Cocco: Hardware-Mapping Co-Exploration towards Memory Capacity-Communication Optimization

Zhanhong Tan
tanzh19@mails.tsinghua.edu.cn
IIIS, Tsinghua University
Beijing, China

Zijian Zhu
zhuzj23@mails.tsinghua.edu.cn
IIIS, Tsinghua University
Beijing, China

Kaisheng Ma*
kaisheng@mail.tsinghua.edu.cn
IIIS, Tsinghua University
Beijing, China

Abstract

Memory is a critical design consideration in current data-intensive DNN accelerators, as it profoundly determines energy consumption, bandwidth requirements, and area costs. As DNN structures become more complex, a larger on-chip memory capacity is required to reduce data movement overhead, but at the expense of silicon costs. Some previous works have proposed memory-oriented optimizations, such as different data reuse and layer fusion schemes. However, these methods are not general and potent enough to cope with various graph structures.

In this paper, we explore the intrinsic connection between network structures and memory features to optimize both hardware and mapping. First, we introduce a graph-level execution scheme with a corresponding dataflow and memory management method. This scheme enables the execution of arbitrary graph patterns with high data reuse and low hardware overhead. Subsequently, we propose Cocco, a hardware-mapping co-exploration framework leveraging graph-level features of networks. It aims to minimize communication overhead, such as energy consumption and bandwidth requirements, with a smaller memory capacity. We formulate the graph-partition scheduling and memory configuration search as an optimization problem and employ a genetic-based method to achieve efficient co-exploration for large and irregular networks. Experiments demonstrate that Cocco obtains lower external memory access, lower bandwidth requirements, and more stable optimization for graph partition compared to the greedy algorithm and dynamic programming introduced in prior works. Cocco also reduces the costs by 1.89% to 50.33% using co-exploration compared to other typical methods.

---

*Corresponding author.

1 Introduction

The evolution of neural network topology has driven the remarkable progress of artificial intelligence from the early single-layer perceptron (SLP) [45, 54] and multi-layer perceptron (MLP) [17, 22, 39] to modern DNNs with plain [36, 57]/inception [59]/residual [20, 55] structures based on manual design, and even irregular structures using neural architecture search (NAS) [53, 75] or random network generation [68]. These technological innovations have resulted in increasingly complex computation graphs, which pose challenges for efficient memory design and deployment.

Memory design is crucial in the accelerator system, as it performs data preparation at the start of each processing stage according to the scheduling scheme, determining energy consumption, bandwidth requirements, and area costs. Figure 1 shows the trade-off between the on-chip memory size and the external memory access in DNN accelerators. A smaller on-chip buffer (left side) saves area but requires more data reloading. A larger buffer (right side) can reduce external memory access and save energy and bandwidth but at the cost of increasing the memory overhead. An excessively large SRAM may not be feasible due to the high silicon area cost, typically ranging from 1 to 2 mm^2/MB in 12nm, and the high energy overhead, dozens of times that of a MAC operation for a large SRAM.

Therefore, the **key problem** is: *between the two extremes in Figure 1, how to find an appropriate memory configuration with efficient workload mapping and data management, especially under the growing complexity of neural network architectures.*
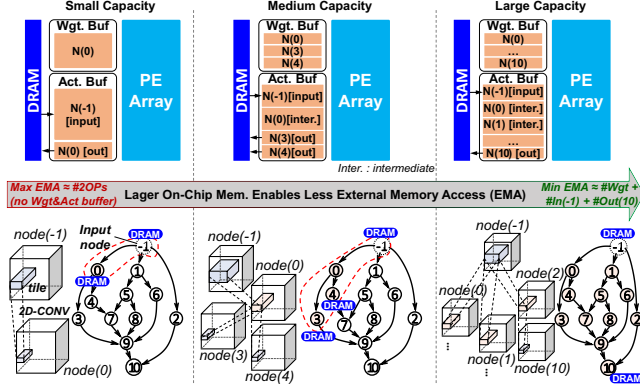
Figure 1. The effect of different memory capacities for a computation graph. Intermediate results can be buffered in the on-chip memory if it is large enough. The on-chip memory of small capacity can only buffer two nodes (marked in the red dotted box), and the larger memory can cover a larger subgraph (right side).

The critical status of memory design has attracted extensive research. Most previous studies focus on simple layer-level optimization (the left one of Figure 1) by applying loop transformation techniques such as tiling and reordering to fit the memory size and reuse the on-chip data [23, 43, 44, 61, 70]. In addition, several works also guide the memory capacity and hierarchy design using design-space exploration [12, 32, 37, 66, 67]. However, these layer-level optimizations are confined to the limited intra-layer reuse, which is insufficient for memory-intensive networks. A subgraph-level scheme (e.g., the middle one and the right one of Figure 1) provides a larger optimization space via inter-layer reuse [3, 4, 38, 73] to reduce the I/O overhead. Therefore, this paper aims to *leverage the subgraph-level computing flow to* optimize the memory capacity *and* external communication *for networks with any topology.*

However, there are **three primary challenges** to fully exploit the subgraph-level optimization.

First, *we need a general execution flow for any sub-graph.* Due to the various kernel sizes and strides, a parent node in a subgraph may have unbalanced data requirements from its consumers, which makes it difficult to determine the tensor tiling scheme and the memory allocation for each node (layer). In the traditional single-layer execution, we usually divide a large tensor into loop tiles, which are processed through a series of regular computing steps. Similarly, we want the sub-graph execution to be a series of elementary computing steps with a simple control flow.

Second, *we require a suitable memory management method for the subgraph execution.* Due to complicated dependency among nodes in a subgraph, careful management is needed to reuse overlapping and inter-layer intermediate data.

Solving these two challenges contributes to a basic hardware execution model compatible with subgraph-level optimization. However, we also encounter the third challenge: *how to partition a model into subgraphs and how much memory to allocate.* The optimization space is huge, so we need to devise a search method with high sampling efficiency to find a proper subgraph partition and memory configuration result.

In this paper, we first introduce a complete graph-level scheme for memory. In particular, it contains a consumption-centric flow that enables the execution of arbitrary subgraphs with low memory footprints (*for challenge 1*). Accordingly, we provide an explicit memory dataflow and the corresponding memory management scheme for effective data reuse (*for challenge 2*). Building on the graph-level memory scheme, we propose Cocco, a hardware-mapping co-exploration framework, to establish a connection between model features and the memory configuration (*for challenge 3*).

Cocco aims to find a combination of on-chip buffers and the corresponding graph-level scheduling for lower memory and communication overhead. In particular, we develop a genetic-based algorithm to efficiently explore the search space of graph partitions and the associated memory configuration for a series of neural networks.

In summary, this work makes the following contributions:

- **Subgraph execution scheme.** We first introduce a consumption-centric flow to determine a low-cost execution sequence by throttling and aligning the dataflow.
- **Efficient dataflow and memory management** for subgraph data reuse. We propose a memory management scheme featuring multiple reconfigurable regions and the corresponding dataflow to support arbitrary subgraph execution with full data reuse.
- **Hardware-mapping co-exploration framework.** Based on the subgraph execution scheme and memory dataflow, we propose Cocco, a genetic-based framework combining the graph-level partition and memory design-space exploration together. Cocco achieves 1.89% to 50.33% lower costs (lower communication with a smaller size) using co-exploration in contrast to other methods.

2 Background and Motivation

2.1 Design of Neural Network Accelerators

The DNN accelerator unit is the most basic execution unit in a computing system, on top of which, we can scale it out to many-core, many-socket, and many-drawer systems [24, 40, 48, 60]. An accelerator unit usually employs a processing element (PE) array on a sophisticated interconnection network to enable efficient tensor-level computation. Each PE typically contains local scratchpads and ALUs to process basic data packets. The global buffer and the weight buffer store activations and weights, and they are generally
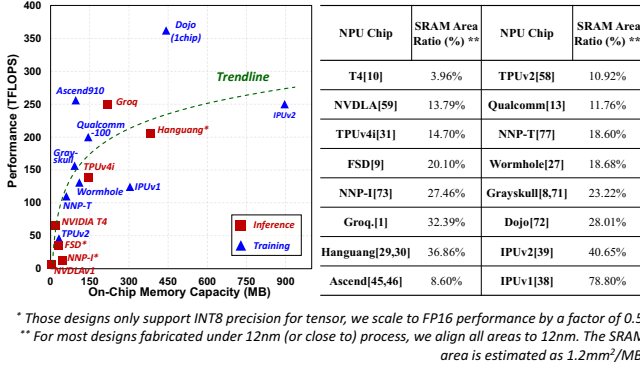
* Those designs only support INT8 precision for tensor, we scale to FP16 performance by a factor of 0.5.
** For most designs fabricated under 12nm (or close to) process, we align all areas to 12nm. The SRAM area is estimated as $1.2mm^2/MB$.

Figure 2. Left: performance v.s. memory capacity of several industrial NPUs. Right: a summary of SRAM area ratio in these accelerators.

located next to the PE array to serve as the data interface and manage data between the PE array and the external memory (e.g., DRAM or other cores). Due to the limited capacity of the global buffer, the compiler has to partition the network execution into a series of elementary workloads that are scheduled along the parallel spatial resources and the temporal dimension [18, 61, 72]. The capacity of the global buffer usually dominates the external memory access and bandwidth requirements, significantly impacting system performance. If the global memory is larger, it is more likely to buffer more intermediate data and avoid data being evicted to DRAM. As shown in Figure 1, a larger buffer expands the scope of elementary workloads from a single layer to a larger subgraph, reducing the communication overhead.

However, choosing an appropriate memory specification is always a challenge. In Figure 2, we surveyed 16 popular industrial neural network processors with various memory/performance/area characteristics, where nine of them target the training domain [6, 11, 24, 34, 35, 40, 41, 48, 60, 63, 69] and seven target model inference [1, 7, 8, 26–28, 49, 65]. According to the survey, we can observe several trends as follows:

1. Memory occupies a significant portion of the silicon footprint on an NPU chip, ranging from 4% to 79% of the area, with capacities from 2.5MB to 896MB.
2. Figure 2 Left shows a trend of diminishing marginal benefit of memory capacity. This is because there is a critical capacity to meet the data reuse and bandwidth requirement at the beginning, and the increments become negligible with higher memory capacity.
3. We can infer that there is a saturated capacity equivalent to the ideal unlimited memory, especially for the inference design. For example, Hanguang [26] is a special SRAM-only inference system without DDR, and the 394MB buffers are large enough to hold the intermediate data in their scenarios.
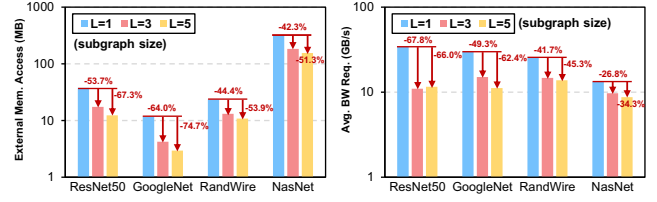

Figure 3. Evaluations on subgraphs fusing different number of layers (denoted as L=1,3,5). Y-axis is in the log domain. The 2TOPS NPU accelerator is configured with a 1MB global buffer and a 1.125MB weight buffer. The bandwidth requirement of weights is from the prefetch of the next subgraph, while that of activations is from the inputs and outputs of each subgraph.

This survey implies a design trade-off between memory capacity and performance based on workloads and commercial considerations. Motivated by the observations above, this paper aims to provide several memory design considerations and study the connection between workload features and memory capacity in an NPU accelerator.

2.2 Workload Deployment

A neural network is usually executed in a DNN accelerator with layer or graph granularities based on the buffer capacity and dataflow.

2.2.1 Layer-level Assignment. This manner assigns tasks layer by layer. Most previous studies employ a tiling-based layer-wise execution manner [10, 21, 30, 37, 50, 61], which elaborates the tiling sizes of tensors to fit in the accelerator buffers and maintain performance. A proper tiling scheme should overlap the data loading latency with the computing time of each tile and try to reduce the repeated access of local weight buffers. Tiles of data are transferred between the external memory and the global buffer, and PEs subsequently fetch data from the global to their local buffers. Given the larger bit-width of partial sums (e.g., 24bit partial sums v.s. 8bit inputs in Simba), the output-centric tiling scheme is more commonly used to calculate the final results before writing back to the global buffer [61].

2.2.2 Graph-level Assignment. Unlike the layer-level assignment that restrains from leveraging inter-layer reuse, a graph-level assignment processes several layers of a neural network as a whole. To demonstrate the effectiveness of the layer-level assignment, we evaluate four networks on a 2TOPS accelerator model, as shown in Figure 3. The results show that fusing layers into subgraphs significantly reduces external memory access by 42.3% ∼ 74.7% and average bandwidth requirements by 26.8% ∼ 67.8%. However, the improvements of larger subgraphs are marginal, indicating that there is an optimal trade-off between inter-layer

reuse and subgraph size, which determines the memory requirement. For example, executing three-layer subgraphs reduces external memory access by 53.7% in ResNet50, while executing five-layer subgraphs only further reduces it by 13.6%.

Several works have studied inter-layer reuse and graph partition. However, they have several limitations in terms of performance and flexibility. LCP [42] groups similar layers into a cluster and executes them as a whole, which makes it challenging to generalize into an arbitrary graph. Fused-CNN [4] and SR-CNN [38] fuse large contiguous layers for plain networks using manually-designed strategies. Irregular-NN [73] attempts to execute a complex subgraph using a DP-based algorithm, but the constrained search space limits the exploration.

To overcome these challenges, we propose an end-to-end framework that automatically optimizes the graph partition and memory configuration for any neural network. Our framework consists of two main components: a graph-level dataflow and a hardware-mapping co-exploration algorithm. We first introduce the graph-level dataflow and its hardware implementation. Then, we present Cocco, an efficient algorithm that explores the trade-offs among memory configurations and graph partition schemes based on workload features.

3 The Proposed Graph-Level Scheme

To execute layers on an NPU core in a graph-level manner, we need an effective approach to reuse intermediate data and decide the memory allocation. This section presents our comprehensive scheme for subgraph execution, which addresses the first two challenges mentioned in Section 1. First, we describe a multi-layer execution flow that minimizes the memory footprint by a friendly tiling approach (*for challenge 1*). Second, we explain how to implement this flow on a real NPU using an efficient data reuse pattern (*for challenge 2*). The consistent target is to reduce the memory footprint and be friendly to implementation.

3.1 Subgraph execution scheme

It is common practice for the layer-level scheduling to partition the output tensor into several tiles as layer-level elementary operations [56, 61, 72, 74], simplifying the scheduling and instruction generation. Likewise, our high-level idea is also to generate a series of explicit **subgraph-level elementary operations**. However, we need to address the challenges of various kernel sizes and strides in different paths to prevent unbalanced data production and unnecessary memory.

A model's subgraph consists of multiple layers (nodes) with dependencies. Section 4 provides detailed information on subgraph partition. In Figure 4(a), we present a straightforward **production-centric scheme** for executing a subgraph

with different kernel sizes in two branches, deriving tile sizes of the subsequent layers based on the predetermined input tile sizes. For example, we can produce a 1×1 tile of Node(0) and a 2×2 tile of Node(2) with a given 5×5 feature map of input Node(-1). In this case, these intermediate results only reduce to 1×1 in Node(3), limited by the smallest input of Node(0), so the remaining results of Node(2) can not be consumed immediately. As shown in Figure 4, three extra data of Node(2) along with sixteen extra source data of Node(1) take up extra memory space. There are more redundant cached data when the subgraph becomes larger and more complicated. Disadvantages of this manner are attributed to the production-centric idea that consumes all related activations from the producers at once.

To avoid the memory overhead of storing unused data, we propose a **consumption-centric scheme** in Figure 4(b), where results of each node are *produced on demand based on consumer(s)* (i.e., output node(s)). For example, given a 1×1 tile of Node(3), we derive the 1×1 tile size for Node(2), which subsequently decides a 3×3 tile for Node(1).

The backward-derivation for each producer node is non-trivial because of diverse kernel sizes and strides in different paths. Therefore, we propose a three-stage flow to determine the behavior of each node, as illustrated in Figure 5. The high-level idea is to let output nodes drive the whole execution and match the data consumption and production in each subgraph-level elementary operation.

The stage-1 is similar to the traditional single-layer scheduling, where the tile size is optimized for higher computation utilization. In order to hold a larger subgraph, the tile size
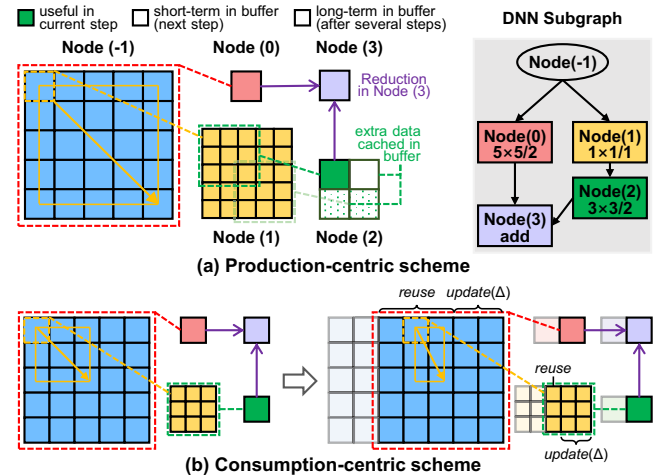

Figure 4. A conceptual comparison between two manners to process a subgraph. The node marked with a negative number represents the input node. The corresponding subgraph is shown in the upper right, where $F \times F/s$ refers to the convolution kernel size (F) and stride (s).
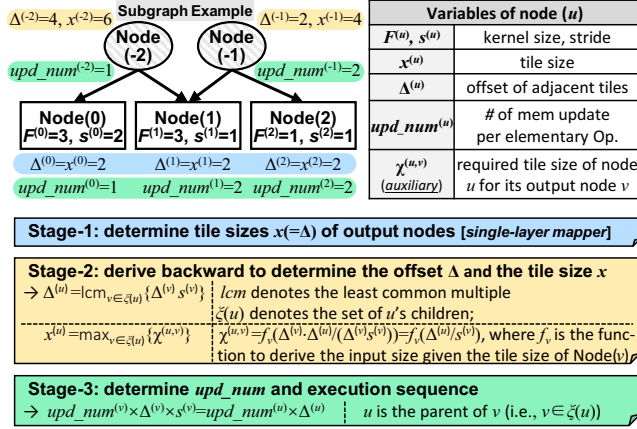
Figure 5. The flow to determine the execution scheme of a subgraph (i.e., the computed tile size of each node, the tile offset, and the processing sequence of nodes). For simplicity, we discuss the 1D-CONV in this example and it is similar in the 2D-CONV case.
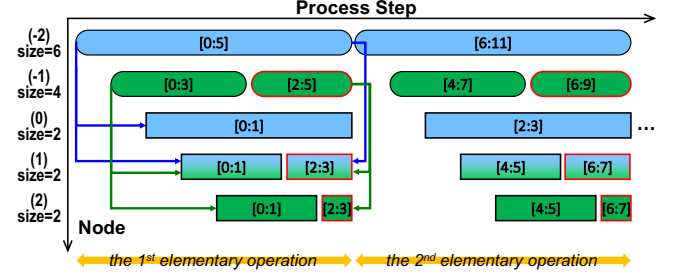

Figure 6. The memory snapshot during two subgraph elementary operations based on the execution scheme of Figure 5 example. The allocated memory size and update offset correspond to x and Δ, respectively (the $[m{:}n]$ notation denotes data ranging from index m to n). The arrows denote the data dependency according to the node relation in the subgraph.

tends to be smaller. In the 1D-CONV example, we set the tile size to be 2 for output nodes.

The stage-2 aims to determine the data update offset Δ and the memory allocation size x for each node based on the consumer(s), processing in the reverse topological order. We use the least common multiply (LCM) operation to determine $\Delta^{(u)}$ of producers for aligning different input offset requirements ($\Delta^{(v)}s^{(v)}$) from consumers. Hence, one producer update may correspond to multiple updates of a consumer. For example, $\Delta^{(-2)} = \text{lcm}\{\Delta^{(0)}s^{(0)}, \Delta^{(1)}s^{(1)}\} = 4 = 2\Delta^{(1)}s^{(1)}$, one update of Node(-2) corresponds to two updates of Node(1). As for the tile size deduction, $f_v(\Delta^{(u)}/s^{(v)})$ is to derive the required input tile size $\chi^{(u,v)}$ for output node v[1], where $\Delta^{(u)}/s^{(v)}$ is the consumer offset (updated data) per producer u update. The maximum result $\chi^{(u,v)}$ of all outputs v is the tile size $x^{(u)}$ of input node u. In this example, $x^{(-2)} = \max\{f_0(2), f_1(4)\} = 6$ and $x^{(-1)} = \max\{f_1(2), f_2(2)\} = 4$.

As mentioned above, since we use LCM to align production and consumption, one producer update may correspond to multiple updates of a consumer. In **the stage-3**, we use upd_num to represent the number of memory update per subgraph elementary operation. The generated result of the example in Figure 5 is shown in Figure 6. upd_num of Node(-1), Node(1), and Node(2) are two, where the second updates are highlighted in red boxes. Note that the $\{upd\_num^{(-2)}, \ldots, upd\_num^{(2)}\}$ solution is not unique, but the unique co-prime one $\{1, 2, 1, 2, 2\}$ corresponds to the minimal elementary operation.

The proposed flow is based on a general directed acyclic computation graph and is not limited to specific layer features. In this way, we can determine the execution scheme for any complex irregular network like NasNet [75] and RandWire [68].

3.2 Memory Management for the subgraph execution

Up to now, we have inferred the execution scheme for subgraphs, and the remaining challenge is how to implement it on hardware efficiently. Figure 7 shows the memory allocation and update scheme for the subgraph execution. Before computing a subgraph, the compiler determines logical blocks for input, intermediate, and output nodes, where the block sizes depend on the tile sizes derived from the execution flow.

For convenient management, we introduce two types of memory regions: MAIN and SIDE. The MAIN region stores the source data for PE (i.e., the tile of $P_0 \times Q_0 \times C$ in Figure 7). The SIDE region reserves the horizontally overlapping data[2]. Considering no reuse requirement for some output nodes, we only need a MAIN region to buffer the results of the current tile. Except for the input nodes (negative numbers) loading data from DRAM, the other nodes update data locally based on the computed results of the input node(s).

In detail, the update scheme leverages the collaboration between the MAIN region and the SIDE region to achieve full reuse across sliding tiles (we consider kernel size > stride). As shown in Figure 7, when the convolution windows slide across the feature maps, the vertical overlap data (e.g., column $q = 5$) are reused locally in the MAIN region. In contrast, the horizontally overlapping data (e.g., the first row of $q = 6 \sim 8$) are loaded from the SIDE region (path ①). Only a subset of data is replaced by the newly calculated results

---

[1]For example, assume node v is a convolution layer with kernel size $F^{(v)}$ and stride $s^{(v)}$, then $f_v(x) = F^{(v)} + (x - 1) \times s^{(v)}$.

[2]We assume the column is the inner loop while the row is the outer loop.

(marked in green). Besides, the bottom horizontal slices write new data to the SIDE region for the next row loop (path ②).

The extra hardware overhead for the proposed memory scheme is slight. Figure 8 presents our 12nm NPU core for the subgraph processing, with a buffer region manager to logically partition the global buffer to support contiguous layer processing. The buffer region manager is a $2N$-depth register file, where N determines the maximum subgraph size, and each entry pair indicates the start and the end address for each region. The area overhead is quite small, and in our test chip, the area ratio is only 0.18% with $N = 64$ and 272-byte size (17-bit address for the 1MB 64bit-width global buffer).

In summary, our high-level idea is to divide the buffer into logical blocks for different layers and try to reuse data for sliding convolution windows. The memory management approach can be compatible with an accelerator as long as it supports the data movement inside the on-chip memory and flexible data assignment for computing. Coupled with our subgraph execution scheme introduced before, intermediate outputs in the subgraph can avoid being recomputed. Only those layers required by other subgraphs are written back to DRAM for further reuse.

4 Memory Communication-Capacity Co-Exploration

The aforementioned hardware model enables arbitrary subgraph execution, but there is always limited buffer capacity
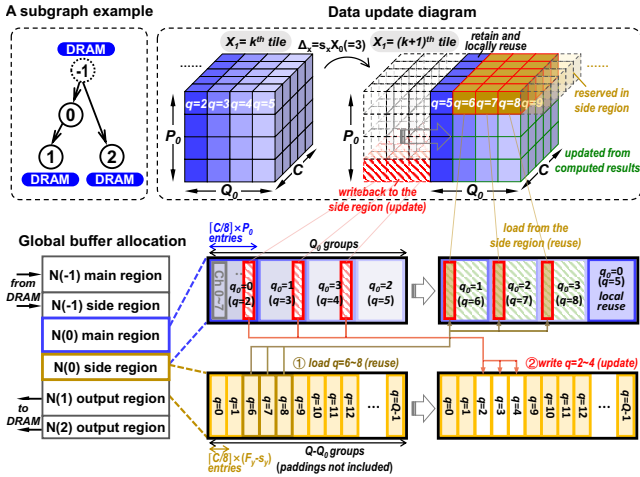
Figure 7. Memory allocation and data update scheme in the global buffer for full data reuse. The data layout used in our implementation is NWHC8c (aligned to 8 channels), which can be changed in another design. P_0 and Q_0 are the height and width of an input tile; C is the input channel size; q is the global width-dimension index of the input tensor; and q_0 is the width-dimension index of an input tile.

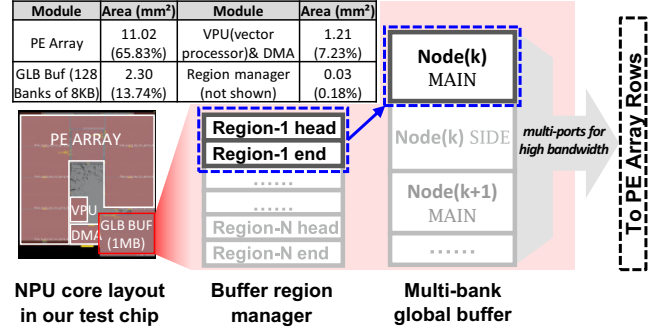| Module	Area (mm²)	Module	Area (mm²)
PE Array	11.02 (65.83%)	VPU(vector processor)& DMA	1.21 (7.23%)
GLB Buf (128 Banks of 8KB)	2.30 (13.74%)	Region manager (not shown)	0.03 (0.18%)

Figure 8. Hardware implementation with the buffer region manager in our 12nm NPU as a demonstration. The layout is an NPU core extracted from part of our in-house chip.

in hardware. Therefore, we need to partition the whole computation graph into a series of subgraphs that fit the memory. Below, we move up to the optimization for graph partition and memory design-space exploration for challenge 3.

4.1 Problem Formulation

4.1.1 Graph-Level Partition.
Formally, a DNN model can be represented as a *computation graph* $G = (V, E)$, where V is the vertex set consisting of all the layers in a DNN model, and E is the edge set that defines the structure of DNN. In particular, an edge $(u, v) \in E$ represents that the output of layer u is an input of layer v.

We aim to find a *partition scheme* $P : V \rightarrow \mathbb{N}$ that assigns each layer to a subgraph, where layer $v \in V$ is computed in the $P(v)$-th subgraph. A valid partition scheme should satisfy that any layer is computed before use. Therefore, for any $(u, v) \in E$, we have $P(u) \leq P(v)$. Moreover, any subgraph should be connected in G, otherwise meaningless.

We cast the partition exploration as an optimization problem. The objective is to find a valid partition scheme P that minimizes the total cost:

$$\sum_i Cost_M(\{v \in V \mid P(v) = i\}), \tag{1}$$

where $Cost_M$ is a cost function of a given subgraph based on a target metric M (e.g., external memory access (EMA) and energy). For each subgraph, the EMA cost contains the loading of weights and input activations and the storage of output activations[3]. The energy cost includes the overhead of EMA, on-chip buffers, and computation units.

4.1.2 Design-Space Exploration (DSE).
Our work further extends the optimization to combine with the memory design-space exploration. In this paper, we focus on the global buffer and the weight buffer, given that they dominate

---

[3]The nodes that are required to write-back to DRAM can be the model output layer or the layers required by the future subgraph.

the overhead of energy and area in an NPU core. As illustrated in Figure 1, a larger buffer capacity can take in more layers inside a subgraph, reducing communication costs but compromising the silicon area. To co-explore the hardware configuration and mapping, we construct an objective function by a linear combination of the hardware and mapping costs:

$$\text{BUF\_SIZE} + \alpha \cdot \sum_i Cost_M(\{v \in V \mid P(v) = i\}), \quad (2)$$

where α is a preference hyper-parameter to adjust the proportion between two costs.

4.2 Baseline Methods

Several optimization methods that exist today can perform graph-level partition. However, most of them fail to directly co-explore hardware and partition. Below, we list four typical methods as our baselines and sketch their features.

4.2.1 Enumeration-based Algorithm.
Fused-CNN [4] applies a straightforward way to enumerate all possible partition schemes and return the best one. Jangda *et al.* [25] proposed state compression dynamic programming to speed up the enumeration-based algorithm. We migrate their methods as our baseline and further improve them by only recording one subgraph in the state to reduce the time complexity.

Nonetheless, there are still exponential states in the improved implementation. Let N be the number of nodes in a graph, and the enumeration-based method may explore up to $O(2^{2N})$ states for irregular networks. Consequently, the search is hard to complete within a reasonable search time for large-scale networks, not to mention the co-exploration with DSE.

4.2.2 Greedy Algorithm.
Halide [47] employs a greedy algorithm to perform function grouping, which can be applied to the graph-level partition. Specifically, it first assigns each layer into a single-layer subgraph. Then it iteratively fuses a pair of subgraphs contributing the greatest benefit until all benefits are negative.

Therefore, this algorithm tends to be trapped at the local minimum. Moreover, since the fusion decision rules are based on a given hardware, the greedy method cannot co-explore with DSE.

4.2.3 Dynamic Programming (DP)-based Algorithm.
For the irregular network scheduling problem, Zheng *et al.* [73] proposed a DP-based algorithm. They arrange the layers based on their depth and perform DP in a sequential manner.

This method is restricted to assigning layers that are contiguous in the depth order into a subgraph, hence the exploration is confined to constrained search space. It is unlikely to find the global optimum, especially for non-plain network

structures. In addition, since the state transition of DP depends on the predefined buffer size, it is also tough to carry out co-exploration.

4.2.4 Simulated Annealing (SA).
SA [33] is a popular optimization algorithm that samples a point and updates it iteratively to improve. It adopts the new sample points with a probability affected by the performance difference and a hyper-parameter named temperature. We employ the customized mutation operations (described in Section 4.4.3) to update the sample points and implement an SA-based algorithm as a baseline.

SA is an alternative optimization method for our framework with compatible operators, but it is not stable as the genetic algorithm in a range of benchmarks, which will be shown in later experiments.

4.3 Genetic Algorithm

Previous research shows competitive performance of the Genetic Algorithm (GA) in several scheduling optimization problems [30, 31]. We summarize several benefits of GA for our hardware-mapping co-exploration problem:

1. **White-box property:** We can track and tune its optimization process conveniently. Therefore, it is easy and intuitive to understand.
2. **Complete search space:** It has the potential to explore the complete search space by customized mutation and crossover operations.
3. **Avoid local optima:** In contrast to the greedy algorithm, GA can naturally jump out of the local minimum benefiting from the diversity of the population.
4. **Flexible initialization:** We can use the results of other optimization algorithms to initialize GA and use GA to finetune the result.
5. **Co-exploration:** Through the proposed GA operations and genome encoding, it can further support partition-DSE co-exploration.

We encode each candidate solution (partition scheme and the corresponding memory configuration for our problem) as a *genome*, and the *population* contains a set of genomes. The GA goes through a series of *generations* to obtain a lower cost. It performs the *crossover* and *mutation* operations on the population in each generation. Specifically, a crossover operation blends two genomes selected from the population to generate one offspring while a mutation operation modifies a genome randomly. At the end of each generation, the evaluation environment evaluates the *fitness* of each genome, and the population in the new generation is selected based on the fitness results.

4.4 Cocco Optimization Framework

Cocco is a GA-based optimization framework that enables the co-exploration of memory configuration and graph-level partition, as shown in Figure 10. The core of Cocco is a
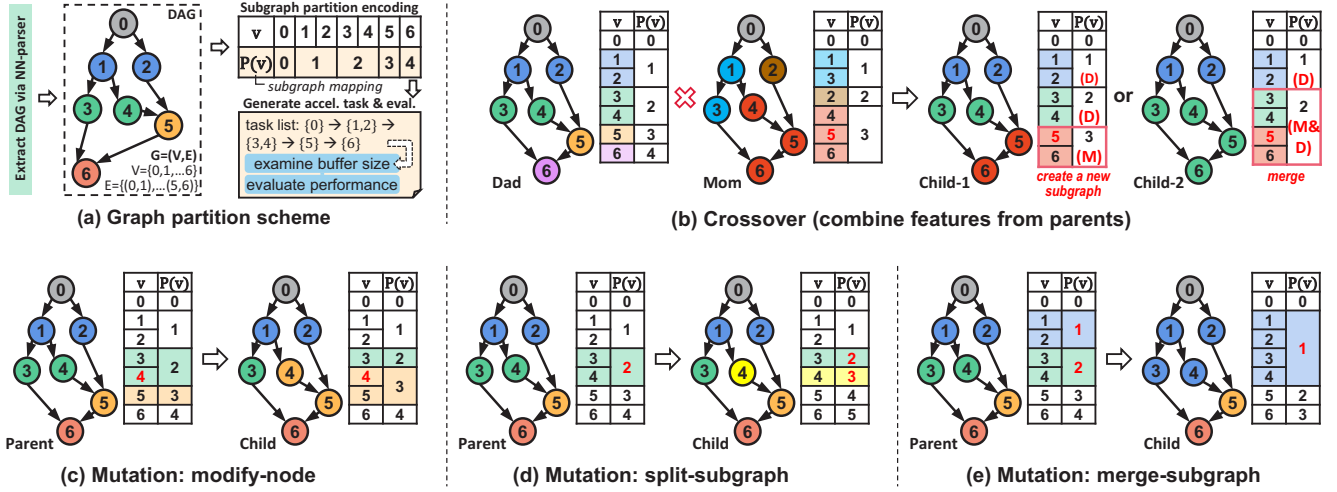
Figure 9. Illustration of crossover and mutation operations in Cocco.

series of operations that explore a complete search space. We build a genetic algorithm based on these customized operations. Fed with the neural network structure and DSE requirements, Cocco goes through several steps to get the optimization results. The execution model described in Section 3 is embedded in the evaluation environment. In the following, we introduce the five stages of Cocco.

4.4.1 Initialization. The first step in Cocco is to generate the initial population, where each genome contains a partition scheme of the computation graph and a memory configuration for DSE. For the DSE part, every genome selects a capacity value in a given range following a uniform distribution. There are two options in Cocco to initialize the partition scheme P of each genome. The first option is random initialization. Precisely, we determine the $P(v)$ for each layer $v \in V$ in topological order, and each $P(v)$ is selected randomly within the valid range. The other option is to initialize the partition scheme from other optimization algorithms.

4.4.2 Crossover. We designed a customized crossover operation to inherit and blend the features of two parents selected from the population. Specifically, each hardware configuration (i.e., memory capacity) in the offspring is the average of its parents and then rounds to the nearest candidate value. For the partition scheme, we assign layers to subgraphs in topological order. Each undecided layer chooses one parent randomly to reproduce the corresponding subgraph. If the reproduced subgraph contains layers that have been decided, we split out a new one excluding those layers, or merge it with one of the subgraphs to which the decided layers belong.

As shown in Figure 9(b), layer 1 and layer 3 select Dad as the parent to reproduce the subgraphs {1, 2} and {3, 4}, respectively. Next, layer 5 selects Mom as its parent, so it
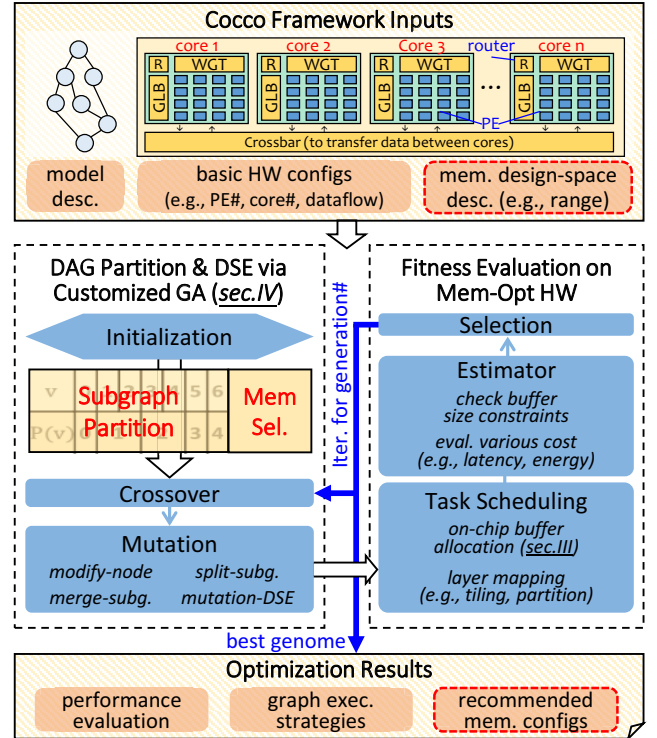
Figure 10. Cocco framework overview.

intends to reproduce subgraph {4, 5, 6}. However, since we have already decided on layer 4 in subgraph {3, 4}, there are two alternatives: creating a new subgraph {5, 6} (Child-1) or merging with subgraph {3, 4} to obtain {3, 4, 5, 6} (Child-2).

4.4.3 Mutation. Four mutation operations are customized for the optimization flow to explore the search space extensively. We guarantee the validity of genomes after each mutation in the implementation. At the bottom of Figure 9,

we show a node-level operation (`modify-node`) and two subgraph-level ones (`split-subgraph` and `merge-subgraph`):

- `modify-node` (Figure 9(c)): Modify the assignment of a randomly selected node u: from $u \rightarrow P(u)$ to $u \rightarrow P'(u)$, where $P'(u)$ can be an existed subgraph or a new one.
- `split-subgraph` (Figure 9(d)): Split a randomly selected subgraph into two or more subgraphs.
- `merge-subgraph` (Figure 9(e)): Merge two randomly selected subgraphs into one subgraph.
- `mutation-DSE` (not shown): Modify the memory configuration to a random one within the range. The new values are sampled based on a normal distribution, where the average is the original value, and the variance is a hyper-parameter.

4.4.4   Evaluation. Since GA tries to maximize the fitness of the genomes, we set fitness to be the opposite of the cost (e.g., Formula 1 and 2). To evaluate the fitness of each genome in the population, we use our simulator (introduced in the next section) to extract the execution costs of subgraphs (e.g., EMA and energy).

During the evaluation, the simulator decodes the subgraph and hardware configuration of each genome and calculates the fitness by aggregating the cost of each subgraph. Particularly, when a large subgraph exceeds the buffer capacity, we perform the `split-subgraph` operation to ensure genome validity. This kind of in-situ tuning can increase the number of valid samples during the optimization operations and thus, improve the sample efficiency.

4.4.5   Selection. At the end of each generation, Cocco performs the tournament selection. Specifically, it holds multiple tournaments among a few randomly selected genomes, and the winners (the genome with the best fitness) of these tournaments form the population of a new generation. This operation facilitates superior fitness in the new generation. The number of genomes in each tournament is decided by a hyper-parameter. The new generation subsequently starts from the crossover step again.

5   Experiments

In the evaluations, we first present the superiority of Cocco for the graph partition; and then demonstrate its outstanding stability and sample efficiency of the co-exploration for the hardware optimization, followed by additional discussions about the results under different configurations.

5.1   Methodology

5.1.1   Evaluated Models. In the following evaluations, we consider three types of model structures: plain (VGG16 [57]), multi-branch (ResNet50, ResNet152 [20], GoogleNet [59], Transformer [64], and GPT [52]), and irregular structure (RandWire-A/B [68] and NasNet [75]). RandWire-A/B are

generated based on the small and regular regime configurations introduced in the paper [68]. FC layers are transformed to 1×1 CONV while pooling and element-wise layers are analyzed as depth-wise CONV without weights. The scalar operations (e.g., activation function) are hidden in the pipeline (e.g., the post-process module following PE in Simba [56]) and their overhead can be ignored.

5.1.2   Accelerator Platform. As shown at the top of Figure 10, we consider a SIMBA-like hierarchical accelerator with a global buffer, a weight buffer, and a 4×4 PE array in each core used in several previous works [56, 61, 71]. Each PE contains an 8×8 MAC array to process a sub-tile from the global buffer. In particular, we model the execution flow based on the scheme described in Section 3. The parallelism of two dimensions of the PE array can be dynamically configured by the mapper results to ensure high utilization. We schedule subgraphs in topological order and prefetch weights of the next subgraph during the current computing. We also extend our platform to support fundamental multi-core studies by interconnecting cores with a crossbar. They share weights to release the burden of each core.

The arithmetic and memory overhead is extracted in a 12nm library based on the synthesized RTL implementations (SRAM based on the ARM memory compiler) with 1GHz. The DRAM energy is set as 12.5pJ/bit [70]. The extra footprint of the plug-in design is mainly a 272-Byte register file to store the head and end logical region addresses of maximal 64 nodes, which is negligible. Based on off-the-shelf evaluators Timeloop [50] and MAESTRO [37] for spatial accelerators, we developed a modified simulator that supports the evaluation of latency and energy. It employs the consumption-centric scheme to determine the tile size of each layer, and the memory access in the model is free from padding data. The latency per subgraph depends on the maximum of the calculation and external communication cycles. We allocate 16GB/s DRAM bandwidth per accelerator core for loading weights and input activations and writing back data for subsequent subgraphs. The off-chip communication consists of weight loading of each layer and the inputs and outputs of each subgraph. As described in Section 3, our subgraph execution scheme avoids recomputing of intermediate outputs.

5.1.3   Baselines. Three optimization baselines for graph partition are the greedy algorithm used in Halide [47], dynamic programming (DP) used in Irregular-NN [73] , and the enumeration-based method as a reference.

For the DSE studies, we compare Cocco with simulated annealing (SA) [33] to demonstrate the better stability of GA. These two methods are both the co-optimization scheme that optimizes partition and hardware settings at the same time. In contrast to co-optimization, the two-step scheme is another method for design-space exploration. Specifically, we
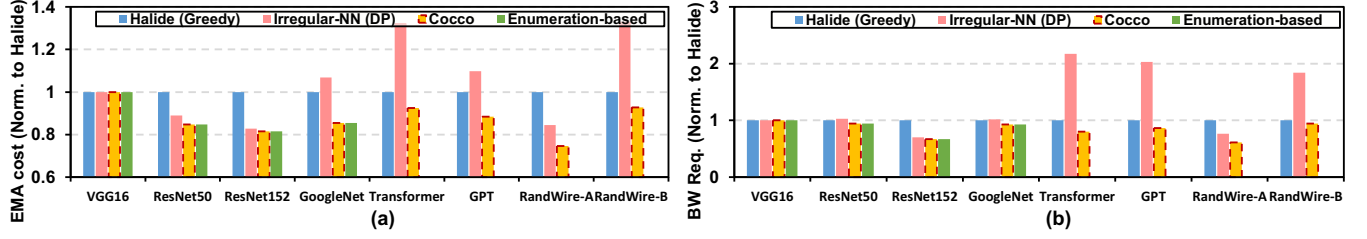
Figure 11. The evaluation results for graph partition using the EMA-opt configuration (EMA as the optimization metric). The enumeration-based method is deterministic, which figures out the optimal solution as a reference in the first four models. It cannot complete for large-scale models (Transformer, GPT, RandWire-A, and RandWire-B) in a reasonable time because of the exponential search space.

use random search (RS) or grid search (GS) to sample memory capacity candidates and then explore the corresponding partition schemes. During the search, we evaluate 5,000 samples for each capacity candidate and keep the best candidate as the output. As for the sampling method, RS randomly samples memory capacity candidates while GS uses a coarser granularity to enumerate the candidates.

5.2 Graph Partition Evaluations

We start by presenting the partition performance on the single-core hardware with a 1MB global buffer and a 1.125MB weight buffer. The number of samples in Cocco is set to be 400,000. We evaluate the external memory access (EMA) and bandwidth requirements of eight models shown in Figure 11, where the results are normalized to the Halide baseline. This experiment aims to validate the effectiveness of our Cocco framework in graph partition. For networks with simpler structures, Cocco can find out the optimal solutions same as the enumeration-based results. For large-scale irregular networks (Transformer, GPT, RandWire-A, and RandWire-B), the enumeration-based method cannot complete in a reasonable time, while Cocco provides better solutions than Halide and DP. A better subgraph partition strategy helps to ease the communication burden, thus reducing the EMA cost and bandwidth requirements.

5.3 Hardware-Mapping Co-Exploration

After learning the superiority of Cocco for the graph partition, we further co-explore the memory configuration and graph partition mapping as the core study of this work. Three categories of exploration methods are used, including the *fixed hardware scheme*, the *two-step scheme* as baselines, and the proposed *co-optimization scheme*. We set three fixed memory configurations with Small capacity, Medium capacity, and Large capacity, followed by a partition-only procedure. The two-step scheme is implemented with decoupled steps for capacity search (RS or GS) and partition (GA). The co-optimization methods include the proposed Cocco and an SA-based one as the comparison. All methods sample up to

Table 1. Hardware-mapping co-exploration for separate buffer. In this table, A refers to the global buffer, and W refers to the weight buffer. We evaluate the cost using Formula 2 (the lower cost, the better), where the metric M is energy. We use RandWire-A as RandWire in the following experiments.

Optimization		ResNet50			GoogleNet		
		Size (A)	Size (W)	Cost	Size (A)	Size (W)	Cost
Fixed HW	**Buf(S)**	512KB	576KB	**1.04E7**	512KB	576KB	4.07E6
	Buf(M)	1024KB	1152KB	1.07E7	1024KB	1152KB	5.06E6
	Buf(L)	2048KB	2304KB	1.24E7	2048KB	2304KB	7.18E6
Two-Step	**RS+GA**	448KB	864KB	**1.04E7**	384KB	432KB	3.88E6
	GS+GA	128KB	864KB	1.07E7	128KB	144KB	3.80E6
Co-Opt	**SA**	256KB	360KB	1.06E7	192KB	144KB	3.78E6
	Cocco	704KB	864KB	**1.04E7**	192KB	432KB	3.75E6
Optimization		RandWire			NasNet		
		Size (A)	Size (W)	Cost	Size (A)	Size (W)	Cost
Fixed HW	**Buf(S)**	512KB	576KB	3.23E6	512KB	576KB	6.14E7
	Buf(M)	1024KB	1152KB	3.92E6	1024KB	1152KB	5.83E7
	Buf(L)	2048KB	2304KB	6.00E6	2048KB	2304KB	5.66E7
Two-Step	**RS+GA**	448KB	792KB	3.31E6	1152KB	2016KB	5.60E7
	GS+GA	128KB	144KB	3.02E6	2048KB	2304KB	5.66E7
Co-Opt	**SA**	192KB	144KB	3.00E6	2048KB	1872KB	5.61E7
	Cocco	256KB	144KB	**2.98E6**	1280KB	2088KB	5.59E7

50,000 points. The energy-capacity co-optimization is used in the following evaluations.

5.3.1 DSE analysis using separate and shared buffer.

We first perform the hardware-mapping co-exploration to determine the suitable memory configuration (except for the fixed-HW scheme) with $\alpha = 0.002$[4] and then solely execute the partition-only Cocco to obtain the final cost. In particular, we also compared the results using two memory designs: separate buffer and shared buffer. For the separate buffer design, activations and weights are stored in different buffers while they share the same space in the shared buffer design. The memory capacity candidates for the global buffer

---

[4]The energy and the capacity units are pJ and Byte, respectively.

Table 2. Hardware-mapping co-exploration for shared buffer. We evaluate the cost using Formula 2 (the lower cost, the better), where the metric M is energy.

Optimization		ResNet50		GoogleNet	
		Size	Cost	Size	Cost
Fixed HW	Buf(S)	576KB	1.01E7	576KB	3.66E6
	Buf(M)	1152KB	1.00E7	1152KB	4.04E6
	Buf(L)	2304KB	1.04E7	2304KB	5.09E6
Two-Step	RS+GA	1280KB	**0.98E7**	640KB	3.65E6
	GS+GA	1344KB	**0.98E7**	512KB	3.65E6
Co-Opt	SA	896KB	1.00E7	192KB	3.75E6
	Cocco	1344KB	0.98E7	384KB	**3.60E6**

Optimization		RandWire		NasNet	
		Size	Cost	Size	Cost
Fixed HW	Buf(S)	576KB	**2.83E6**	576KB	6.36E7
	Buf(M)	1152KB	3.03E6	1152KB	5.73E7
	Buf(L)	2304KB	3.90E6	2304KB	5.51E7
Two-Step	RS+GA	320KB	2.85E6	2560KB	5.47E7
	GS+GA	832KB	2.86E6	3072KB	5.42E7
Co-Opt	SA	256KB	2.92E6	1728KB	5.56E7
	Cocco	384KB	**2.83E6**	2624KB	**5.37E7**

(for activations) range from 128KB to 2048KB with a 64KB interval, while that for the weight buffer range from 144KB to 2304KB with a 72KB interval. The exploration range of the shared buffer is from 128KB to 3072KB with an interval of 64KB.

The evaluation using separate buffers is shown in Table 1, where Cocco achieves better optimization with up to 1.89% (compared to SA in ResNet50) to 50.33% (compared to Fixed-HW(L) in RandWire) lower cost compared to various baselines across four models. The two-step scheme fails to combine the information between different sizes, so it is generally worse than the co-optimization method.

The capacity results also reflect the inherent capacity preference of different models. The data amount in GoogleNet and RandWire is relatively smaller, and thus their capacity requirements are lower. In contrast, the data amount in NasNet is larger, so a high capacity is preferred.

As shown in Table 2, the evaluation of the shared buffer setting shows a similar trend. Furthermore, we can observe that most of the cost results of the shared buffer are lower than those using the separate configuration. Although the shared buffer design requires additional control flows, it indeed improves efficiency than the separate buffer design.

5.3.2 Sample efficiency analysis. We next study the sample efficiency of the two-step and the co-optimization scheme in Figure 12. We record the cost trends of the first 50,000 samples on ResNet50, GoogleNet, and RandWire during the exploration. Overall, Cocco shows a consistent convergence trend on these three networks. And it converges faster and
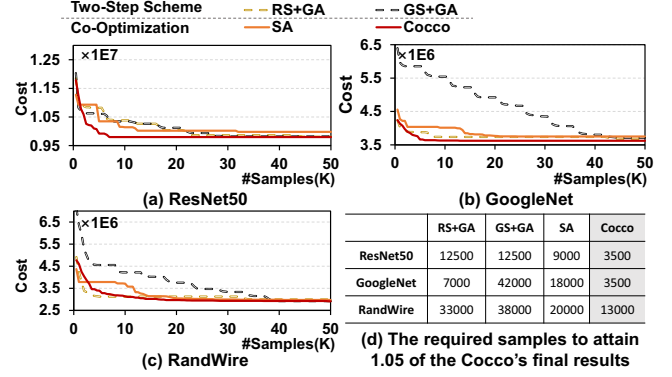

Figure 12. The convergence curve of Cocco compared with other baselines in the hardware-mapping co-explorations. The optimization method requiring fewer samples in (d) has higher sample efficiency.
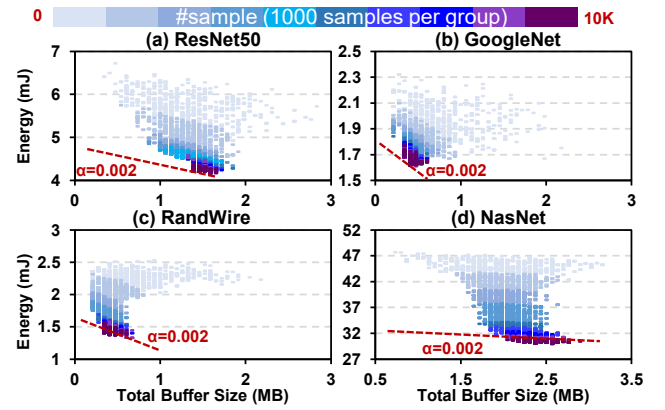

Figure 13. The visualization of sample points distribution during optimization. The slope of the red dashed line denotes the preference between energy and capacity cost. The point on the line with a lower intercept has a smaller cost.

achieves lower costs compared to other baselines, exhibiting a higher sample efficiency. The two-step methods perform graph-partition separately under different capacities, so they fail to utilize the partition information between capacities. Particularly, the GS method uses a deterministic search direction (search from large to small capacity in this experiment), so the convergence time depends on the optimal capacity. Since GoogleNet and RandWire require relatively small buffers, GS takes a considerable number of samples to converge.

5.3.3 Optimization procedure analysis. We next study how the distribution of sample points changes during the optimization procedure of Cocco. While searching for 20 generations with 500 genomes each, we divided them into ten groups with different colors in Figure 13. The results show that the distribution moves towards a lower intercept
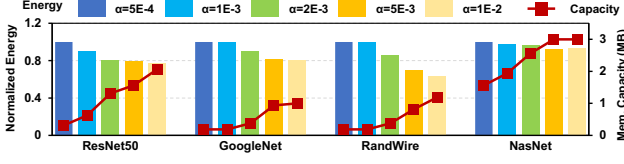
Figure 14. The trade-off between energy and memory capacity. The optimization target is to minimize the cost defined in Formula 2, where the metric M is energy. Energy results of each model are normalized to the first $\alpha(= 0.0005)$ results.

Table 3. Multi-core and batch evaluation using the energy-capacity co-opt configuration. Size denotes the shared buffer size in each core.

Core#	Batch	ResNet50			GoogleNet		
		Energy(mJ)	Lat.(ms)	Size(KB)	Energy(mJ)	Lat.(ms)	Size(KB)
1	1	4.21	4.59	1344	1.61	2.05	384
	2	6.32	8.98	1728	2.18	3.91	896
	8	11.88	35.93	2880	5.64	15.53	1472
2	1	4.38	2.48	768	1.66	1.04	192
	2	6.46	4.78	1088	2.34	1.99	384
	8	13.01	19.12	1664	5.84	7.97	960
4	1	4.29	1.39	448	1.34	0.54	192
	2	6.58	2.68	640	2.20	1.07	192
	8	11.50	10.71	1664	6.24	4.30	448

Core#	Batch	RandWire			NasNet		
		Energy(mJ)	Lat.(ms)	Size(KB)	Energy(mJ)	Lat.(ms)	Size(KB)
1	1	1.26	1.47	384	28.57	49.92	2624
	2	2.25	2.74	704	47.68	99.87	3072
	8	8.66	10.85	1664	133.03	396.90	3072
2	1	1.41	0.95	192	29.18	24.93	1728
	2	2.37	1.80	384	48.80	49.73	2624
	8	8.39	7.16	1280	153.25	227.19	3072
4	1	1.39	0.71	192	28.00	14.56	960
	2	2.91	1.40	192	45.03	28.58	1664
	8	9.24	5.55	960	131.98	133.38	2816

of the α-slope line and gets more centralized in the later generations during the optimization process of Cocco.

5.4 Sensitivity Study about Cocco framework

5.4.1 Study of α in the cost function. The results shown in Figure 14 demonstrate the effectiveness of α in adjusting the preference between the memory capacity and the given metric (energy is used here). The optimization trades the memory capacity for lower energy cost with the increase of α. In addition, a larger memory capacity indeed contributes to lower energy, but the yields show differences because of their various model-inherent graph and layer patterns. For example, NasNet is more memory-intensive and more structure-complex than the other three models, so it requires a larger memory capacity for less energy consumption.

5.4.2 Study of performance v.s. memory capacity. Figure 2 shows that the increase of capacity is sub-linear with

performance. To study this observation, we scale our model to the multi-core version and share weights of a subgraph across cores. Different cores only buffer a subset of weights and transfer the data between cores, similar to BSD in Tangram [18] or data-rotation in NN-Baton [61]. The overhead of the interconnection crossbar is extracted from the implemented Arteries IP [5].

An accelerator with more cores can cover a larger subgraph but bring more core-to-core overhead. As shown in Table 3, in most cases, energy increases from the single-core to dual-core configuration because of the communication overhead. Moreover, profiting from the data-sharing mechanism, the required memory of each core drops with the increase of core number.

5.4.3 Batch size study. For the batch size evaluation shown in Table 3, the latency with a larger batch size principally presents a sub-linear increase, which benefits from the lower bandwidth requirement of weights via the inter-sample data reuse. In addition, such data reuse amortizes the energy burden per batch processing. And owing to the better weight reuse in multi-batch processing, a larger batch size does not require a proportional capacity.

6 Related Works

6.1 Intra-layer Optimization

Prior works focus on the data reuse for intra-layer assignments, like output-stationary in ShiDianNao [14] and Envision [46], weight-stationary in NeuFlow [15] and Nvdla [49], input-stationary in SCNN [51], and row-stationary in Eyeriss [13]. Based on these primitive dataflow patterns, extensive studies explored the optimal tiling and reordering schemes via brute-force, feedback-based, and constraint optimization approaches [23, 30, 50]. These works focus on layer-level optimization, missing the graph information at a higher level. The efficiency of tile updates depends on the memory architecture. Simba [56, 74] and NN-Baton [61] view each tile as an independent workload so that the tile size has a prominent impact on memory access due to halo regions. Motivated by traditional vision processors, Ascend [40] and DRQ [58] employ line buffers to achieve data reuse in the row direction, but the line buffer cannot well support the 2D-tiling reuse in both row and column directions.

6.2 Inter-layer Optimization

Intra-layer scheduling is sub-optimal, which is limited by the data reuse within a layer. Therefore, Fused-CNN [4], SR-CNN [38], and LCP [42] introduce layer fusion method that cache intermediate data on-chip to reduce data transfer overhead using handcrafted or heuristic methods for fusion partition. Although Irregular-NN [73] suggests a customized-DP algorithm, the exploration space is constrained because the layers in an assignment need to be successive in a specific

order. A recent work named DNNFuser [29] employs an RL-based method, but their formulation towards 1D layer-fusion is hard to handle complex irregular networks. Tangram [18] and Atomic [72] schedule DNN workloads on a multi-core (scalable) accelerator, but they focus on executing a single layer on each core at a time rather than processing multiple layers with local data reuse. Also, some previous works [2, 19, 62] tackle the workload placement problem for multiple devides without discussing the downstream execution on each device.

Cocco proposes an automatic framework for inter-layer scheduling with a comprehensive memory scheme. It focuses on the fundamental core-level temporal execution that can be potentially scaled up to the multi-core or multi-device scenario with a spatial parallelism mechanism.

6.3 Design-Space Exploration for Memory

Memory design exploration methods lie primarily on two sides: analysis-driven and search-driven. For the analysis-driven method, Chen *et al.* [12] leverage red-blue pebble models to derive the proper memory capacity representations. Subsequently, Cai *et al.* [9] propose Olympus, which generalizes a framework to a batch of successive layers and also fills up with more scheduling and data reuse techniques. However, they are difficult to represent a subgraph with complex inter-layer connections. As for the search-driven method, Xiao *et al.* [67], Kwon *et al.* [37], and Feng *et al.* [16] explore the memory configuration for the layer-level assignment using the brute-force search, while Kao *et al.* [32] employ a genetic algorithm to improve the efficiency. These works principally focus on the layer-level information, while in comparison, Cocco exploits graph-level features for the better optimization.

7 Conclusion

While layer-level scheduling is widely studied to improve memory efficiency, graph-level optimization remains relatively unexplored. This paper proposed a graph-level dataflow with the corresponding memory management scheme that enables flexible graph partitions with high memory utilization. On top of it, we propose Cocco, a framework to provide a recommended memory configuration with graph-level scheduling strategies. Cocco shows outstanding graph partition ability compared to the greedy algorithm and DP employed in previous works and enables efficient graph-level hardware-mapping co-exploration. This paper helps to provide an implementation philosophy for the accelerator memory and better deployment for it.

Acknowledgments

References

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 145–158.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). OpenReview.net, Vancouver, BC, Canada, 3983–3993.

[3] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. In *Proceedings of Machine Learning and Systems (MLSys)*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org, Austin, TX, USA, 1–14.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter A. Milder. 2016. Fused-layer CNN accelerators. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Taipei, Taiwan, 22:1–22:12.

[5] Arteries. 2022. Arteries IP Homepage. https://www.arteris.com.

[6] Ljubisa Bajic and Jasmina Vasiljevic. 2020. Compute substrate for Software 2.0. In *Proceedings of the IEEE Hot Chips 32 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–31.

[7] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and Redundancy Solution for the Full Self-Driving Computer. In *Proceedings of the IEEE Hot Chips 31 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–22.

[8] John Burgess. 2019. RTX ON - The NVIDIA TURING GPU. In *Proceedings of the IEEE Hot Chips 31 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–27.

[9] Xuyi Cai, Ying Wang, Kaijie Tu, Chengsi Gao, and Lei Zhang. 2022. Olympus: Reaching Memory-Optimality on DNN Processors. *IEEE Transactions on Computers (TC)* 71, 8 (2022), 1939–1951.

[10] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2022. Marvel: A Data-Centric Approach for Mapping Deep Learning Operators on Spatial Accelerators. *ACM Transactions on Architecture and Code Optimization* 19, 1 (2022), 6:1–6:26.

[11] Karam Chatha. 2021. Qualcomm® Cloud Al-100: 12TOPS/W Scalable, High Performance and Low Latency Deep Learning Inference Accelerator. In *Proceedings of the IEEE Hot Chips 33 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–19.

[12] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication Lower Bound in Convolution Accelerators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, San Diego, CA, USA, 529–541.

[13] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Seoul,

South Korea, 367–379.

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. ACM, Portland, OR, USA, 92–104.

[15] Clément Farabet, Berin Martini, B. Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. IEEE Computer Society, Colorado Springs, CO, USA, 109–116.

[16] Kaijie Feng, Xiaoya Fan, Jianfeng An, Xiping Wang, Kaiyue Di, Jiangfei Li, Minghao Lu, and Chuxi Li. 2021. ERDSE: efficient reinforcement learning based design space exploration method for CNN accelerator on resource limited platform. *Graphics and Visual Computing* 4 (2021), 1–11.

[17] Ken-ichi Funahashi. 1989. On the approximate realization of continuous mappings by neural networks. *Neural Networks* 2, 3 (1989), 183–192.

[18] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Providence, RI, USA, 807–820.

[19] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, Stockholm, Sweden, 1662–1670.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Sun Jian. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Las Vegas, NV, USA, 770–778.

[21] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, Virtual Event, USA, 943–958.

[22] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 5 (1989), 359–366.

[23] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 554–566.

[24] Drago Ignjatovic, Daniel W. Bailey, and Ljubisa Bajic. 2022. The Wormhole AI Training Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, San Francisco, CA, USA, 356–358.

[25] Abhinav Jangda and Uday Bondhugula. 2018. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Andreas Krall and Thomas R. Gross (Eds.). ACM, Vienna, Austria, 261–275.

[26] Yang Jiao, Liang Han, Rong Jin, Yi-Jung Su, Chiente Ho, Li Yin, Yun Li, Long Chen, Zhen Chen, Lu Liu, Zhuyu He, Yu Yan, Jun He, Jun Mao, Xiaotao Zai, Xuejun Wu, Yongquan Zhou, Mingqiu Gu, Guocai Zhu, Rong Zhong, Wenyuan Lee, Ping Chen, Yiping Chen, Weiliang Li, Deyu Xiao, Qing Yan, Mingyuan Zhuang, Jiejun Chen, Yun Tian, Yingzi Lin, Wei Wu, Hao Li, and Zesheng Dou. 2020. A 12nm Programmable Convolution-Efficient Neural-Processing-Unit Chip Achieving 825TOPS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, San Francisco, CA, USA, 136–140.

[27] Yang Jiao, Liang Han, and Xin Long. 2020. Hanguang 800 NPU - The Ultimate AI Inference Solution for Data Centers. In *Proceedings of the IEEE Hot Chips 32 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–29.

[28] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter C. Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David A. Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1–14.

[29] Sheng-Chun Kao, Xiaoyu Huang, and Tushar Krishna. 2022. DNNFuser: Generative Pre-Trained Transformer as a Generalized Mapper for Layer Fusion in DNN Accelerators. *arXiv preprint arXiv:2201.11218* abs/2201.11218 (2022), 1–8.

[30] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, San Diego, CA, USA, 44:1–44:9.

[31] Sheng-Chun Kao and Tushar Krishna. 2022. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores. In *IEEE International Symposium on High-Performance Computer Architecture, (HPCA)*. IEEE, Seoul, South Korea, 814–830.

[32] Sheng-Chun Kao, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. 2022. DiGamma: Domain-aware Genetic Algorithm for HW-Mapping Co-optimization for DNN Accelerators. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Cristiana Bolchini, Ingrid Verbauwhede, and Ioana Vatajelu (Eds.). IEEE, Antwerp, Belgium, 232–237.

[33] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. 1983. Optimization by Simmulated Annealing. *Sci.* 220, 4598 (1983), 671–680.

[34] Simon Knowles. 2017. Scalable Silicon Compute. In *Workshop on Deep Learning At Supercomputer Scale, NIPS*. OpenReview.net, Long Beach, CA, USA, 1–22.

[35] Simon Knowles. 2021. Graphcore. In *Proceedings of the IEEE Hot Chips 33 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–25.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., Lake Tahoe, Nevada, United States, 1106–1114.

[37] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, Columbus, OH, USA, 754–768.

[38] Juhyoung Lee, Dongjoo Shin, Jinsu Lee, Jinmook Lee, Sanghoon Kang, and Hoi-Jun Yoo. 2019. A Full HD 60 fps CNN Super Resolution Processor with Selective Caching based Layer Fusion for Mobile Devices. In *Proceedings of the Symposium on VLSI Circuits*. IEEE, Kyoto, Japan, 302–303.

[39] Grzegorz Lewicki and Giuseppe Marino. 2004. Approximation of functions of finite variation by superpositions of a Sigmoidal function. *Appl. Math. Lett.* 17, 10 (2004), 1147–1152.

[40] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture, HPCA*. IEEE, Seoul, South Korea, 789–801.

[41] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *Proceedings*

of the IEEE Hot Chips 31 Symposium (HCS). IEEE, Cupertino, CA, USA, 1–44.

[42] Xinhan Lin, Shouyi Yin, Fengbin Tu, Leibo Liu, Xiangyu Li, and Shaojun Wei. 2018. LCP: a layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*. ACM, San Francisco, CA, USA, 16:1–16:6.

[43] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Austin, TX, USA, 553–564.

[44] Yufei Ma, Yu Cao, Sarma B. K. Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Monterey, CA, USA, 45–54.

[45] Marvin Minsky and Seymour Papert. 1987. *Perceptrons - an introduction to computational geometry*. MIT Press, .

[46] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 2017. Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, San Francisco, CA, USA, 246–247.

[47] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (2016), 83:1–83:11.

[48] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David A. Patterson. 2020. Google's Training Chips Revealed: TPUv2 and TPUv3. In *Proceedings of the IEEE Hot Chips 32 Symposium (HCS)*. IEEE, Palo Alto, CA, USA, 1–70.

[49] NVIDIA. 2018. THE NVIDIA DEEP LEARNING ACCELERATOR. In *Proceedings of the IEEE Hot Chips 30 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–18.

[50] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel S. Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Madison, WI, USA, 304–315.

[51] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. ACM, Toronto, ON, Canada, 27–40.

[52] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. In *Preprint*. OpenAI, , 1–12.

[53] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the 33rd Conference on Artificial Intelligence (AAAI)*. AAAI Press, Honolulu, Hawaii, USA, 4780–4789.

[54] Frank Rosenblatt. 1957. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, .

[55] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Computer Vision Foundation / IEEE Computer Society, Salt Lake City, UT, USA, 4510–4520.

[56] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, Columbus, OH, USA, 14–27.

[57] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*. Computational and Biological Learning Society, San Diego, CA, USA, 1–14.

[58] Zhuoran Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. 2020. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1010–1021.

[59] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Boston, MA, USA, 1–9.

[60] Emil Talpes, Douglas Williams, and Debjit Das Sarma. 2022. DOJO: The Microarchitecture of Tesla's Exa-Scale Computer. In *Proceedings of the IEEE Hot Chips 34 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–28.

[61] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. 2021. NN-Baton: DNN Workload Orchestration and Chiplet Granularity Exploration for Multichip Accelerators. In *Proceedings of the IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1013–1026.

[62] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient Algorithms for Device Placement of DNN Graph Operators. In *Advances in Neural Information Processing Systems (NeurIPS)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). OpenReview.net, Virtual, 1–13.

[63] Tenstorrent. 2021. Grayskull. https://tenstorrent.com/grayskull/.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems (NIPS)*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). OpenReview.net, Long Beach, CA, USA, 5998–6008.

[65] Ofri Wechsler, Michael Behar, and Bharat Daga. 2019. Spring Hill (NNP-I 1000) Intel's Data Center Inference Chip. In *Proceedings of the IEEE Hot Chips 31 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–12.

[66] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 268–281.

[67] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1055–1068.

[68] Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. 2019. Exploring Randomly Wired Neural Networks for Image Recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Seoul, South Korea, 1284–1293.

[69] Andrew Yang. 2019. Deep Learning Training At Scale Spring Crest Deep Learning Accelerator (Intel® Nervana™ NNP-T). In *Proceedings of the IEEE Hot Chips 31 Symposium (HCS)*. IEEE, Cupertino, CA, USA,

1–20.

[70] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Lausanne, Switzerland, 369–383.

[71] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, New York, USA, 874–887.

[72] Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2022. Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, South Korea, 475–489.

[73] Shixuan Zheng, Xianjue Zhang, Daoli Ou, Shibin Tang, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2020. Efficient Scheduling of Irregular Network Structures on CNN Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 11 (2020), 3408–3419.

[74] Brian Zimmer, Rangharajan Venkatesan, Yakun Sophia Shao, Jason Clemons, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Ross Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Stephen W. Keckler, and Brucek Khailany. 2019. A 0.11 pJ/Op, 0.32-128 TOPS, Scalable Multi-Chip-Module-based Deep Neural Network Accelerator with Ground-Reference Signaling in 16nm. In *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI)*. IEEE, Kyoto, Japan, 300.

[75] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, (CVPR)*. Computer Vision Foundation / IEEE Computer Society, Salt Lake City, UT, USA, 8697–8710.