CSCE 614 Group 15 Final Report

Akeem Olawoyin akeemedes@tamu.edu
Eden Garza edengarza@tamu.edu
Eugene Lin yuchunlin@tamu.edu
Haolin Zhang chris_zhang@tamu.edu

***Abstract*—In this technical report, we reproduce Cocco, a novel approach leveraging Genetic Algorithms (GAs) to optimize subgraph partitioning for efficient execution of machine learning models. Building upon the methodology of the original work, we integrate the Timeloop tool to simulate and evaluate the performance of various partitioning strategies on target hardware. By encoding subgraph partitioning as a genetic representation, our approach iteratively evolves partitioning solutions through selection, crossover, and mutation. This enables us to identify configurations that maximize compute efficiency and minimize communication overhead. Experimental results demonstrate the effectiveness of Cocco in achieving significant performance improvements, showcasing its potential for optimizing large-scale machine learning workloads.**

I. Introduction

The increasing complexity of neural network architectures, driven by advances in artificial intelligence, has posed significant challenges for hardware systems designed to execute them efficiently. Modern deep-learning models often require intricate memory management and optimized dataflows to ensure that computational resources are utilized effectively while minimizing communication overhead and memory costs. To address these challenges, the Cocco [4] framework was introduced as a hardware-mapping co-exploration approach, targeting the optimization of subgraph partitioning for deep learning accelerators. By leveraging graph-level partitioning and hardware-aware exploration, Cocco [4] aims to reduce external memory access and energy costs, ultimately enhancing the overall performance and energy efficiency of machine learning systems.

In this technical report, we present an experimental reproduction of the Cocco [4] framework, focusing on its ability to find optimal subgraph partitioning schemes using three distinct methodologies: Genetic Algorithms (GA), Greedy algorithms, and Enumeration-based methods. Each approach is explored for its strengths and trade-offs in navigating the complex optimization space of subgraph partitioning. Our implementation builds upon the original framework by using Timeloop [2], a tool for simulating and analyzing hardware configurations, to evaluate the computational and memory performance of different partitioning strategies.

The Genetic Algorithm is employed for its ability to explore large and complex search spaces, avoiding local minima through mutation and crossover operations, while ensuring a comprehensive search for optimal solutions. Greedy algorithms are included for their simplicity and speed in generating reasonable solutions, particularly in scenarios where rapid decision-making is needed. Enumeration-based methods, on the other hand, provide an exhaustive approach, systematically exploring all possible partitioning configurations to identify the optimal solution. This method, though computationally intensive, serves as a baseline to evaluate the effectiveness of heuristic approaches.

Our reproduction effort extends the original Cocco [4] framework by systematically comparing these three methodologies across diverse neural network models, ranging from regular to highly irregular structures. We examine how different partitioning strategies impact the trade-offs between on-chip memory usage, external communication, and computation latency. By iteratively refining and validating these methods, our study highlights the robustness and versatility of Cocco [4] in optimizing subgraph execution.

This work not only demonstrates the effectiveness of the Cocco [4] framework but also provides insights into the potential of co-exploration techniques for hardware-aware optimization of deep learning models. Our findings contribute to the broader field of hardware-software co-design, offering a detailed analysis of how partitioning strategies and hardware configurations can be tuned to achieve optimal performance for memory-intensive neural networks. Through this reproduction and extension, we aim to further the understanding of subgraph partitioning and its critical role in enabling efficient and scalable deep learning inference on modern hardware accelerators.

The article is organized as follows: Section II describes the problem such as the partition scheme and the objective of the problem. Section III discusses the proposed solution in the original paper such as the genetic algorithm, greedy algorithm, and the enumeration-based algorithm. Section IV discusses the evaluation methods for the setup of the experiment and the models we will use in this paper. Section V analyzes the results. Section VI concludes the paper. Section VII shows the contribution of every group member in this project.

II. Problem Description

Modern deep-learning models are represented as computational graphs consisting of layers (nodes) and data dependencies (edges) in a Directed Acyclic Graph(DAG). Efficient execution of these graphs on hardware accelerators is critical for achieving high performance and energy efficiency. However, the growing complexity of neural network architectures, including both regular structures like CNNs and irregular structures like Transformers that are widely used for Large Language Models, poses significant challenges for memory and communication optimization.

A key bottleneck lies in managing the limited on-chip memory capacity of hardware accelerators. Small on-chip memory requires frequent external memory access, increasing

energy consumption and bandwidth demands. Conversely, larger memory configurations reduce communication overhead but incur significant silicon costs and energy penalties. This creates a trade-off between memory capacity, communication costs, and overall system performance. To address this, it is essential to partition a computational graph into subgraphs that can be efficiently executed within the constraints of the hardware memory while minimizing external communication. The problem can be formally defined as follows:

A. Graph Partitioning

Given a directed acyclic graph (DAG) $G = (V, E)$, where V represents layers (nodes) of a neural network and E represents the dependencies between layers, the goal is to partition the graph into a series of subgraphs. Each subgraph must fit within the on-chip memory capacity and maintain the data dependency constraints between layers. In practice, we achieve these requirements by traversing the graph in topological order.

B. Optimization Objective

1) External Memory Access (EMA): Reducing the frequency and volume of data transferred between external memory (e.g., DRAM) and on-chip memory.

2) Communication Overhead: Minimizing bandwidth requirements for transferring weights and intermediate activations between subgraphs.

3) Energy Consumption: Reducing the energy required for computation. This is directly realted to the previous two objectives, as both external memory accesses and communication overhead increase the energy costs associated with training a deep neural network.

C. Constraints

1) SRAM size: On-chip memory capacity limits the size of each subgraph.

2) Ordering: Subgraphs must preserve the topological order of the original graph to ensure valid execution.

3) Avoid recomputation: Intermediate results needed by subsequent subgraphs must be managed effectively to avoid recomputation.

The vast search space for partitioning large and irregular graphs presents additional challenges. Simple heuristics like Greedy algorithms or enumeration-based approaches may fail to explore the entire optimization space effectively, often getting trapped in local optima. Advanced methods such as Genetic Algorithms (GA) are promising as they enable efficient exploration of large search spaces while avoiding local minima by iteratively evolving candidate solutions.

III. Proposed Solution

To tackle the intertwined challenges of graph partitioning and hardware mapping in deep learning accelerators, we choose a synergistic framework that blends intelligent algorithms, hardware-aware strategies, and novel execution flows. At its core, the solution transforms the problem into an adaptive exploration process, balancing memory constraints, energy efficiency, and computational performance while embracing the complexity of modern neural networks.

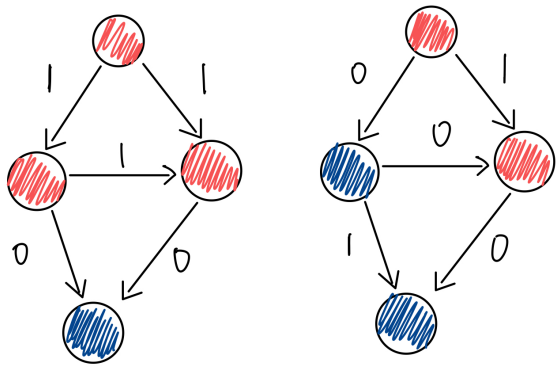

Fig. 1. Decision variables visualization. If a decision variable on an edge is 1, the edge's nodes are grouped to a subgraph.

Rather than treating subgraph partitioning as an isolated, static problem, we redefine it as a collaborative interplay between computation and memory. Subgraphs are dynamically crafted to maximize on-chip memory utilization, minimize redundant data movement, and align with the accelerator's natural data flow. By weaving layers together into meaningful partitions, the solution prioritizes coherence in execution while embracing the diversity of model architectures.

The heart of the solution lies in a co-exploration framework that views hardware memory configuration and graph partitioning as a single optimization landscape. Instead of fixing one and optimizing the other, we navigate their interdependencies to discover configurations that amplify overall efficiency. This unified approach eliminates the disconnect between the model's computational structure and the hardware's physical constraints.

At the forefront of our strategy is the integration of diverse optimization techniques, each contributing unique strengths:

A. Genetic Algorithms (GA)

Inspired by the resilience of natural evolution, GA introduces a vibrant search process where candidate solutions evolve, adapt, and refine through crossover and mutation. This injects flexibility and creativity into the search for optimal configurations.

1) Chromosome Encoding and Decoding: We can regard the graph partition problem as a 0 and 1 assignment problem. We encode each edge as a gene in a chromosome so that each gene is either 0 or 1 and the length of the chromosome is simply the number of the edges in the graph. By doing this, we can determine the subgraph. For example, as shown in Fig. 1, we determine the subgraph by the edges connected between nodes. If the edge connecting two nodes is 1, then those two nodes are grouped into a subgraph.

2) Chromosome Selection Scheme: The selection scheme is important to the genetic algorithm, which determines the

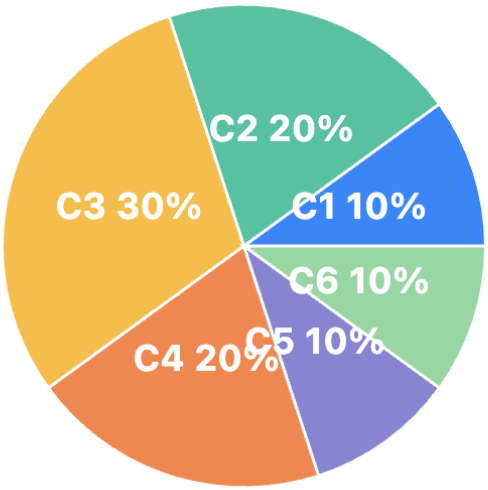

Fig. 2. The visualization of the roulette wheel selection scheme. If the chromosome has a better fitness value, it has a bigger probability of being selected to generate offspring.

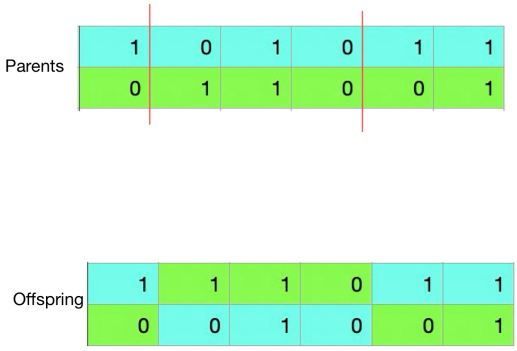

Fig. 3. The visualization of the double-point crossover

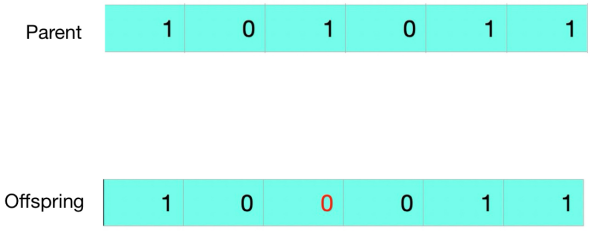

Fig. 4. The visualization of the single point mutation. The third bit in the parent chromosome has been flipped to generate new offspring.

offspring of the next generation. We may consider the steady-state selection scheme in this paper. The steady-state selection scheme in genetic algorithms involves incrementally updating the population by replacing only a small number of individuals in each generation, typically one or two. New offspring are generated through selection, crossover, and mutation, and then inserted into the population, often replacing the least-fit members to maintain or improve overall fitness. This approach ensures gradual evolution, preserves population diversity, and reduces the computational cost of evaluating an entirely new generation. Steady-state selection is particularly advantageous in our experiments since it requires continuous optimization or where fitness evaluations are expensive.

We may also consider another important selection scheme, the roulette wheel selection scheme. It utilizes the probability based on the fitness value to determine the offspring for the next generation. For example, as shown in Fig. 2, the 3rd chromosome has a better fitness value so it has a bigger probability of being selected. This maintains the diversity of the offspring since even though the fitness value of a chromosome is not good, it's still able to be selected. With this scheme, the algorithm will not likely fall into some local optimal; instead, this ensures the global search ability of the genetic algorithm.

3) Crossover and Mutation: Crossover and mutation are essential operators in genetic algorithms, facilitating the exploration and exploitation of the solution space. Crossover combines genetic material from two parent solutions to produce offspring, enabling the exchange of beneficial traits. Common methods include single-point, multi-point, and uniform crossover, each determining how parental information is mixed. Mutation, on the other hand, introduces random changes to an individual's genetic representation, ensuring diversity and helping the algorithm escape local optima. Since the gene is represented as a binary number, the mutation technique only does bit-flipping selected genes. Together, these operators balance the search process: crossover emphasizes exploration by creating novel combinations, while mutation ensures diversity and robustness against falling into the local optimal. For example, as shown in the Fig. 3, we cut two chromosomes on two points and swapped two segments to generate new offspring. For another example, as shown in Fig. 4, the third gene has been flipped to generate the new offspring.

B. Greedy Algorithms

In the original paper, the greedy algorithm is utilized as a baseline to compare with others. The greedy algorithm is inclined to search the solution space with a lower cost function value. However, it is likely to fall into the local optimal. Nevertheless, the greedy approach acts as a sprinter, generating

Algorithm 1 Genetic Algorithm for Graph Partitioning

Require: Graph $G = (V, E)$, population size P, number of generations G_n, mutation rate m_r
Ensure: Optimized partitioning scheme

```
Initialize a population of size P with random partitioning schemes
for generation = 1 to G_n do
    Evaluate the fitness of each individual in the population
    Select individuals based on fitness using a selection strategy (e.g., tournament selection)
    Perform crossover on selected individuals to generate offspring
    Apply mutation to offspring with probability m_r
    Replace the current population with the offspring
end for
return Best partitioning scheme found
```

fast preliminary results that provide a baseline to compare with others.

Algorithm 2 The Greedy Algorithm for Graph Partitioning

Require: Graph $G = (V, E)$, memory capacity C
Ensure: Partitioning scheme

```
Initialize each node v ∈ V as its own partition
while there exists a pair of partitions that can be merged do
    Find the pair of partitions that maximize benefit under memory constraint C
    Merge the selected pair of partitions
end while
return Final partitioning scheme
```

C. The Enumeration-Based Algorithm

Enumeration-based algorithms offer a deterministic and exhaustive approach to solving optimization problems by exploring all possible configurations in the search space. While computationally intensive, this method serves as a reference point to evaluate the efficacy of heuristic approaches. In our graph partition algorithm, the enumeration algorithm simply enumerates all possibilities of a combination of subgraphs. Along with the growing size of the model, the enumeration-based algorithm is not used to find the subgraph.

This ensemble of methods ensures the solution is both adaptive and robust, capable of responding to the irregularities of large neural networks.

Central to our framework is the Timeloop [2] simulator, which acts as both a guide and validator in the optimization process. Timeloop [2] precisely quantifies the ripple effects of each partitioning decision, measuring metrics such as memory access patterns, communication costs, and energy efficiency. Its insights form a feedback loop, enabling iterative refinement of the partitioning and memory configuration strategies.

Algorithm 3 Enumeration-Based Algorithm for Graph Partitioning

Require: Graph $G = (V, E)$, memory capacity C
Ensure: Optimal partitioning scheme

```
Generate all possible partition schemes for G
Initialize the optimal cost C_opt = ∞ and the corresponding scheme
for each partition scheme P_i do
    Compute the cost C(P_i) based on metrics such as external memory access and energy
    if C(P_i) < C_opt then
        Update C_opt ← C(P_i)
        Update the optimal scheme
    end if
end for
return Optimal partitioning scheme
```

In rethinking execution, we adopt a consumption-centric execution flow. Unlike traditional methods that precompute and store intermediate results, this philosophy generates outputs on demand. By producing only what is needed at any given moment, the approach significantly reduces memory overhead, aligning computation more closely with practical requirements. This agility transforms subgraph execution into a lean, efficient process that thrives within the constraints of limited on-chip memory.

Ultimately, the solution is more than a collection of techniques—it is a cohesive and adaptive framework that learns and evolves. By integrating intelligent algorithms, precise simulation tools, and innovative execution strategies, the framework addresses the challenges of memory-constrained accelerators. It redefines how we optimize the relationship between neural networks and the hardware they run on, offering a blueprint for efficient, scalable, and energy-conscious execution in the next generation of deep learning systems. Through this comprehensive solution, we not only tackle the immediate problem but also lay the groundwork for advancing hardware-aware neural network optimization.

IV. Evaluation Methods

Our evaluations were conducted on a MacBook Pro equipped with an Apple M1 Max chip featuring 10 CPU cores and 16 GB of unified memory. The device operates with active cooling, ensuring no thermal throttling occurred during the tests. System performance metrics, such as memory and CPU utilization, were monitored using macOS's built-in Activity Monitor.

We utilized a Docker image pulled from the official Timeloop repository, adhering to the recommended installation instructions. All necessary Python packages were installed to meet the dependencies for running the Timeloop framework. During evaluation, Docker reported a peak memory usage of 7.98 GB and CPU utilization of 139 percentile.

We ran the experiments for the Vgg16 model [3], ResNet50 Model [1], and the Transformer model [5] with the greedy

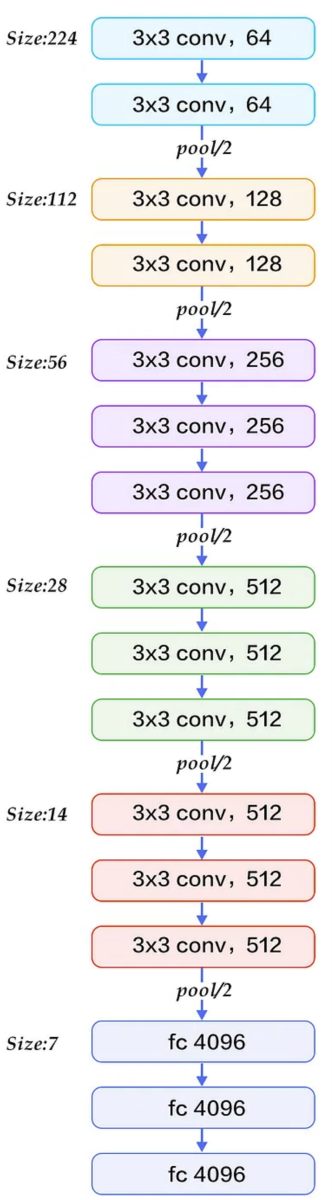

Fig. 5. VGG16 model structure

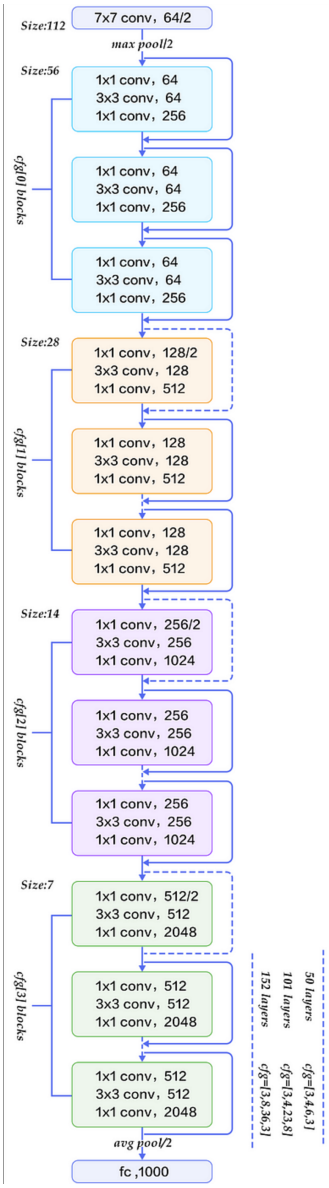

Fig. 6. ResNet50 model structure

algorithm, the enumeration-based algorithm, and the genetic algorithm. These models are shown in Fig. 5, Fig. 6, and Fig. 7, respectively. Since the enumeration-based algorithm will need to search the space over 2^{68}, we terminated the algorithm in a reasonable time. The transformer model is even more complex than the ResNet50; thus, we are not able to give the result in a reasonable time to compare with others.

V. RESULTS

Our experiments show that Cocco [4] is able to perform as well as the other approaches, given the right resources.

Starting by evaluating the performance of the three different approaches (Greedy, Enumeration, and Genetic (Cocco)) on the VGG16 architecture, we see that Cocco [4] is able to find the optimal mapping just as effectively as the other two approaches. This is expected since the model is simple and straightforward, lending itself to the greedy approach. In this case, it is important to note that Cocco [4] and the enumeration-based approach took far longer than the greedy approach.

On the ResNet architecture, we see an interesting story painted by Figures 8 and 9. In Fig. 8, we see that the greedy algorithm and Cocco [4] find the optimal solution while the enumeration-based approach does not, but in Fig. 9 we see that the enumeration-based approach and Cocco [4] outperform the greedy algorithm. This means that, while the other approaches may be finding optimal solutions on one metric, they may not be considering others. Cocco [4] is able to balance the needs of these multiple requirements and remain optimal in both

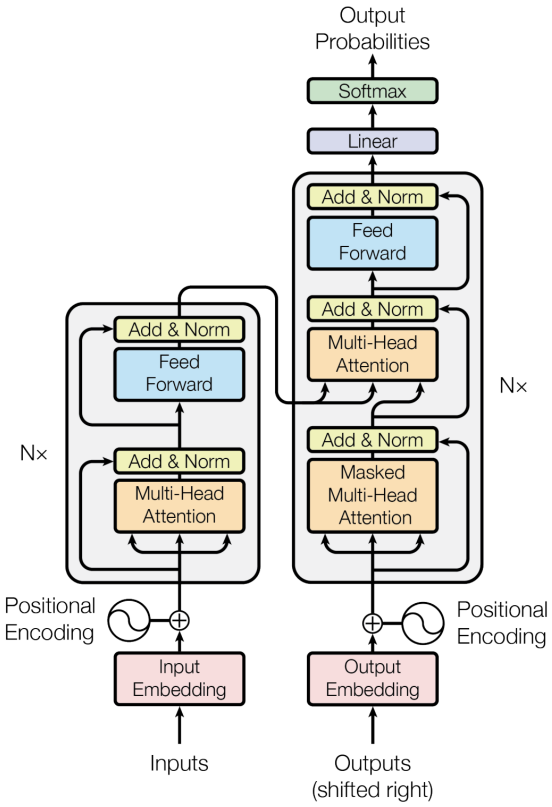

Fig. 7. Transformer model structure

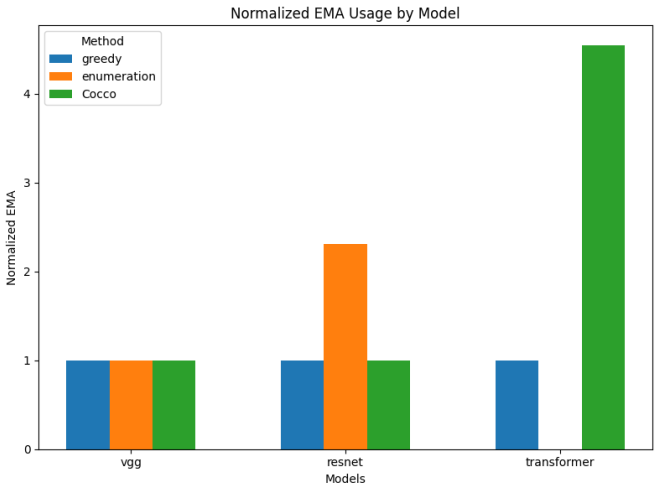

Fig. 8. A comparison of external memory accesses (EMA) by algorithm across all three architectures: VGG16, ResNet, and Transformer.

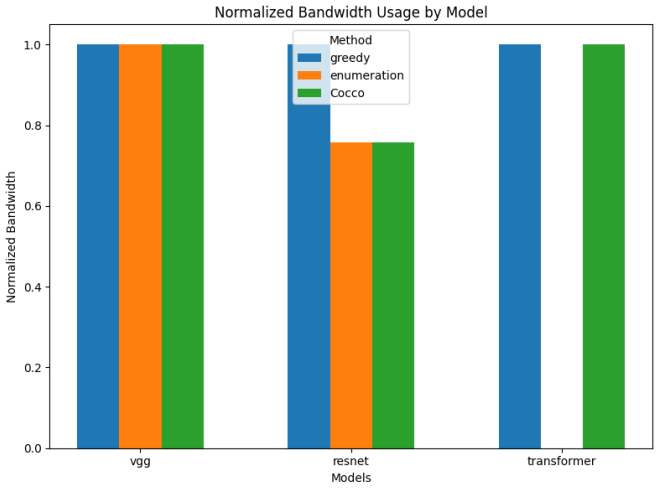

Fig. 9. A comparison of bandwidth usage by algorithm across the same architectures as in Fig. 8

categories.

The Transformer architecture was the most complex architecture of the three and the enumeration-based approach paid the cost, not being able to complete in a reasonable amount of time. The greedy algorithm was able to complete quickly, however we can see that, although it was able to reduce external memory accesses, it has the same bandwidth requirements as Cocco [4]. For this model in particular, time was a big constraint. The genetic algorithm was limited in its ability to run, leading to execution ending prior to finding the optimal solution.

Taking the results across the three different models into account, it is clear to see that each approach has both positive and negative attributes. For a small model such as VGG16, all three approaches achieved the optimal solution, however the greedy approach did so the quickest. The tradeoff becomes evident as we increase complexity to ResNet, where the greedy approach and the enumeration-based approach both achieve optimal performance but on different metrics while Cocco [4] does so in both. Finally, the Transformer model, although not entirely ideal for Cocco [4] based on our experimental results, highlights the biggest downside of the enumeration-based approach while showing that, even with being cut off prior to converging on the most optimal solution, Cocco [4] can content with an algorithm designed to quickly converge on what could be a local minimum.

VI. Conclusion

As deep neural networks continue to gain complexity and popularity, the need for efficient training methods has become paramount. The methods introduced in the original Cocco [4] paper and reproduced by us show that utilizing a Genetic Algorithm (GA) to efficiently create and schedule the evaluation of deep neural networks as subgraphs has the potential to reduce external memory accesses and bandwidth requirements during training. Additionally, due to Cocco's [4] ability to handle arbitrary graph patterns both for subgraph partitioning and memory mapping, the framework proves to be an enticing choice for a broad audience.

In simplistic models, such as VGG16, Cocco performs equal to greedy and enumeration-based methods, showing that it is viable in the simple case. As complexity increases with models

such as ResNet of Transformer, we begin to see the benefit of the GA by being able to optimize for both external memory communications as well as memory bandwidth requirements. The key feature of being able to optimize for both, as opposed to falling into a suboptimal local minimum like the greedy algorithm approach or not scaling efficiently with complex neural networks like the enumeration-based approach, is what distinguishes Cocco as a powerful approach for designing hardware solutions for machine learning.

VII. Work Division

- Akeem Olawoyin: Algorithms
- Eden Garza: Timeloop and documentation
- Eugune Lin: Timeloop, experiments, and documentation
- Haolin Zhang: Documentation

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[3] K. Simonyan, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] Z. Tan, Z. Zhu, and K. Ma, "Cocco: Hardware-mapping co-exploration towards memory capacity-communication optimization," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 69–84. [Online]. Available: https://doi.org/10.1145/3617232.3624865

[5] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.