CSCE614-Term Project Group 7

Emily Chang
dept. Electrical & Computer Engr.
Texas A&M University

Yu-Ting Lin
dept. Electrical & Computer Engr.
Texas A&M University

Chi-Wei Ho
dept. Electrical & Computer Engr.
Texas A&M University

***Abstract*—Cambricon-D Dataflow combined with Diffy-PE (DiffyPE) introduces a novel approach to optimizing hardware acceleration for diffusion models, significantly reducing computational redundancy through temporal differential computation [5]. Inspired by this innovative architecture, this study leverages SCALE-Sim, https://scalesim-project.github.io/, a CNN accelerator simulator, to explore how varying array configurations impact critical hardware performance metrics [7] [6]. Specifically, we investigate the emergence of memory bottlenecks, the influence of array size requirements on overall performance, and the challenges posed by uneven resource allocation across hardware components. By simulating different configurations for key diffusion model benchmarks (GUID128, GUID256, GUID512, and STBL512), this research aims to provide actionable insights into the scalability and practicality of DiffyPE-inspired designs in modern hardware accelerators [5].**

In this topic, we implemented Cambricon-D with Scale-sim to optimize diffusion models (GUID128, GUID256, GUID512) working on NVIDIA A100 hardware accelerator [2]. We analyze various metrics of diffusion models, especially focusing on average system access bandwidth and performance speedup.

I. Introduction

The growing demand for high-performance hardware accelerators for diffusion models, fueled by applications in image generation and noise reduction, has highlighted the need for more efficient architectural designs. Traditional architectures often suffer from computational redundancy, particularly in iterative processes, where successive computations exhibit only minimal differences. Cambricon-D tackles this inefficiency through its differential computing approach, with DiffyPE serving as a pivotal configuration that integrates temporal differentials with a streamlined processing element design [5].

DiffyPE's architecture is particularly compelling due to its ability to strike a delicate balance between computational efficiency and hardware resource utilization. Its emphasis on processing deltas instead of raw data introduces new opportunities and challenges in hardware optimization, particularly in terms of memory management and array size configurations. Motivated by these innovations, this study employs SCALE-Sim [7] [6] to simulate and analyze the performance of DiffyPE-inspired architectures across a range of scales.

In addition to DiffyPE, other configurations of Cambricon-D include Diffy-Dataflow combined wutg Cambricon-D-PE (DiffyDF), as well as Cambricon-D-Dataflow paired with AsyncOutlier-PE (AsyncPE) [5].

- DiffyDF utilizes a per-operator dataflow similar to Diffy, focusing on computing convolution operators within the context of narrower delta values. However, it reverts back to broader raw values outside of the operator, employing the same PE design as Cambricon-D. This design serves as a benchmark for evaluating the performance gains attributable solely to the dataflow architecture.
- AsyncPE introduces an outlier-aware approach by employing a separate sparse PE array to handle outliers. This array operates asynchronously with the inlier array, synchronizing only after computing an entire layer. By separating inlier and outlier processing, AsyncPE seeks to optimize the handling of edge cases while maintaining overall efficiency [5].

This research focuses on three critical aspects:

1) Memory Bottlenecks: Investigate how different configurations influence memory traffic and bottlenecks, which can limit overall performance.
2) Impact of Array Size Requirements: Exploring the relationship between array dimensions and computational efficiency, and how scaling affects latency and throughput.
3) Uneven Resource Allocation: Analyzing the distribution of hardware resources across processing elements and memory components, identifying inefficiencies, and potential optimizations.

Using benchmark models such as GUID128, GUID256, and GUID512, this work provides a detailed evaluation of DiffyPE-inspired designs. These models are different variations of the guided diffusion model, a gradient-based conditioned DDPM with 0.4 to 0.5 billion parameters. Evaluations were performed on the LSUN dataset (with bedroom and cat images, both at 256x256 resolution) [8] and the ImageNet dataset (at resolutions of 128x128 and 512x512) [3]. The models are referred to as GUID128, GUID256, and GUID512 based on their resolutions throughout the tables and figures. The findings of this study help bridge the gap between simulation and real-world implementation, providing valuable insights for developing scalable and efficient hardware accelerators for diffusion models. In this project, we focus exclusively on evaluating GUID128, GUID256, and GUID512.

II. Problem description

Diffusion models are widely utilized in image generation tasks due to their ability to generate high-quality outputs such as super-resolution images and painting. However, these models suffer from significant computational redundancy, which hampers their efficiency. This issue arises because diffusion models perform iterative computations on slightly altered input data at each timestep, recalculating the entire model at each step.

While differential computing offer a potential solution to this problem by reducing this redundancy with input variations (deltas) [5] rather than recalculating the entire data set, it introduces substantial memory overhead challenges, particularly due to non-linear operations such as activation functions. These operations necessitate frequent access to large raw input data, which can fragment computation process and lead to increased memory traffic, further exacerbating inefficiencies.

Given these challenges, our primary goal is to address the inefficiencies inherent in diffusion models by proposing an improved differential-based hardware accelerator—Cambricon-D. This design aims to reduce computational redundancy and memory overhead by utilizing a more efficient processing architecture. By focusing on the incremental changes (deltas) and optimizing memory access patterns, Cambricon-D seeks to enhance both the speed and efficiency of diffusion models, ultimately enabling faster and more scalable performance in real-world applications.

III. Proposed solution in the paper

The paper introduces Cambricon-D, an efficient hardware accelerator designed to address computational redundancy in diffusion models. The most critical idea in Cambricon-D focus on differential values, or ”deltas” [5] between iterations rather than recalculating the entire dataset, it is possible to reduce both the computational and memory access costs, aiming to increase processing efficiency while maintaining the model's performance accuracy. The remain key innovations of Cambricon-D include:

A. Sign-Mask Dataflow

This novel approach allows full-network differential computing while mitigating the memory traffic caused by non-linear activation functions. Instead of fetching large raw input tensors, only 1-bit sign values are loaded to perform activation functions like ReLU. This significantly reduces memory overhead. The key insight is that we can read only the sign bits of the tensor from the off-chip memory, instead of the entire raw activating tensor [5].

B. Outlier-Aware PE Array

Cambricon-D incorporates an optimized PE (Processing Element) array design that handles deltas efficiently, accounting for rare outliers. It confines synchronization issues and ensures precision by partitioning activations into groups with constrained outlier counts. By increasing the bitwidth to incorporate all of the outliers is inefficient. The methodology provided is to incur a significant accuracy penalty because the diffusion model performance is sensitive to these outliers [5].

C. Hardware-Software Co-Design

By integrating near-data processing (NDP) techniques, the system efficiently updates raw input tensors and maintains sign bits, further optimizing memory usage and performance. We only need to transmit the delta values to the off-chip memory interface, and the NDP engine would handle the updates [5].

In this implementation, we focus on the sign bit data flow and the NDP engine, while ignoring the remaining details for short-term representation. First, we apply three convolutional layers followed by ReLU activation. Second, the sign bit management reduces memory traffic by maintaining a separate tensor that stores the sign of each activation value. Each layer executes the function outlined in Algorithm 1.

```
Algorithm 1: Cambricon-D Algorithm Process Layer
Input: Input: $x$: Input Tensor, $conv\_layer$:
       Convolution Layer, $layer\_name$: Name of the
       Layer
Output: Output: $x$: Processed Tensor, $sign\_bits$: Sign
        Bit Tensor, $pe\_info$: Processing Element
        Information
$x \leftarrow conv\_layer(x)$ ;
$sign\_bits \leftarrow \text{clamp}(\text{sign}(x), \text{min} = 0)$ ;
$sign\_bits \leftarrow \text{byte}(sign\_bits)$ ;
$\delta \leftarrow x - \text{round}(x)$ ;
$outlier\_mask \leftarrow |\delta| > 0.1$ ;          // Identify
  outliers based on threshold
$outlier\_count \leftarrow \sum \text{outlier\_mask}$ ;
$pe\_info \leftarrow \{\text{bitwidth} : 8, \text{outlier\_count} :$
  $outlier\_count\}$ ;
$x \leftarrow \text{ReLU}(x)$ ;
return $x, sign\_bits, pe\_info$ ;
```

The sign bit maintenance extracts the sign bit of each element in the tensor using $torch.sign(x)$. The $torch.sign(x)$ returns -1 for negative values, 0 for zero, and 1 for positive values. Besides, $clamp(min = 0)$ ensures that the sign bit is either 0 or 1. The $byte()$ converts the result to a byte tensor.

We wrote the algorithm in $scalesim/cambricon\_d.py$. However, we encountered some fatal execution errors and are currently trying to resolve them.

IV. Evaluation methods

To assess the performance of Cambricon-D, the following evaluation methods and tools were applied:

A. Simulator

The SCALE-Sim simulator was employed for systolic array-based accelerators to simulate various convolutional neural network layers [7] [6]. This tool provided detailed insights into performance metrics such as compute cycles, utilization, mapping efficiency, and memory bandwidth. However, the simulator is specifically designed for convolution neural network (CNN) model structure. For diffusion model like GUID128, GUID 256 and GUID512, we manually generated the topology and the details is displayed in subsection E.

B. Configurations

The configuration file is used to specify the architecture and run parameters for the simulations [7] [6]. We use NVIDIA A100 GPU as the hardware accelerator baseline. In this evaluations, A100 has a PE array size of 128 by 128 at 1GHz. It also has 1.5 TB/s of memory bandwidth. We would use this baseline as a representative of the A100 GPU [5].

- First, we have to set MAC systolic array to number of rows and columns. The descriptions are following by
  - **ArrayHeight**: Number of rows of the MAC systolic array
  - **ArrayWidth**: Number of columns of the MAC systolic array

  Three hardware configurations (array sizes) were applied to evaluate:
  - 128x128
  - 256x256
  - 512x512

  For this project, the default array size was set to 128x128.
- IfmapSramSzkB, FilterSramSzkB and OfmapSramSzkB depend on the memory architecture and total available SRAM size. These values must be derived from design constraints or assumed if not explicitly provided [6]. The A100 configuration was tested using a fixed architecture with the following SRAM sizes:
  - IfmapSRAMSize: 40960 kB
  - FilterSRAMSize: 40960 kB
  - OfmapSRAMSize: 20480 kB
- IfmapOffset, FilterOffset and OfmapOffset are offsets used to generate addresses for each mapped SRAM. [6]:
  - IfmapOffset: 0
  - FilterOffset: 10000000
  - OfmapOffset: 20000000
- The memory bandwidth set 1.5TB/s is according to NVIDIA A100 [2] [5].
- Dataflow: Output Stationary was utilized to ensure consistency across all simulations.
- Memory Bank: Our estimation is based on typical memory architectures of accelerators (e.g., 32 banks for high bandwidth architectures [1]). For simplification, we set it to 4.

C. Metrics Evaluated

There are some metrics we can evaluated:

- Compute Cycles: Total cycles required to complete computations for each layer.
- Hardware Utilization: The percentage of available hardware actively used during computation.
- Mapping Efficiency: How effectively computational tasks were mapped onto the hardware resources.
- Memory Bandwidth: Monitored DRAM bandwidth for IFMAP, Filter, and OFMAP accesses in terms of words per cycle.

Average memory access and speedup are among the most intuitive metrics for performance analysis. In particular, the average memory bandwidth measures measures the amount of data accessed per cycle on average. To calculate the average memory access, one simply sum up the average bandwidth of the input feature map (IFMAP), filter, and output feature map (OFMAP). Therefore, the formula is given by:

$$\begin{aligned}\text{Average Memory Access} = &\ \text{Average\_IFMAP\_BW}\\ &+ \text{Average\_Filter\_BW}\\ &+ \text{Average\_OFMAP\_DRAM\_BW}\end{aligned}$$

On the other hands, the speedup is typically calculated as a ratio of the time taken by the baseline implementation to the time taken by the optimized implementation. To compute speedup, a baseline reference (e.g., a non-optimized version or a theoretical worst-case scenario) is necessary. If we assume the baseline involves more stall cycles or a lower utilization rate, we can estimate the speedup as follows:

- Baseline Compute Time: Suppose the baseline implementation had Stall Cycles that greater than zero, leading to more total cycles than the optimized one.
- Optimized Compute Time: Given that Stall Cycles are equal to zero, the optimized compute time is simply the Compute Cycles.

The speedup can be calculated using compute cycles of the baseline ($C_{baseline}$) and the optimized ($C_{optimized}$):

$$Speedup = \frac{Cycles_{baseline}}{Cycles_{optimized}} \tag{1}$$

D. Topologies

The topology file contains the feature dimensions for each layer in the given neural network workload. This is a CSV file, with each row listing all the required hyper-parameters for a given layer. The layers are typically described in terms of convolutional layer parameters for networks such as AlexNet, ResNets, UNet-2D, and others. The topology file represents several key features for each layer, including the following: [7] [6]

- **Layer Name**: User defined tag
- **IFMAP Height**: Dimension of IFMAP matrix
- **IFMAP Width**: Dimension of IFMAP matrix
- **Filter Height**: Dimension of one Filter matrix
- **Filter Width**: Dimension of one Filter matrix
- **Channels**: Number of Input channels
- **Num Filter**: Number of Filter matrices. This is also the number of OFMAP channels
- **Strides**: Strides in convolution

However, scale-sim is specifically designed for CNN models and DNN models. For diffusion model, we encountered a lack of sufficient topology resources. The only relevant resource we found was the guided-diffusion repository, provided by https://github.com/openai/guided-diffusion.

A diffusion model is a type of generative model used in machine learning that generates samples by progressively reversing a noisy process. It begins with pure noise and iteratively refines the noise into a coherent image or other output. At each step, the noise level is reduced while preserving the underlying structure of the target output. [3] We accessed the guided-diffusion GitHub repository and retrived the Pytorch files: $128x128\_diffusion.pt$, $256x256\_diffusion.pt$ and $512x512\_diffusion.pt$

Nevertheless, we were only able to transfer $128x128\_diffusion.pt$ to the topology. For $256x256\_diffusion.pt$ and $512x512\_diffusion.pt$, both models were too large. It causes Python program fail to afford it with errors. Therefore, we referred to the structure of Denoising Diffusion Probabilistic Models (DDPM) to build the topologies [4].

Due to technical issues, such as the inability to execute the topologies generated with $128x128\_diffusion.pt$ in scale-sim, we ultimately substituted the topology file with $alexnet.csv$.

E. Comparative Analysis

Comparisons might be drawn between the three configurations in :

- 128x128: This configuration exhibits high utilization and mapping efficiency, making it well-suited for scenarios requiring minimal memory overhead. However, it results in longer compute cycles due to the smaller array size, which limits parallelism and increases the time taken per operation.
- 256x256: This configuration strikes a balanced performance, offering a compromise between memory bandwidth requirements and computational efficiency. It provides moderate memory access demands while maintaining a reasonable compute cycle time, making it versatile for a variety of workloads. The moderate array size ensures better utilization of resources compared to the smaller 128x128 configuration.
- 512x512: While this configuration boasts the shortest compute cycles due to its larger array size, it suffers from low utilization and mapping efficiency. The larger array results in significant resource underutilization, particularly when the workload cannot fully exploit the available parallelism. This configuration tends to waste memory bandwidth and processing resources, especially in tasks that do not fully scale with the array size.

F. Contribution

In summary, our contributions to this project include understanding the Scale-sim structure, applying the NVIDIA A100 configuration, replacing the default configuration settings, and introducing the concept of generating diffusion model topologies, which we then constructed within Scale-sim.

V. Results

A. Overview of Differences

- 128x128 model
  - Compute Cycles: Longest (e.g., Layer 0 requires 14807 cycles)
  - Hardware Utilization: Highest (Layer 4 reaches 88.07%)
  - Mapping Efficiency: Highest (94.53%)
  - Average DRAM Bandwidth: IFMAP, Filter, and OFMAP DRAM bandwidth demands are relatively low, with OFMAP bandwidth at 128 words/cycle.
- 256x256 model
  - Compute Cycles: Significantly reduced (Layer 0 requires 10475 cycles)
  - Hardware Utilization: Moderate (Layer 4 is 41.20%)
  - Mapping Efficiency: Moderate (Layer 4 is 47.27%)
  - Average DRAM Bandwidth: OFMAP bandwidth increases to 256 words/cycle.
- 512x512 model
  - Compute Cycles: Shortest (Layer 0 requires only 8309 cycles)
  - Hardware Utilization: Lowest (Layer 4 is 9.12%)
  - Mapping Efficiency: Lowest (Layer 4 is 11.82%)
  - Average DRAM Bandwidth: OFMAP bandwidth increases significantly to 511 words/cycle.

Fig. 1-3 show the visualized results of the simulation for GUID128, GUID256 and GUID512 using A100 accelerator.

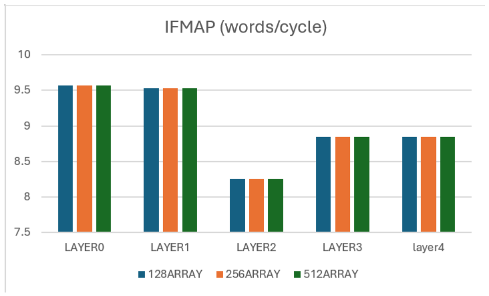

Fig. 1. IFMAP Bandwidth Across Layers for Different Models

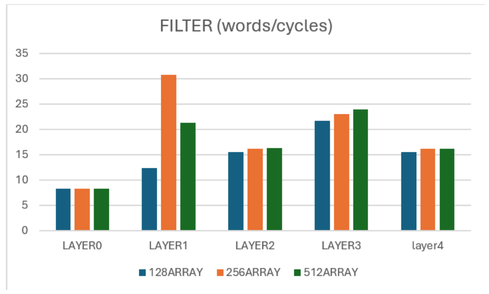

Fig. 2. Filter Bandwidth Across Layers for Different Models

B. Evaluation I : Average Memory Access

This chart illustrates the average memory access bandwidth for three types of data: IFMAP (Input Feature Map), Filter, and OFMAP (Output Feature Map), across different model configurations—128x128, 256x256, and 512x512. The x-axis

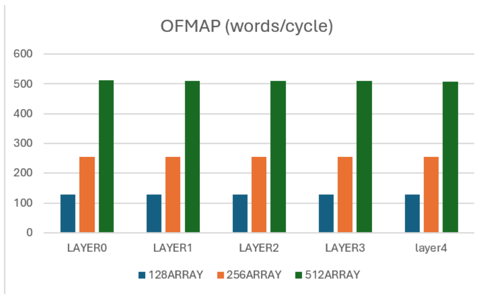

Fig. 3. OFMAP Bandwidth Across Layers for Different Models

represents the type of data, while the y-axis shows the memory access bandwidth.

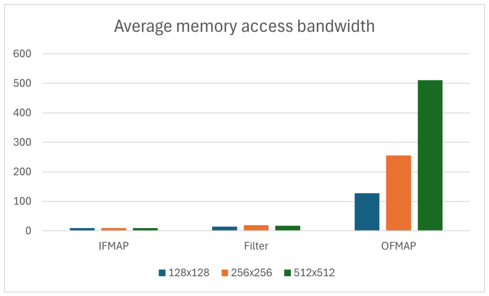

Fig. 4. Average Memory Access Bandwidth in Different models

Fig. 4 presents three series, each corresponding to different model sizes: 128x128, 256x256, and 512x512, depicted in blue, orange, and green, respectively. The bandwidth requirement for OFMAP is significantly higher compared to IFMAP and Filter across all model sizes, with the 512x512 model showing the highest memory bandwidth for OFMAP. In contrast, IFMAP and Filter exhibit relatively low and similar bandwidth demands, irrespective of model size.

Overall, the chart indicates that the output feature map (OFMAP) is the primary contributor to memory access bandwidth, particularly as the model size increases.

C. Evaluation II : Speedup

We use the information provided by /test_runs/log file our config generated/DETAILED_ACCESS_REPORT.csv to calculate the speedup, but it requires estimating the total access time based on access patterns across different memory levels. Speedup typically refers to the performance improvement of one system compared to another, such as an optimized system versus a baseline system. The start and stop cycles for each SRAM or DRAM access indicate the beginning and end of the access as following:

- SRAM IFMAP Read Time = SRAM IFMAP Stop Cycle - SRAM IFMAP Start Cycle
- SRAM Filter Read Time = SRAM Filter Stop Cycle - SRAM Filter Start Cycle
- SRAM OFMAP Write Time = SRAM OFMAP Stop Cycle - SRAM OFMAP Start Cycle
- DRAM IFMAP Read Time = DRAM IFMAP Stop Cycle - DRAM IFMAP Start Cycle
- DRAM Filter Read Time = DRAM Filter Stop Cycle - DRAM Filter Start Cycle
- DRAM OFMAP Write Time = DRAM OFMAP Stop Cycle - DRAM OFMAP Start Cycle

These times represent the time consumed for each type of access in the corresponding memory. We use the scale configuration in default scale-sim as the baseline cycle to calculate speedup among different models. The result is shown in **Fig. 5**.

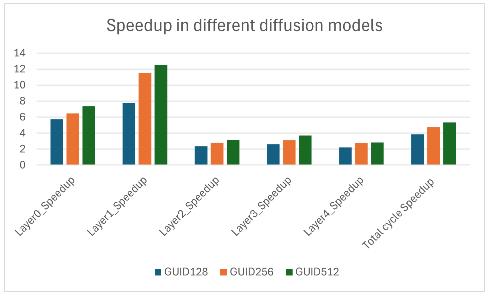

Fig. 5. Speedup in different diffusion models

In Layer 2, GUID256 and GUID512 achieve a notably higher speedup, with GUID512 reaching the highest (approximately 14x speedup). For other layers (Layer 3 and Layer 4), the speedup is lower and more consistent, with all models showing similar performance. Overall Speedup (Total Cycle Speedup) GUID512 demonstrates the best total speedup, slightly outperforming GUID256 and GUID128. GUID128 has the lowest total speedup, indicating that it may not fully utilize the hardware optimizations.

This likely reflects that Layer 2's data or computation aligns well with the current acceleration architecture (e.g., SRAM/DRAM access patterns, parallel compute units, etc.). Further analysis of Layer 2's characteristics may reveal optimization methods that can be extended to other layers. In summary, GUID512 achieves the highest total speedup, making it well-suited for compute-intensive applications. In Layer 3 and Layer 4, speedup improvements are limited, suggesting potential hardware bottlenecks in memory access or computational capacity.

- Optimizations scale better with larger models, especially for computationally intensive layers (like Layer 2).
- Uniform performance in Layers 3 and 4 suggests a bottleneck or inherent limitation in those stages across all model sizes.
- The total cycle speedup highlights that larger models leverage hardware or algorithmic optimizations more effectively.

D. Causes of Differences

1) Impact of Array Size:
    - Compute Cycles decrease as array size increases due to higher parallelism.

- Hardware Utilization and Mapping Efficiency decrease with larger arrays, especially in the 512x512 configuration, where the computational workload cannot fully utilize all hardware resources.
- Memory Bandwidth Requirements increase significantly with larger arrays because larger arrays generate more output data.

2) Memory Bottleneck:
   - In the 512x512 configuration, memory bandwidth becomes the primary performance bottleneck. Although compute cycles are the shortest, extremely high memory bandwidth demands lead to low hardware utilization.
3) Uneven Resource Allocation:
   - In larger configurations, computational demand is more distributed, leading to a decline in mapping efficiency. This is particularly evident in scenarios with lower input resolutions or smaller computational workloads, where larger arrays cannot efficiently allocate resources.

E. Explanation of Results

- GUID128: Suitable for resource-constrained scenarios, with high hardware utilization and mapping efficiency. However, longer compute cycles limit overall performance.
- GUID256: Provides a balance between performance and resource demands, with moderate compute cycles and reasonable memory requirements.
- GUID512: Ideal for high-demand scenario but suffers from low hardware utilization, poor mapping efficiency, and extremely high memory bandwidth requirements, leading to underutilized resources.

VI. Conclusion

The evaluation of Cambricon-D-inspired architectures using SCALE Sim demonstrates the scalability and limitations of different array configurations:

- **GUID128:** This model achieved the highest hardware utilization and mapping efficiency, making it ideal for resource-constrained scenarios. However, it suffers from longer compute cycles, which limit its overall performance in high-demand applications.
- **GUID256:** This best balances computational efficiency and resource demands. It features reduced compute cycles compared to 128x128 while maintaining moderate utilization and mapping efficiency, making it suitable for a wide range of applications.
- **GUID512:** While this configuration achieves the shortest compute cycles due to increased parallelism, it struggles with low hardware utilization and mapping efficiency, particularly in workloads that fail to fully utilize the available resources. Additionally, the high memory bandwidth demands result in performance bottlenecks, making it best suited for large-scale, high-load computational tasks.

Our findings indicate that tasks involving GUID128 are notably faster and more power-efficient, while those involving GUID256 and GUID512 necessitate greater computational power and bandwidth. However, these larger tasks can exploit parallelism and specialized hardware features to manage the increased workload with greater efficiency. A comparative analysis with the research presented in *Cambricon-D: Full-Network Differential Acceleration for Diffusion Models* [5] reveals a congruent conclusion. Smaller configurations, such as 128x128 arrays, are particularly suited for scenarios demanding high efficiency with minimal resource overhead. Intermediate sizes, like 256x256 arrays, offer an optimal balance, providing versatility across a range of workloads. In contrast, larger configurations, such as 512x512 arrays, excel primarily in high-demand contexts where parallelism can be fully leveraged. When such resources are not fully utilized, however, their benefits become diminished due to the inefficiency of underuse.

This analysis highlights the importance of selecting the appropriate hardware configuration based on workload characteristics and performance requirements, ensuring scalability and efficient resource utilization in diffusion model acceleration.

A. Challenges Faced

In summary, the main challenges we encountered during this project were as follows:

- Completing the implementation of the Cambricon-D algorithm on Scale-Sim and overcoming all the error we encountered.
- Generating the guided-diffusion model topologies and integrating them into Scale-Sim, the task that involved not only ensuring the correctness of the model structure but also dealing with resource limitations and ensuring smooth integration into the simulation environment.

B. Future work

Our future work will focus on addressing the technical challenges we encountered while executing Cambricon-D successfully in Scale-sim. This includes resolving compatibility issues, improving performance optimizations, and fine-tuning the simulation environment to better support the architecture. In addition, we aim to enhance NDP capabilities by exploring more complex operations directly at memory and aiming to extend the capabilities of the current Cambricon-D architecture.

Another key aspect of our future research will be extending the capabilities of the current Cambricon-D architecture, particularly in areas such as local delta accumulation or partial activations, thereby reducing unnecessary computations and memory usage, which could lead to significant improvements in both speed and energy efficiency. Ultimately, our goal is to provide insights into how these advancements can be applied to broader use cases, enabling Cambricon-D to handle a wider range of workloads while improving overall efficiency across different AI tasks.

ACKNOWLEDGMENTS

Thanks to the contributions of our team members. Emily Chang edited the report in LaTeX and managed the GitHub repository. Yu-Ting Lin provided the report outline of report and contributed most of experimental data and results. Chi-Wei Ho was responsible for presentation video and provided various technical solutions.

VII. Appendix

```
class CambriconDConvReLU(nn.Module):
def __init__(self,
save_disk_space=False,
verbose=True,
config=”,
topology=”,
input_type_gemm=False):

# super(CambriconDConvReLU, self).__init__()

# Data structures
self.config = scale_config()
self.topo = topologies()

# File paths
self.config_file = ”
self.topology_file = ”

# Define only convolution and ReLU layers from the original architecture
# Assume some example convolution layers (the actual configuration may vary)
self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)

# ReLU activation
self.relu = nn.ReLU()

# Sign bit tensor storage for each convolution layer
self.sign_bits =

# PE array parameters for handling deltas and outliers
self.pe_array =
’conv1’: ’bitwidth’: 8, ’outlier_count’: 0,
’conv2’: ’bitwidth’: 8, ’outlier_count’: 0,
’conv3’: ’bitwidth’: 8, ’outlier_count’: 0,


# Member objects
self.runner = r.run_nets()
self.runner = sim()

# Flags
self.read_gemm_inputs = input_type_gemm
self.save_space = save_disk_space
self.verbose_flag = verbose
self.run_done_flag = False
self.logs_generated_flag = False

self.set_params(config_filename=config, topology_filename=topology)

def forward(self, x):
# Forward pass with only Conv and ReLU layers, maintaining sign bits and handling PE array
x, self.sign_bits[’conv1’], self.pe_array[’conv1’] = self.process_layer(x, self.conv1, ’conv1’)
x, self.sign_bits[’conv2’], self.pe_array[’conv2’] = self.process_layer(x, self.conv2, ’conv2’)
x, self.sign_bits[’conv3’], self.pe_array[’conv3’] = self.process_layer(x, self.conv3, ’conv3’)
return x

def process_layer(self, x, conv_layer, layer_name):
# Perform convolution
x = conv_layer(x)

# Maintain sign bit tensor
sign_bits = torch.sign(x).clamp(min=0).byte()

# Handle PE array deltas and outliers
delta = x - torch.round(x) # Example delta calculation
outlier_mask = delta.abs() > 0.1 # Identify outliers (threshold is an example)
outlier_count = outlier_mask.sum().item()

# Update PE array info
pe_info =
’bitwidth’: 8, # Assume an 8-bit representation
’outlier_count’: outlier_count


# Apply ReLU
x = self.relu(x)
return x, sign_bits, pe_info
```

References

[1] R. B. Abdelhamid, Y. Yamaguchi, and T. Boku, “A scalable many-core overlay architecture on an hbm2-enabled multi-die fpga,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 1, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3547657

[2] N. Corporation, “NVIDIA Ampere Architecture Whitepaper,” NVIDIA, Tech. Rep., 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[3] P. Dhariwal and A. Nichol, “Diffusion models beat gans on image synthesis,” *CoRR*, vol. abs/2105.05233, 2021. [Online]. Available: https://arxiv.org/abs/2105.05233

[4] J. Ho, A. Jain, and P. Abbeel, “Denoising diffusion probabilistic models,” *CoRR*, vol. abs/2006.11239, 2020. [Online]. Available: https://arxiv.org/abs/2006.11239

[5] W. Kong, Y. Hao, Q. Guo, Y. Zhao, X. Song, X. Li, M. Zou, Z. Du, R. Zhang, C. Liu, Y. Wen, P. Jin, X. Hu, W. Li, Z. Xu, and T. Chen, “Cambricon-d: Full-network differential acceleration for diffusion models,” in *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[6] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “A systematic methodology for characterizing scalability of dnn accelerators using scale-sim,” in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.

[7] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “Scale-sim: Systolic cnn accelerator simulator,” *arXiv preprint arXiv:1811.02883*, 2018.

[8] F. Yu, Y. Zhang, S. Song, A. Seff, and J. Xiao, “LSUN: construction of a large-scale image dataset using deep learning with humans in the loop,” *CoRR*, vol. abs/1506.03365, 2015. [Online]. Available: http://arxiv.org/abs/1506.03365