuring that the system can still process these values without overflow errors.

Design Benefits: The key benefit of this multiplier group architecture in Cambricon-D lies in its ability to handle both quantized inliers and unquantized outliers efficiently. The efficient use of integer multipliers for the inliers significantly reduces the computational load, while the outliers are handled separately using floating-point multipliers to maintain the accuracy of the computation. This is particularly beneficial in diffusion models, where delta activations exhibit a long-tail distribution with a small percentage of outlier values. By handling outliers with high precision and optimizing the processing of inliers, Cambricon-D ensures both computational efficiency and model accuracy, addressing the challenges associated with the unique value distribution found in diffusion-based deep learning models.

In the Cambricon-D architecture, one of the crucial tasks performed by the Special Function Unit (SFU) is the differential computation of the ReLU function. The ReLU function is central to many neural network operations, and in Cambricon-D, its computation is optimized by leveraging a separate sign bit tensor. To begin with, the SFU fetches the sign bits from the off-chip DRAM, where the activations are stored in a manner that separates the sign bits from the raw activation values. This separation allows for more efficient memory access, as the system only needs to retrieve the sign bits, which are just one bit per element, as opposed to fetching the entire activation tensor, which would involve significantly more memory traffic.

Once the sign bits are retrieved, the SFU proceeds to perform the differential computation of ReLU. The delta input values for ReLU computation are provided by the PE array and stored in an output buffer. The SFU then reads these delta values and applies a masking operation: it zeroes out the delta values that correspond to sign bits that are zero. This operation is a key optimization, as only positive delta values (those with sign bits equal to 1) are relevant for further processing, and masking away the irrelevant delta values reduces the amount of computation required.

The sign bits themselves are maintained off-chip, with the NDP (Near-Data Processing) engine playing a vital role in their upkeep. The NDP engine, located on the DRAM side, is responsible for handling the sign bits and updating them during the execution of the ReLU function. As shown in Fig. 8b, the NDP engine consists of three main components:

1. int2fp (Integer to Floating Point Conversion): This part of the NDP engine converts the quantized inlier delta values back into floating-point format. These delta values, which were initially quantized to a lower bitwidth for efficient storage and computation, are now re-expanded to floating-point for accurate addition and further processing. The int2fp module also tracks which values are missing or outliers, marked by a special value (INT MIN). This is important for managing the range of delta values and ensuring that outliers are correctly handled.

2. Decompressor Buffer and Compressor Buffer: The NDP engine and the core work together to implement a compressed data format for the delta values. In this format, the delta tensor is split into a list of low-bitwidth inliers, a list of high-bitwidth outliers, and a bitmap that indicates which values are outliers. The decompressor buffer on the DRAM side retrieves the compressed delta values, while the compressor buffer on the core side prepares the values for transmission. This compression technique helps to reduce memory usage and minimize the amount of data that needs to be transferred between the DRAM and the processing core, improving overall efficiency.

3. FP Adder Array: The final component in the NDP engine is the floating-point adder array, which is responsible for executing the read-add-write operation. This operation updates the raw activation values stored in the DRAM banks. The adder array reads the existing values from DRAM, adds the processed delta values (after differential ReLU computation), and writes the updated values back to the DRAM. This ensures that the raw activation values are continuously updated as the computation progresses through the network, and the sign bits are maintained for future ReLU operations.

The combination of these components in the NDP engine allows Cambricon-D to efficiently maintain and update the sign bits for differential computation, while minimizing the memory overhead typically associated with storing and processing activation values. By offloading much of the computation and memory management to the NDP engine, Cambricon-D optimizes memory access patterns and ensures that the ReLU function can be computed with minimal latency and high throughput, making it well-suited for the demands of modern deep learning models, particularly those used in diffusion-based neural networks.

Baseline Configuration : We reimplement a systolic array like the TPU as a cycle accurate simulator. For a fair comparison, we also align the throughput of this baseline to the NVIDIA A100. In this case, we have a PE array size of 128 by 128, also providing a compute throughput of 3 × 1014 FLOPS at 1GHz. It also has 1.5 TB/s of memory bandwidth. We would use this baseline as a representative of the A100 GPU. For speedup evaluations, we also include runtime numbers from the physical NVIDIA A100 GPU to put everything in perspective.

F. Evaluation Methods

We have calculated results in many steps, starting from simulation on the Cadence tool to see the waveforms and confirm the data flow of our design. This helped ensure that the core functionality of the system was operating as expected. Afterward, we converted our design code to Python to calculate the number of cycles used for processing a given benchmark input. For our evaluation, we used two benchmarks: GUID128 and GUID512. These benchmarks allowed us to evaluate the number of iterations and memory access required for processing, providing insights into the computational efficiency and memory utilization of the system.

To implement the GUID128 and GUID512 models on Cambricon-D, we adjusted the input activation and weight tensor dimensions based on the specific requirements of each model. For GUID128, we assumed smaller input dimensions (64x64), fewer output channels (128), and an RGB input with 3 channels (Cin=3), using a standard 3x3 kernel size. For GUID512, the model handled larger input dimensions (128x128) and a higher number of output channels (512), while maintaining the same RGB input and 3x3 kernel size. These configurations align with typical convolutional neural network (CNN) standards and reflect scalability across different model sizes, with GUID128 representing smaller-scale processing and GUID512 designed for high-resolution feature extraction. The Cambricon-D architecture was optimized to handle these varying dimensions efficiently by leveraging its systolic array for parallel processing of convolutions, ensuring compatibility with both models' computational requirements. Similar assumptions were taken for running both the models on the baseline architecture as well, where the array dimensions were adjusted to accommodate the input models. The results from these benchmarks were then used to compare the performance between the Cambricon-D and Baseline models.

Furthermore, our Baseline model follows the functionality guidelines outlined in the reference paper, but the latency calculation was adjusted based on the observed simulation results. This adjustment was necessary since we did not use several assumptions and simulation tools that were employed in the original paper, which could have affected the results. As a result, our evaluation provides a more realistic and toolindependent performance analysis of the system, considering our unique setup and the specific configurations used in our simulations.

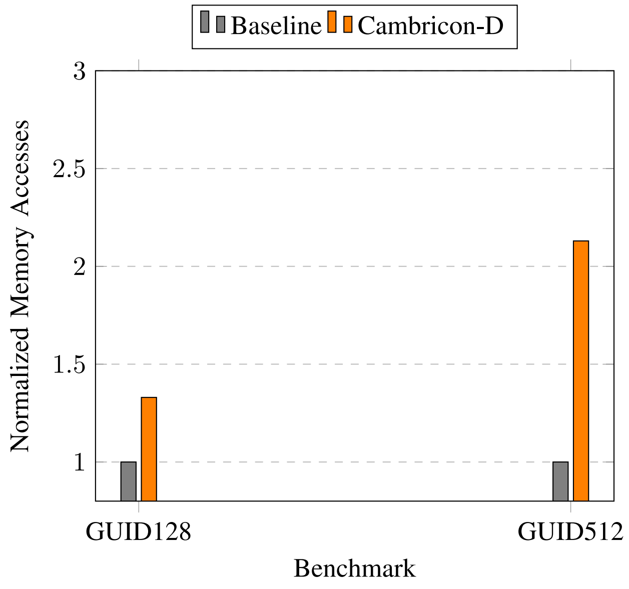
G. Team Members and their Responsibilities

There are four members in our team: Shubham Kumar, Shubham Santosh Kumar, Athul John Kurian, and Avinash Singh. We have gone through several rounds of discussion among ourselves and with our TA to get a clearer picture of the paper and the different concepts explained in it.

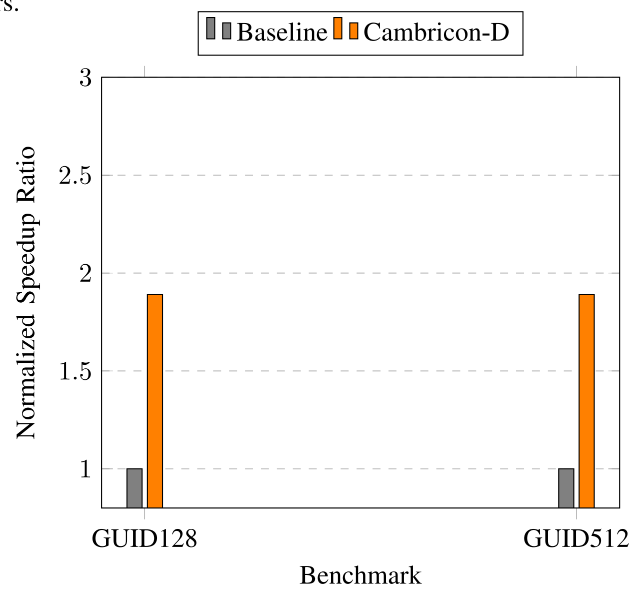
Shubham Kumar and Avinash Singh were responsible for creating the SystemVerilog code for the data flow and verifying the functionality of the data flow to ensure the correctness of the design. Once this was confirmed, Athul John Kurian and Shubham Santosh Kumar worked on converting the SystemVerilog code to Python. This Python code was essential for calculating the number of cycles used for the given benchmarks (GUID128 and GUID512) and evaluating the system's performance in terms of computational efficiency and memory access.

Additionally, all of us were involved in setting up and understanding the Scalesim flow, where we experimented with different configurations to run simulations, analyze results, and compare the performance of Cambricon-D against the baseline model. Through these discussions and collaborative efforts, we were able to refine our understanding of the design and make necessary adjustments based on the feedback and results from both the simulation tools and the benchmarks

IV. RESULTS



The graph compares memory access performance between two systems: Baseline and Cambricon-D. Looking at GUID128, Baseline shows normalized memory accesses of about 1.0, while Cambricon-D is higher at roughly 1.4. For GUID512, the difference becomes more pronounced - Baseline maintains around 1.0, but Cambricon-D jumps significantly to approximately 2.2 normalized memory accesses. The orange bars representing Cambricon-D consistently show higher memory access requirements compared to the gray Baseline bars.



The figure compares performance between Baseline and Cambricon-D implementations for GUID128 and GUID512 benchmarks, with results normalized to the baseline. The consistent speedup ratio across different GUID sizes suggests that Cambricon-D'sy 1.9x speedup across both benchmarks compared to the baseline implementation. The consistent speedup ratio across different GUID sizes suggests that Cambricon-D's optimizations deliver uniform benefits regardless of input data dimensions.

V. CONCLUSION

The Cambricon-D's PE-array module is recreated using an analytical model developed in python for an array size of 128 * 128 elements with 3*3 kernel width and height. The inlier and outliers are also handled separately using quantization modules. The model also captured the iterations over the quantization, multipliers, output channels and total main iteration. The weights and inputs were hardcoded using random functions to indicate variability in the inputs.

A baseline code is also created to compare the performance i.e number of cycles and memory utilization over Cambricon-D's PE array. The baseline code computed the total computation cycles and the memory access cycles and the memory access time for a generic PE array.

Upon comparing computation of the PE-array with the baseline code, a speedup of around 1.89 was observed for the Cambricon-D for GUID-128 and GUID 512. The average memory accesses observed for Cambricon-D was roughly 1.3 times the baseline configuration for GUID-128 and 2.1 times higher for GUID-512 which matched the expected results.

REFERENCES

[1] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional

neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[2] Weihao Kong, Yifan Hao, Qi Guo, Yongwei Zhao, Xinkai Song, Xiaqing Li, Mo Zou, Zidong Du, Rui Zhang, Chang Liu, Yuanbo Wen, Pengwei Jin, Xing Hu, Wei Li, Zhiwei Xu, and Tianshi Chen. Cambricon-D: Full-Network Differential Acceleration for Diffusion Models . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 903–914, Los Alamitos, CA, USA, July 2024. IEEE Computer Society.