PCGC: A performance compact graph compiler based on multilevel fusion-splitting rules

Dong Dong
Beihang University
Hongxu Jiang (jianghx@buaa.edu.cn)
Beihang University
Hanqun Lin
Beihang University
Yanfei Song
Beihang University

Research Article

Keywords: DNN Graph Compilers, Subgraph Splitting, Partial Dynamic Tuning, Multilevel Fusion Rules, Edge Computing

Posted Date: December 12th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-2348223/v1

Abstract

The existing deep learning compilers are unable to perform efficient hardware performance-related graph fusion when both time and power consumption are considered. The operator optimization cost is too high because of excessive fusion or skipping fusion. In addition, the compilers optimize the computational graph of Deep Neural Networks (DNN) by performing static graph transformation based on the greedy algorithm, only considering the runtime performance and ignoring the cost of the tuning process. To solve these problems, this paper proposes PCGC, a DNN computational graph optimization compiler. Through the performance feedback at runtime, PCGC designs a computational graph fusion and splitting optimization strategy based on multilevel operator layer fusion-splitting rules. First, PCGC uses a rule-guided graph segmentation algorithm to recursively segment the computational graph into smaller subgraph to achieve an efficient and detailed search. Then, PCGC uses the cost model to receive the feedback of hardware performance information, uses the cost model and operator fusion rules to guide the partial fusion and split of the nodes and edges of the computational graph, and flexibly generates the optimal subgraph according to different hardware. Finally, in the fusion process, the operator computing attributes are considered. The graph-level node and operator-level cyclic fusion are closely combined to optimize the search space of partial fusion. Compared with other advanced accelerators, PCGC optimizes the overall power consumption on an embedded GPU by an average of 130.5% when the time consumption on each hardware is not lower than the average time consumption. On Domain Specific Architecture (DSA), PCGC optimizes power consumption by an average of 66.5%. On FPGA, PCGC optimizes power consumption by 66.1%. In a sense, PCGC can achieve high-speed inference in specific power supply scenarios, reducing the carbon emissions of edge computing.

1 Introduction

In the past decade, the rapid development of deep learning technology has brought a burden to its compilation and deployment on actual operating devices. On the one hand, the continuous expansion of the scale of DNN leads to the rapid growth of its requirements for hardware computing power. It often needs to run on a variety of general and special hardware, such as different types of CPU, GPU, FPGA, DSA [1], etc. On the other hand, the application scenarios of deep learning are more and more extensive, and the requirements for performance are gradually increased, resulting in the emergence of new network structures and a large number of new operators. Like YOLO [2], BERT [3], GPT [4], etc., they all consist of operators with different types, shapes, and connection relationships [5-9]. The diversity of hardware platforms and the increasing number of operators causes the cost of manually developing and implementing an optimal operator for each scenario to multiply.

The model compilation acceleration framework for deep learning can automatically generate optimized code for any hardware device, solving the problem of high labor costs. Among them, computational graph fusion optimization is the most important [10]. The computational graph of DNN is a directed graph, which includes: a group of nodes, each node representing an operation; A set of directed edges, each marking a relationship between nodes (data transfer and control dependencies). The speed of

neural network training and inference can be improved by computational graph optimization [11-15]. At present, the mainstream frameworks Tensorflow, Pytorch, TVM [16] and so on all adopt various graph optimization methods to accelerate the calculation: Tensorflow provides graph optimization API, which users can call directly; TVM uses operator fusion (operator fusion: combining multiple operators into one kernel without storing the intermediate results back into global memory) and so on.

The main graph optimization methods in existing deep learning compilers mainly have the following two problems: (1) It will cost a lot of manpower to manually invoke the interface tuning methods. (2) Tuning methods independent of hardware cannot maximize the utilization of hardware resources.

Most deep learning compilers classify operators and formulate operator fusion rules through self-designed intermediate representations (IR), but such static rules cannot adapt to specific hardware platforms and generate optimal subgraph. TVM uses Halide IR [17] to represent tensor operators, which can represent computation and scheduling processes in a relatively flexible way. But TVM also has the problem that the hardware of the computational graph mapping operator is not suitable. Although TVM claims to support mainstream deep learning frameworks and a large number of neural network models, there are few networks that TVM can adapt to DSA and FPGA at present, and few other network models are supported for the following reasons: Relay quantization currently only supports limited types of operations, and VTA [18] of TVM only supports limited types of operations.

The traditional coarse-grained operator abstraction makes the deep learning system only exist in the granularity of a single operator. The calculation is performed by two layers of scheduling hardware, namely the data flow chart execution engine and the scheduler hardware. The scheduling overhead is high, the parallelism between operators is not effectively utilized, and the interaction between the two kinds of parallelism operators is ignored. When the operation of a single operator does not saturate the target device, hardware parallelism cannot be exploited to the fullest extent. TASO [19] and Rammer [20] also package independent operations into a kernel for concurrent execution, but because of their reliance on manual scheduling templates, these methods do not support scenarios where custom operations are required.

AKG [21] uses the polyhedral model to represent the tensor operator, reducing the search space and adopting different ways of "calculation and scheduling" by calculating the characteristics of polyhedral [22] and drawing on TVM. However, this combination is difficult to achieve in the existing compilation work based on handwritten scheduling, because in complex scenarios, the mapping relationship between the underlying calculation and data is difficult to specify by handwriting. For DSA, in order to make the operator meet the memory limit and make full use of computing resources, it is necessary to optimize the operator based on the standard reference implementation. In order to make the optimized operator have hardware reusability and meet the hardware performance requirements, it is necessary to propose a multi-hardware computational graph optimization compiler that takes into account both operator layer and hardware performance requirements.

In this paper, the information of the operator layer and the hardware layer will be respectively used for the reasonable segmentation and fusion of the layer hardware resources compact. In the operator layer, based on the multistage fusion rules of different operators, the layers are fused in advance. Mark the subgraph to be evaluated for dynamic tuning; By collecting the actual operating power consumption and time consumption of hardware, the cost evaluation model is established, and the feedback guide subgraph is disassembled to balance the final operating performance. Compile and adapt for HUAWEI A310, Nvidia Jetson nano GPU and FPGA ZCU102, realizing adaptation and optimization of multiple back-end cores.

To sum up, this paper has the following contributions.

• Through the multilevel rule-splitting strategy, PCGC divides the computational graph into smaller subgraph to achieve an efficient and detailed search.

• PCGC uses the cost model to receive hardware information feedback, uses the cost model and operator fusion rules to guide the partial fusion decomposition of graph nodes and edges, and flexibly generates the optimal subgraph according to different hardware devices.

• PCGC combines hardware back-end code generation tools to further improve the performance of DNN inference. Compared with other advanced accelerators, PCGC optimizes the overall power consumption on an embedded GPU by an average of 130.5% when the time consumption on each hardware is not lower than the average time consumption. On DSA, PCGC optimizes power consumption by an average of 66.5%. On FPGA, PCGC optimizes power consumption by 66.1%. In a sense, PCGC can achieve high-speed inference in specific power supply scenarios, reducing the carbon emissions of edge computing.

This paper is organized as follows. §2 summarizes the relevant work and research motivation of this paper. §3 introduces the overall design and core strategy of this method. Then §4 shows the experimental results. §5 summarizes the work of this paper.

2 Background

2.1 Related works

Figure 1 shows most of the existing advanced DNN accelerated compilers, which we divide into graph optimization and tensor optimization compilers. MetaFlow [23], based on the maximum flow algorithm, is used to minimize the number of graph replacements across different subgraphs. However, this graph segmentation algorithm is prone to lose the opportunity of operator fusion optimization across subgraph. Moreover, the cost model only has high accuracy on DNN dense linear operators, and cannot support operators without thinning. TASO mainly uses graph substitution and uses the operators already in the operator library to generate equivalent subgraphs and replace them. However, it does not contain double computation (repeating the same computation with the same input), which will be considered an invalid

subgraph and will occupy a large amount of scheduling tuning resources. Rammer, because of its reliance on manual scheduling templates, is expensive to compile and cannot support scenarios that require custom operator tuning operations. APOLLO [24], proposed bottom-up multi-layer optimization. Firstly, polyhedral cyclic fusion is applied. And in order to do this fusion we have to first break the graph into a micro-graph fusion. The parallelism between operators is done by combining the unrelated results of the first two layers. But the integration of the lower level is not considered, so many integration opportunities will be lost. Although it made the second fusion repair (only two reduction-related fusion operations were completed), it still could not adapt to the rapidly developing neural network, and there may be some omissions. Computationally intensive operators do not consider fusion when doing parallel optimization, because such operators are easy to exhaust hardware resources. At present, multi-branch parallel *CONV* is quite popular in neural networks. A simple "computation-intensive operator" cannot divide parallelable and non-parallelable, which will miss the possible optimization opportunities. IOS [25], proposed a cost model-based dynamic parallel method between operators, which is mainly a recursive algorithm. The application platform of this algorithm is GPU. The concepts of stage and group are listed in detail, and the computational graph is grouped clearly and logically. But it doesn't say how the cost model does it, and it's not strongly related to the hardware. The optimization space can also be compressed, and the authors of the paper show that some models require further constraints on the optimization space. These DNN optimization compilers do not take into account the power loss of hardware and blindly pursue inference time-consuming optimization.

2.2 Motivation

The existing methods of graph tuning cannot maximize the utilization of hardware resources. It is not feasible to rely solely on the operator fusion rules, but relying entirely on the cost model may cause the computational graph compilation engine to impose too much power on the tuning process. In order to consider the cost of the optimization of DNN, the replacement of the hardware-compactness of the computational graph is carried out while the search space is reduced, and the optimization of the computational graph is combined with the operator optimization method to bring the maximum benefit to the optimization calculation of DNN. Therefore, this paper will start with the multi-rule operator and computational graph fusion method, and through the hardware performance of part of the molecular graph cost model repartition, reduce the search space, optimize the model inference power consumption and time.

3 Design And Implementation

3.1 Overall of System Architecture

PCGC is applied to embedded inference scenarios and supports inference acceleration in most DNN. PCGC is compatible with mainstream deep learning frameworks, mainly facing DSA, FPGA, GPU and other hardware.

The main process of PCGC is shown in Fig. 2.

PCGC takes the *.ONNX* of the model as input and converts it to TVM's Relay IR for further processing. For the subgraph at this time, according to the static operator fusion rule, the operator fusion is carried out for the first time to form a new composite subgraph. The operators in the subgraph are parallelized according to the hardware resources, and the second operator fusion is carried out to form a larger aggregation subgraph. On the subgraph, cross-boundary optimization operations such as constant propagation and common subexpression elimination are carried out. Next, according to the record of the previous fusion operator, the list of the original operator and the fusion operator is formed. The cost model is built on the target hardware platform, and the fusion effect is evaluated. If the evaluation results show that the performance becomes worse after fusion, then split the operator. Finally, the generated subgraph is used as the input of the operator layer to generate the corresponding hardware code.

The purpose of using Relay IR is to be more compatible with existing deep learning compilers so that this method can be more widely used in the field of deep learning compilation acceleration. The composition operator in the original model is expanded into a fine-grained operator, one is to further optimize the interior of the original operator and the other is to unify the operator type, reduce the number of operators provided by the framework, as well as facilitate the subsequent static fusion and cost model construction. Operator fusion and parallelization process accelerate the inference process from the point of view of reducing the reading of main memory and maximizing the utilization of hardware resources. Arranging operator fusion before parallelization processing can reduce the amount of data from parallelization processing and can cooperatively deal with the parallelism between operators and within operators. Finally, the cost model is used to evaluate the operator’s power and time, and the fusion operator is segmented so that the graph optimization scheme can dynamically adapt to different hardware backends. After fusion and parallelization, the evaluation of the performance of the operator by the cost model is closer to the result of its actual operation on hardware, and the predicted value is more accurate.

3.2 Intermediate Reference

PCGC accepts mainstream models such as Pytorch and Tensorflow as input and transforms them into custom IR that describe the structure of the computational graph. The IR defines the calculation type of each operator, the number of inputs of the operator, the number of outputs of the operator, and other necessary parameters of the calculation. For the fusion operators, it is composed of several operators, and the information on these operators is recorded. The resulting semantic tree is shown in Fig. 3.

3.3 Static Fusion

In this paper, a static fusion strategy with multilevel fusion rules is proposed, which combines operator characteristic to achieve reasonable fusion within and between graph nodes. The overall process is described below. According to the IR completed by the transformation, the computational graph is traversed in reverse order from the output, and the operators that meet the fusion rules are fused. Iterate

through the process several times until there is no operator in the graph that can be fused. Re-integrate the computational graph after fusion.

This paper uses the greedy strategy for operator fusion rules. The purpose of operator fusion is to reduce the number of times of moving data from main memory and in order to ensure that the possible fusion operators are not omitted in the subsequent cost model evaluation. As far as possible, all the operators that exist in data dependence are fused. However, it is necessary to ensure that the amount of computation of the fusion operator will not exceed the limit of capacity of the hardware, so some fusion which is generally considered to be very weak in improving performance is also abandoned. The criteria for basic integration should meet:

- There is data dependence between fusion operators.
- Fusion should not create directed loops between subgraphs.
- After fusion, the hardware resources allow the calculation to be completed without moving data from the main memory.

The operators commonly used in deep learning are divided into the following five categories: (1) Algebraic operators. Represents the calculation operation of the corresponding elements between two tensors, including *RELU*, *ADD*, *MUL*, etc. (2) Broadcast operator. Operators that deal with tensors of different shapes in the calculation process by copying operation, if the shape of the tensor needs to be adjusted in the calculation process of algebraic operators, it is also counted in the broadcast operator, including *BN* and so on. (3) Reduction operator. The operation of reducing the number of elements contained in the tensor according to the specified axis, including *SUM*, *ARGMAX*, etc. (4) Complex operator. There are some common operators in neural networks, such as *CONV5×5*, *FC* and so on. (5) Infusible operator. Refers to the operator that has no profit after fusion with other operators, including constants and so on. (6) Two-phase operator. Different operators are fused on different hardware, and it is impossible to determine whether there is a benefit after fusion, including *MAXPOOL*, *CONV1×1* and so on. On the basis of fusion criteria, the following static fusion rules are supported by a large number of historical experiments, as shown in Fig. 4. (The same kind of operator may be classified as different types of operators according to different specific parameters. For example, *CONV1×1* is a two-phase operator and *CONV5×5* is a complex operator. The specific classification depends on the actual situation.)

The stable fusion rule means that the fusion will improve the performance on most hardware platforms. For example, the fusion of *CONV + RELU + BN* conforms to rule one of stable fusion. The prohibition rule means that in most cases the fusion will degrade the performance or have no impact on the overall performance. In addition to prohibiting the fusion types prohibited by the fusion rules, the operator can carry out fusion on the premise that it conforms to the basic fusion rules. If the operator type conforms to the stable fusion rule, the operator will not be segmented later, otherwise, whether the fusion is effective or not will be evaluated by the cost model in the operator segmentation stage, and if the performance of the fusion is reduced, the fusion operator will be disassembled again. The operator fusion completed at

this stage cannot guarantee all positive effects on inference, because the performance of different operators on different hardware fluctuates. Therefore, hardware-aware operator segmentation is carried out to improve the fitness of the backend.

3.4 Parallel Processing

We learn from Rammer's idea [20] and adapt it for parallel optimization method for nodes and edges of multilevel rules combined with the characteristics of operators. The underlying hardware resources are abstracted into virtual computing units, and the computing to be carried out is scheduled to these computing units, so that the operators can complete parallelism. Through the operator parallelism within the fusion operator and the parallel between the fused operators, the hardware resources are used. The parallel structure is shown in Fig. 5.

Internal parallelism of operators: Combing the parallel logic between operators when merging operators, and parallel computing without data dependence within operators after fusion in the case of hardware resources permitting. Through the technique of cyclic optimization, the parallelism is not limited to the operators to be fused, but the computation within each operator is also accelerated.

Inter-operator parallelism: After operator fusion, continue to do parallel operations on the fusion operator. When the fusion operator is parallel, the internal parallelism of the operator is re-adjusted at the same time. If the extra resources can achieve inter-operator parallelism in the case of reducing the internal parallelism of operators, the inter-operator parallelism strategy is given priority.

1) Cross-boundary optimization: Traverse the computational graph using topological sorting. First, select a node with a degree of 0 to execute, then delete the node and its connected edges, and then find the next node with a degree of 0 to execute. In the process of traversal, constant folding, inline optimization and common sub-expression extraction are optimized.

2) Constant folding: If most of the inputs on which a calculation depends are constants, the result is calculated directly, replacing the nodes in the graph.

3) Inline optimization: Expand the function nodes with small code and simple calculations, and delete the useless nodes.

4) Common subexpression elimination: In a program, if several expressions have the same type, parameters, and input, they are called common subexpressions. For common subexpressions, only one of the expressions needs to be evaluated, and the values of the other expressions can be obtained by assignment. Record the calculation content that has been processed during the traversal process. If the calculation content to be processed is the same, directly replace the computational graph.

3.5 Cost Model

A dynamic partial rule-guided graph optimization method related to hardware performance is proposed in this paper. The main function of the cost model in PCGC is to evaluate the time and power consumption

of each operator running on the target hardware platform without actual deployment and to provide a benchmark for dynamic fusion and segmentation of computational graph. In order to achieve the automatic optimization of the target hardware platform perception, for each different hardware, it is necessary to build a new most appropriate cost model. The cost model using the deep learning method requires a large amount of data during construction, and it takes quite a long time to obtain data in actual use. In order to speed up the optimization process, the amount of data must be reduced. Therefore, the semi-supervised regression model is used as the benchmark model.

The process of building a cost model is shown in Fig. 6.

First, extract the corresponding pre-training model according to different hardware architectures, such as CPU, GPU, FPGA, DSA, etc.,. By traversing the computational graph, PCGC extracts the features of all operators (the fusion operator and the operators that make it up), including the size of the input data, the type of the input data, the number of operators that make up the fusion operator, and the Type of operation. At the same time, PCGC will also dynamically adjust the representation form of the computational graph according to the information of the hardware to be deployed. Hardware information mainly includes supported parallelism, supported memory transactions (such as 8 bytes, 32 bytes read and write), shared memory restrictions, register restrictions, etc. Then PCGC selects 10% operators to deploy on the hardware, and the actual measurement time and power consumption are used as label data. According to the semi-supervised algorithm, the label data is used to build the model and predict the unlabeled data. Finally PCGC adds high confidence data to the labeled dataset, uses it as labeled data and retrains the model. When more than a certain proportion of data in the data set is marked, the training ends and a trained cost model is obtained.

Algorithm 1: Multi-level Rule Guided Cost Model Construction

Data: A list of ops P
Accuracy correlation value AC
Result: A cost model for specific hardware $Cost()$

```
LoadPretrainedModel();
T = P.randomShuffle().getTopRank();
Labeled = [];
while |Labeled| / |P| < AC do
    getGroundTruth(T);
    Labeled.extend(T);
    train(Labeled);
    eval(P);
    T = P.sort().getTopRank();
end
```

In order to describe the execution process of the cost model in more detail, this paper gives Algorithm 1: Multilevel Rule Guided Cost Model Construction.

The input of the algorithm is the operator list *P* to be tested and a parameter *AC* related to the model accuracy, and finally outputs a trained cost model. Line1 loads the pre-trained model, and Line2 randomly shuffles the input operator list and takes out the top ones as part of preparing to obtain runtime data. Line4 ~ line10 is the process of model training. Firstly, obtain the actual inference time, power consumption or other required indicators of the operators in the list *T* on the hardware, and then add them to the label data set. A cost model is trained on labeled data and then used to predict other unlabeled data. Take out the operators with the highest predicted performance and prepare for the next round of actual testing. When the proportion of labeled data to the overall data exceeds *AC*, the construction of the cost model ends.

3.6 Dynamic Re-cutting of Subgraph

Based on the computational graph fusion method guided by static and dynamic multilevel rules mentioned earlier, this paper further proposes a dynamic partial graph optimization based on hardware performance feedback to realize the re-segmentation of partial subgraph. The purpose is to better meet the requirements of graphic optimization of target hardware inference. Traversing the computational graph in reverse order, according to the record of the fusion operator, the operators which do not conform to the stable fusion rules are re-segmented, and the operators before and after segmentation are recorded. The content of the record is mainly the operator characteristics needed by the cost model.

After segmentation, the internal parallel logic of some fusion operators is disrupted, which needs to be rearranged according to the rules of inter-operator parallelism, and it needs to be considered when training the cost model. The time-consuming and power-consuming cost models are used to evaluate the operators before and after segmentation. If the time and power consumption decrease after segmentation, the segmentation operator will be adjusted according to the needs of users. If the user requires the minimum time consumption, it will be segmented in the case of reduced time consumption; if the user requires the lowest power consumption, it will be segmented in the case of reduced power consumption. In order to prevent excessive time or power consumption, the upper limit of power consumption is set, and if the upper limit is exceeded, the segmentation operator is re-adjusted. The specific process is shown in Fig. 7.

Compared with other computational graph optimization methods, this method provides optimal computing diagrams for different hardware, solves the problem of low coupling between layer and hardware, and saves a lot of manual optimization. use the computer to automatically generate the possible optimal subgraph and evaluate its performance after actual deployment; comprehensively consider and optimize the power consumption and time of the inference process. The method of fusion before segmentation is used to combine dynamic optimization with static optimization to reduce the optimization cost and improve the optimization effect.

Algorithm 2: Subgraph Dynamic Re-cutting

Data: A list of subgraphs G
Fusion record R
Optimization Target $Target$
Threshold H
Result: A list of subgraphs after splitting G'

```
1  G' = ∅;
2  while G ≠ ∅ do
3      g = G.pop();
4      R_g = R.findRecord(g);
5      if R_g = ∅ then
6          G'.push(g);
7          continue;
8      end
9      min = Cost(g, None, Target);
10     mode = (g, None);
11     for r in R_g do
12         cost = Cost(g, r, Target);
13         if cost < min and Cost(G + g, r, !Target) < H then
14             min = cost;
15             mode = (g, r);
16         end
17     end
18     if r = None then
19         G'.push(g);
20     end
21     else
22         sg = g.spilt(r);
23         G.push(sg);
24     end
25 end
```

In order to elaborate the subgraph re-decomposition process in more detail, this paper presents Algorithm 2: Subgraph Dynamic Re-cutting.

The input of the algorithm is a current subgraph list *G*, the record *R* in the operator fusion process, the optimization target *Target*, and the threshold *H* specified by the user. *Target* is a string sequence corresponding to the optimization target, such as "time", "power" and so on. The output of the algorithm is subgraph sequence *G'* after splitting. Line2 ~ line25 traverse the list in *G* in turn, looking for possible optimizations. The function in Line4 searches for the record corresponding to the subgraph in *R*, which guide the process of subgraph generation. Each fusion point is also a splittable point. If there is no corresponding fusion record, it means that the granularity of the subgraph is the finest, and there is no possibility of further splitting, so it is directly added to *G'*. Line11 ~ line17 traverse these records in turn, and make several possible splits of the subgraph *g* according to the records. Line12 calls the cost model to evaluate the performance of the main optimization target after splitting. When the number is less than the currently recorded optimal value and after applying this split to the overall model, the secondary

optimization objective will not exceed the threshold *H*, then replace the optimal value and the corresponding split mode. If after evaluation, it is found that the performance is the best without splitting, then directly add the subgraph *g* to *G'*. If there is an optimal split, it will be split according to its pattern, and the split subgraph will be added to *G* again, waiting for the next evaluation.

3.7 Several implementation methods

Several implementation methods of PGCF are described below. The actual implementation is not limited to several implementation methods described in this paper. In order to better explain how this method is effectively optimized, first of all, some related technical concepts and principles are described.

Example 1

Through this example, the principle of reducing model inference time by this method is simply explained. A simple DNN structure is shown on the left side of Fig. 8. According to the invention, the result of the optimization is shown on the right side of Fig. 8. The original structure needs to store the data in memory after *CONV* calculation, and in the process of preparing for *BN*, it needs to move the data from memory to the on-chip cache. After calculating the *BN*, you also need to move the data twice to start the calculation of the *RELU* layer. According to the principle of operator fusion, the optimized computational graph structure will store the data of *CONV* calculation into the on-chip cache and directly carry out *BN* operation. After the *BN* calculation, the data is also stored in the cache, and then the *RELU* operation is performed directly. Compared with the original structure, the process of moving data from memory four times between operators is saved, and a lot of inference time is saved.

Example 2

Through this example, the principle of the strong correlation between this method and hardware is simply explained. The left side of Fig. 9. shows the structure before the fusion of the two operators in a neural network, the middle of Fig. 9 shows the structure of the two operators fused on hardware *A*, and the right side of Fig. 9 shows the structure of the fusion of the two operators on hardware *B*. For different hardware A and B, this method will construct different cost models, and output different computational graph according to the operator running time evaluated by the cost model.

For hardware *A*, we build a cost model *p*. The cost model *p* predicts that the running time of *CONV1* is *t1*, the running time of *CONV2* is *t2*, and the running time of the composite operator after fusion is *t3*. Because $t3 < t1 + t2$, it is decided to fuse the operator. The output is the computational graph in the middle of Fig. 9.

For hardware *B*, we build a cost model *q*. The cost model *q* predicts that the running time of *CONV1* is *t1'*, the running time of *CONV2* is *t2'*, and the running time of the composite operator after fusion is *t3'*. Because $t3' > t1' + t2'$, it is decided to split the operator and output the computational graph shown on the right side of Fig. 9.

Although the method of using the same set of operator fusion strategies for all hardware platforms has a small optimization cost, these strategies are obviously not suitable for every kind of hardware, and certain performance will be lost on some hardware. This method dynamically modifies the output computational graph in the face of different hardware, so that the power consumption of the model running on all corresponding hardware is minimized, while considered that operator fusion and splitting rules are based on time-consuming optimization.

Example 3

This example is used to simply illustrate the process of operator fusion of complex models by this method. Figure 10 shows part of the structure in Googlenet [26] on the left and the structure after static operator fusion using this method on the right. The inverse order traverses the computational graph, and the *Concat* operator inputs the infusible operator, so the operator is not fused. *CONV1×1* and *CONV3×3* satisfy the basic fusion rules but do not satisfy the stable fusion rules, so they are fused into the *CONV3-CONV1* operator, and the benefits of the operator fusion will be reevaluated later. *CONV1×1* operator and *CONV5×5* operator satisfy the basic fusion rules and are fused into the *CONV5-CONV1* operator. *CONV1×1* and *MaxPool3x3* satisfy the basic fusion rules but do not satisfy the stable fusion rules, so they are fused into *MaxPool3-CONV1* operators. *CONV1×1* and *MaxPool3×3* do not meet the basic fusion rules, so they are not fused. *CONV3×3* and *RELU* satisfy the stable fusion rule and merge into the *CONV3-RELU* operator, and the fusion operator will not be segmented later. The fusion of other operators is similar.

Example 4

Through this example, the parallel processing of the computational graph by this method is simply explained. Figure 11 marks the part of the computational graph after static operator fusion for parallel processing, and the other parts only do parallel processing within the operator. There is data dependence between operators in the blue box, so serial processing is done. There is no data dependence between operators in the red box, so parallel processing can be done. The degree of parallelism is determined according to the abstract virtual computing unit. If there are four computing units that can do convolution operations in the hardware at the same time, the four operators in the red box can be completely parallel, and the final running time only depends on the operator with the longest running time. If there is only one computing unit in the hardware that can do convolution operation, then the total operator of the red box can only be completely serial, and the final running time is the sum of the four operators.

Example 5

Through this example, the process of operator segmentation of DNN structure by this method is briefly explained. The operators that do not meet the stable fusion rules are re-evaluated by the cost model, and the inference time or power consumption before and after fusion are predicted respectively (selected according to the actual needs), and the operators with higher power consumption after fusion are re-disassembled. As shown in Fig. 12, after the evaluation of the cost model, *CONV1-CONV3* finds that the

inference time and power consumption are reduced after fusion, so the fusion state is maintained. After the evaluation of the cost model, it is found that the inference time and power consumption of the *CONV1-CONV5* fusion operator increase after fusion, so the two operators are disassembled and become unfused again.

4 Evaluation

4.1 Environment Setup

PCGC supports the model files generated by most mainstream frameworks, such as PyTorch, Tensorflow, MXnet, Caffe, Mindspore, etc., and can transform the model files into unified graph expansion representation files. This work is at the level of TVM: at the level of relay optimization, we carry out further subgraph fusion combined with hardware information. We will add two additional passes before all TIR pass to achieve operator fusion, namely the subgraph fusion pass guided by the cost model and the subgraph splitting pass. We use Python to describe the algorithm flow and use C + + to implement the corresponding hardware execution engine. The time and power consumption of a subgraph are measured directly in the execution engine to guide the scheduling process. The execution engine is based on the back-end libraries provided by TVM and also provides backend support for other DSA. In order to execute multiple sets of operators concurrently, PCGC puts different models into the corresponding CUDA and CANN flows as required. If there are sufficient computing resources, the kernels in different CUDA and CANN streams will execute in parallel. During the whole experiment, we used cuDNN 7.6.5 [27], CUDA 11.4, Nvidia driver 470.74, CANN5.1, TensorRT 7.13 [28], and TVM 0.8 as the basic library.

4.2 Results

In the experiment, we benchmark several modern DNN, including ResNet [29], YOLO and so on. In this experiment, in addition to other operators such as *Concat*, we also use the operator types shown in these models as the basic scheduling units of computational graph. Some models (for example, ResNet) may have some potential parallelization opportunities. For example, for ResNet-50 and ResNet-34 [30], we can achieve 10–15% acceleration through parallel sample convolution.

In the evaluation of these models, we conducted five experiments each and reported the average performance. Each experiment is tuned on Nvidia Tesla V100 GPU and inferred on FPGA ZCU102, Nvidia Jetson nano and Huawei A310. Considering the inference delay, we try to optimize the power consumption according to the target.

We designed experiments to evaluate the average Model running power consumption of a single image. In our experiment, we prepare 10,000 images for edge computing hardware to perform inference, and collect the average operating power through the official API. At the same time, record the time consumption of each group of experiments, and compare the average time consumption of each image in the final inference with the average time consumption of the most advanced model accelerators.

There are eight baselines in this lab: XLA [31], TASO, MetaFlow, TVM (VTA), APOLLO, AKG, Xilinx DPU and TensorRT. For a fair comparison, we only compare corresponding accelerators on different hardware. Figure 13 and Table 1,2,3,4,5,6 show the actual measured values of PCGC on the benchmark DNN with other benchmark accelerators. Compared with existing accelerators, PCGC can achieve a substantial optimization of power consumption under different hardware and time-consuming inference tasks of different models.

For the FPGA platform, we choose Xilinx ZYNQ ZCU102 for verification. In this article, we use Vivado HLS (v2022.1) for C code compilation and logic simulation. We simulate and debug the algorithm function, and add hardware constraints and timing simulation to the simulation platform. Finally, the simulation platform generates and loads onboard bitstream that is deployed to Zynq UltraScale + MPSoC ZCU102. By analyzing the deployment experimental results of this framework, the correctness verification and performance evaluation are verified. We set its working frequency as 230 MHz for all designs and use 16-bit fixed data type.

We run the model YOLOv5, perform inference on 10,000 pictures with a size of 1024×1024, and record the average power per second as 3.15W through Xilinx official tools. More optimized inference efficiency than existing accelerators.

Table 1
Average inference performance data for running YOLOv5 on ZCU102

No.	Board	Scale (w*h)	Inference (FPS)	Performance Efficiency(GOPS/W)	Ave Power(W)
1	ZCU102	1024*1024	36	128.1	3.15
2	ZCU102	640*512	87	128.1	2.82

Table 2
Performance comparison between accelerators running YOLOv5 on ZCU102

	Xilinx DPU	TVM(VTA)	PCGC
No.1 Inference (FPS)	31	32	36
No.2 Inference (FPS)	79	81	87
No.1 Ave Power(W)	4.49	4.83	3.15
No.2 Ave Power(W)	4.52	4.68	2.82
Performance Efficiency(GOPS/W)	97.9	100.4	128.1

For the GPU platform, we chose the low-power Nvidia edge computing platform Jetson nano. Here we can compare more accelerators, because most accelerators only support GPU. By running the DNN models, we count the inference time consumption of 10,000 images with a size of 1024×1024 and the average

power recorded by official tools. The actual test data is shown in Table 3–4, which achieves more optimized inference than existing accelerators.

Table 3
Average inference time for a single image running the model on Jetson nano (in ms)

	TASO	TVM	TensorRT	XLA	MetaFlow	PCGC
YOLOv5s	158.1	169.5	**152.3**	193.4	171.8	**159.3**
YOLOv4 (16FP)	547.4	562.3	**528.5**	694.9	596.4	**551.7**
YOLOv4 (32FP)	742.3	761.4	**699.7**	803.8	783.1	**741.3**
YOLOv4-tiny(batch 1)	43.7	45.2	**41.9**	53.3	49.6	**44.9**
YOLOv4-tiny(batch 4)	38.8	38.9	**35.6**	51.1	45.2	**39.7**
Resnet-50	**12.1**	13.9	12.2	18.7	15.3	**13.4**
Resnet-34	11.9	**10.1**	10.8	16.2	14.1	**11.3**

Table 4
Average inference power consumption of models running on Jetson nano (in W)

	TASO	TVM	TensorRT	XLA	MetaFlow	PCGC
YOLOv5s	1.87	1.93	1.96	1.89	1.91	**1.41**
YOLOv4 (16FP)	2.31	2.32	2.51	2.19	2.54	**1.83**
YOLOv4 (32FP)	2.64	2.56	2.73	2.48	2.83	**1.97**
YOLOv4-tiny(batch 1)	1.67	1.72	2.12	1.93	2.41	**1.04**
YOLOv4-tiny(batch 4)	1.69	1.74	2.16	1.91	2.43	**1.01**
Resnet-50	1.66	1.59	1.89	1.67	1.74	**0.82**
Resnet-34	1.68	1.61	1.96	1.72	1.81	**0.91**

For DSA, we choose the Atlas200DK of Huawei A310 chip for the experiment. The experimental process is similar to that of an embedded GPU. We also count the inference time consumption of 10,000 images with a size of 1024×1024 and the average power recorded by official tools. We obtain the time-consuming and power consumption of the model by analyzing the profiling data, and record them. We find that PCGC performs better in terms of power optimization.

Table 5
Average inference time for a single image of the model running on Atlas200DK (in ms)

	APOLLO	AKG	PCGC
YOLO-X-nano	28.3	25.4	**28.4**
YOLO-X-tiny	30.4	27.0	**30.4**
YOLO-X-s	26.5	23.3	**26.4**
YOLO-X-x	100.5	95.2	**100.4**
YOLO-v5-s	55.0	52.4	**55.7**
YOLO-v5-x	103.1	101.5	**103.8**

Table 6
Average inference power consumption of models running on Atlas200DK (in W)

	APOLLO	AKG	PCGC
YOLO-X-nano	4.83	4.55	**2.74**
YOLO-X-tiny	4.96	4.91	**2.23**
YOLO-X-s	5.22	4.98	**2.51**
YOLO-X-x	5.31	4.76	**3.19**
YOLO-v5-s	5.15	4.81	**3.04**
YOLO-v5-x	5.10	4.79	**3.08**

Currently, we have a larger model to explore in the tuning section. We experimented with some computer vision (classification, detection, etc.) models in the fields of natural language processing and convolutional neural networks, compared with the benchmark baseline of Ansor [32], and the experimental platform is Nvidia V100 GPU. In the comparison experiment, we conducted at least 512 task trails. For those tasks with a small number of tasks (such as BERT, etc.), we increased the number of trails to 20,000 to ensure convergence. Figure 13 shows the speedup relative to Ansor.

For the current results, neither Ansor nor our results use the functionality of Tensor-Core. Some benchmark deep learning acceleration libraries such as CuBLAS [33] and CuDNN may achieve significant acceleration on some computing-intensive models (Transformers [34]) with the help of Tensor-Core.

5 Conclusion

In this paper, we proposed a new subgraph fusion and split optimization method, which could maximize the inference performance of the model on the hardware platform through the partial optimization of multilevel rules and the cost model of hardware performance index constraints. In addition, we also considered the execution rules of different operators on the hardware platform, optimized the parallel and partial execution of the model computational graph, and then evaluated the computing time and power consumption of nearly ten benchmark networks on FPGA, GPU, and DSA by using the image data set to evaluate the performance of the whole compiler system. Compared with other advanced accelerators, PCGC optimized the overall power consumption on Jetson nano by an average of 130.5% when the time consumption on each hardware was not lower than the average time consumption. On Atlas200DK, PCGC optimized power consumption by an average of 66.5%. On ZCU102, PCGC optimized power consumption by 66.1%. In a sense, PCGC could achieve high-speed inference in specific power supply scenarios, reducing the carbon emissions of edge computing.

Declarations

Ethical Approval

Written informed consent was obtained from all the participants prior to the enrollment (or for the publication) of this study.

Competing interests

All authors disclosed no relevant relationships.

Authors' contributions

Dong Dong and Hanqun Lin participated in the design of the study and performed the statistical analysis. Dong Dong and Hongxu Jiang conceived of the study, and participated in its design and coordination and helped to draft the manuscript. Dong Dong and Yanfei Song prepared figures and carried out experiments. All authors read and approved the final manuscript.

Funding

This work was supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

Availability of data and materials

The datasets generated or analyzed during this study are available from the corresponding author on reasonable request.

References

1. Jouppi N P, Young C, Patil N, et al. A domain-specific architecture for deep neural networks[J]. Communications of the ACM, 2018, 61(9): 50–59.
2. Redmon J, Divvala S, Girshick R, et al. You only look once: Unified, real-time object detection[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 779–788.
3. Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.
4. Floridi L, Chiriatti M. GPT-3: Its nature, scope, limits, and consequences[J]. Minds and Machines, 2020, 30(4): 681–694.
5. Looks M, Herreshoff M, Hutchins D L, et al. Deep learning with dynamic computation graphs[J]. arXiv preprint arXiv:1702.02181, 2017.
6. Millidge B, Tschantz A, Buckley C L. Predictive coding approximates backprop along arbitrary computation graphs[J]. Neural Computation, 2022, 34(6): 1329–1368.
7. Duan Y, Wang J, Ma H, et al. Residual convolutional graph neural network with subgraph attention pooling[J]. Tsinghua Science and Technology, 2022, 27(4):653–663.
8. Ingolfsson T M, Vero M, Wang X, et al. Reducing neural architecture search spaces with training-free statistics and computational graph clustering[J]. 2022.
9. Yuan Q, FT Szczypiński, Jelfs K E. Explainable graph neural networks for organic cages[J]. Digital Discovery, 2022, 1.
10. Liu Y, Zhang H, Xu D, et al. Graph transformer network with temporal kernel attention for skeleton-based action recognition[J]. Knowledge-Based Systems, 2022, 240:108146-.
11. Ganguly S, Bhowal P, Oliva D, et al. BLeafNet: A Bonferroni mean operator based fusion of CNN models for plant identification using leaf image classification[J]. Ecological informatics: an international journal on ecoinformatics and computational ecology, 2022:69.
12. Niu W, Guan J, Wang Y, et al. DNNFusion: accelerating deep neural networks execution with advanced operator fusion[J]. ACM, 2021.
13. Cza B, Zf A. Convolutional analysis operator learning for multifocus image fusion. 2022.
14. Schneider S, Wu K L. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes[J]. Acm Sigplan Notices, 2017, 52(6):648–661.
15. Menon P, Mowry T C, Pavlo A. Relaxed operator fusion for in-memory databases[J]. Proceedings of the Vldb Endowment, 2017, 11(1):1–13.
16. Chen T, Moreau T, Jiang Z, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 578–594.

17. Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. Acm Sigplan Notices, 2013, 48(6): 519–530.
18. Moreau T, Chen T, Vega L, et al. A hardware–software blueprint for flexible deep learning specialization[J]. IEEE Micro, 2019, 39(5): 8–16.
19. Jia Z, Padon O, Thomas J, et al. TASO: optimizing deep learning computation with automatic generation of graph substitutions[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 47–62.
20. Ma L, Xie Z, Yang Z, et al. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020: 881–897.
21. Zhao J, Li B, Nie W, et al. AKG: automatic kernel generation for neural processing units using polyhedral transformations[C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 1233–1248.
22. Wang J, Guo L, Cong J. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga[C]//The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2021: 93–104.
23. Jia Z, Thomas J, Warszawski T, et al. Optimizing DNN computation with relaxed graph substitutions[J]. Proceedings of Machine Learning and Systems, 2019, 1: 27–39.
24. Zhao J, Gao X, Xia R, et al. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization[J]. Proceedings of Machine Learning and Systems, 2022, 4: 1–19.
25. Ding Y, Zhu L, Jia Z, et al. Ios: Inter-operator scheduler for cnn acceleration[J]. Proceedings of Machine Learning and Systems, 2021, 3: 167–180.
26. Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 1–9.
27. Chetlur S, Woolley C, Vandermersch P, et al. cudnn: Efficient primitives for deep learning[J]. arXiv preprint arXiv:1410.0759, 2014.
28. Vanholder H. Efficient inference with tensorrt[C]//GPU Technology Conference. 2016, 1: 2.
29. He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770–778.
30. Koonce B. ResNet 34[M]//Convolutional Neural Networks with Swift for Tensorflow. Apress, Berkeley, CA, 2021: 51–61.
31. Sabne A. Xla: Compiling machine learning for peak performance[J]. 2020.
32. Zheng L, Jia C, Sun M, et al. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning[C]//14th USENIX symposium on operating systems design and implementation (OSDI 20). 2020: 863–879.

33. Barrachina S, Castillo M, Igual F D, et al. Evaluation and tuning of the level 3 CUBLAS for graphics processors[C]//2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, 2008: 1–8.
34. Parmar N, Vaswani A, Uszkoreit J, et al. Image transformer[C]//International conference on machine learning. PMLR, 2018: 4055–4064.

Figures

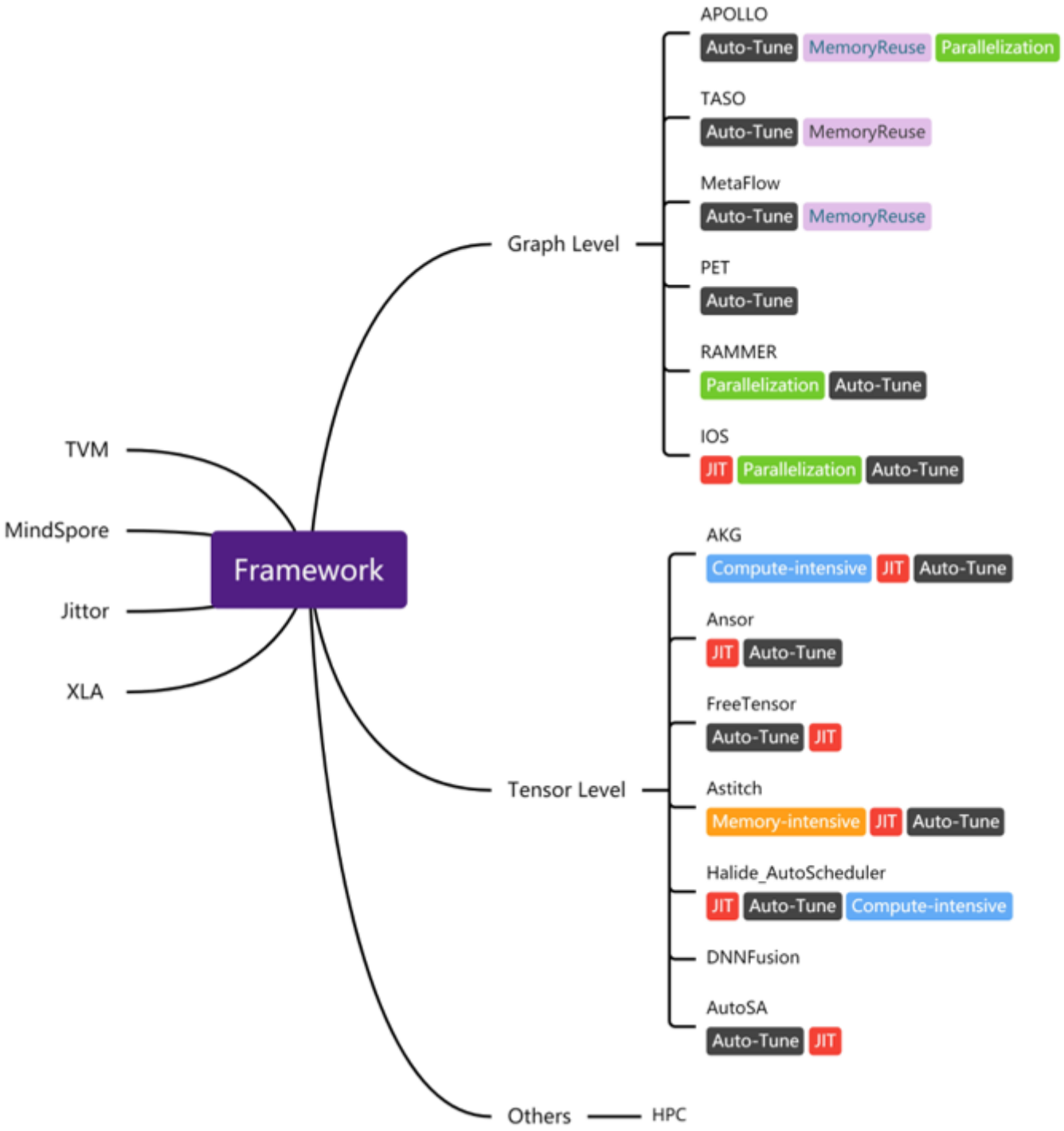

Figure 1

Advanced DNN Accelerated Compilers

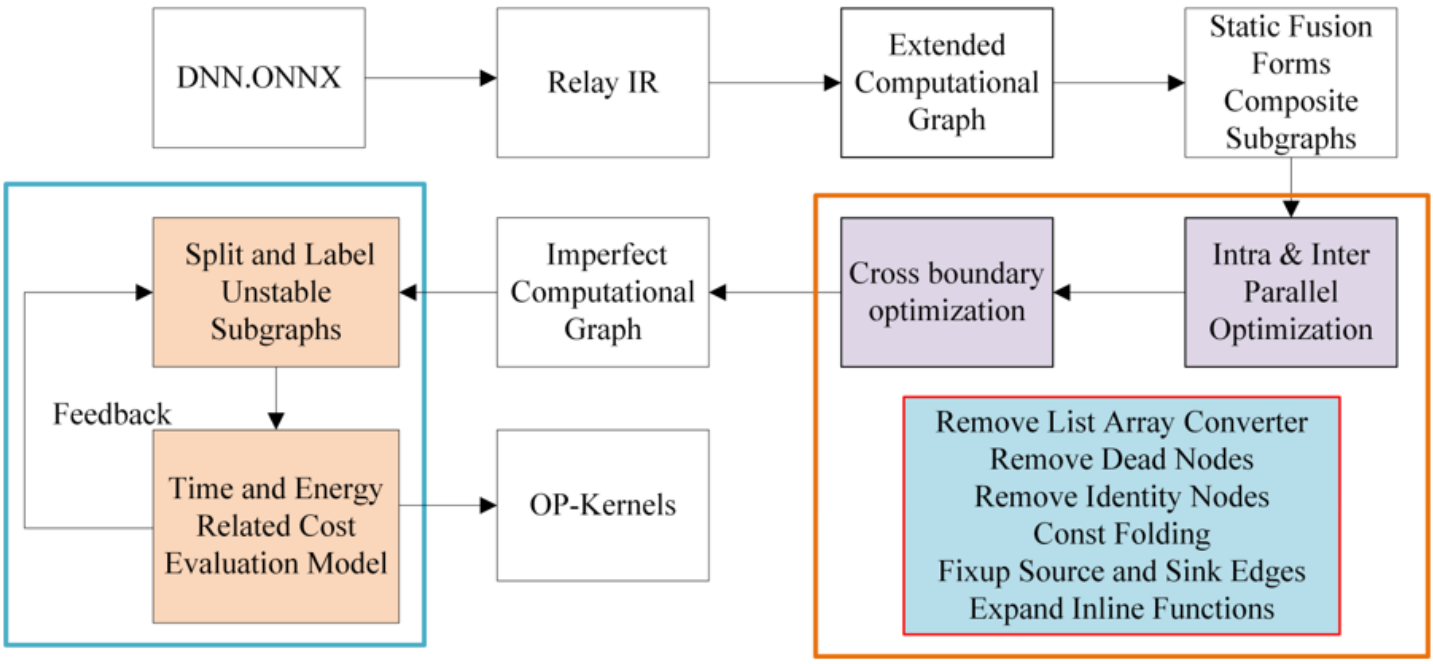

Figure 2

Overall of PCGC

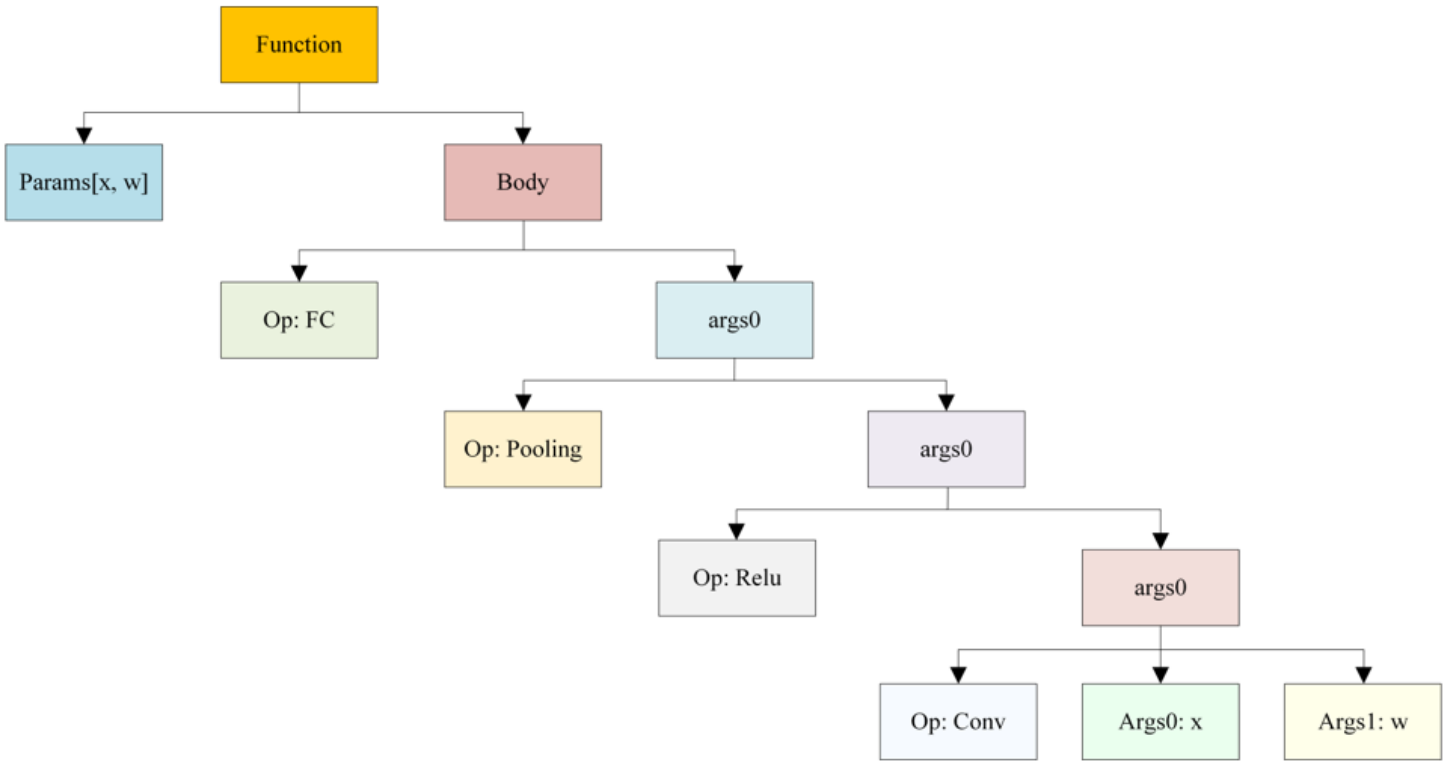

Figure 3

Semantic Tree of a Simple Network

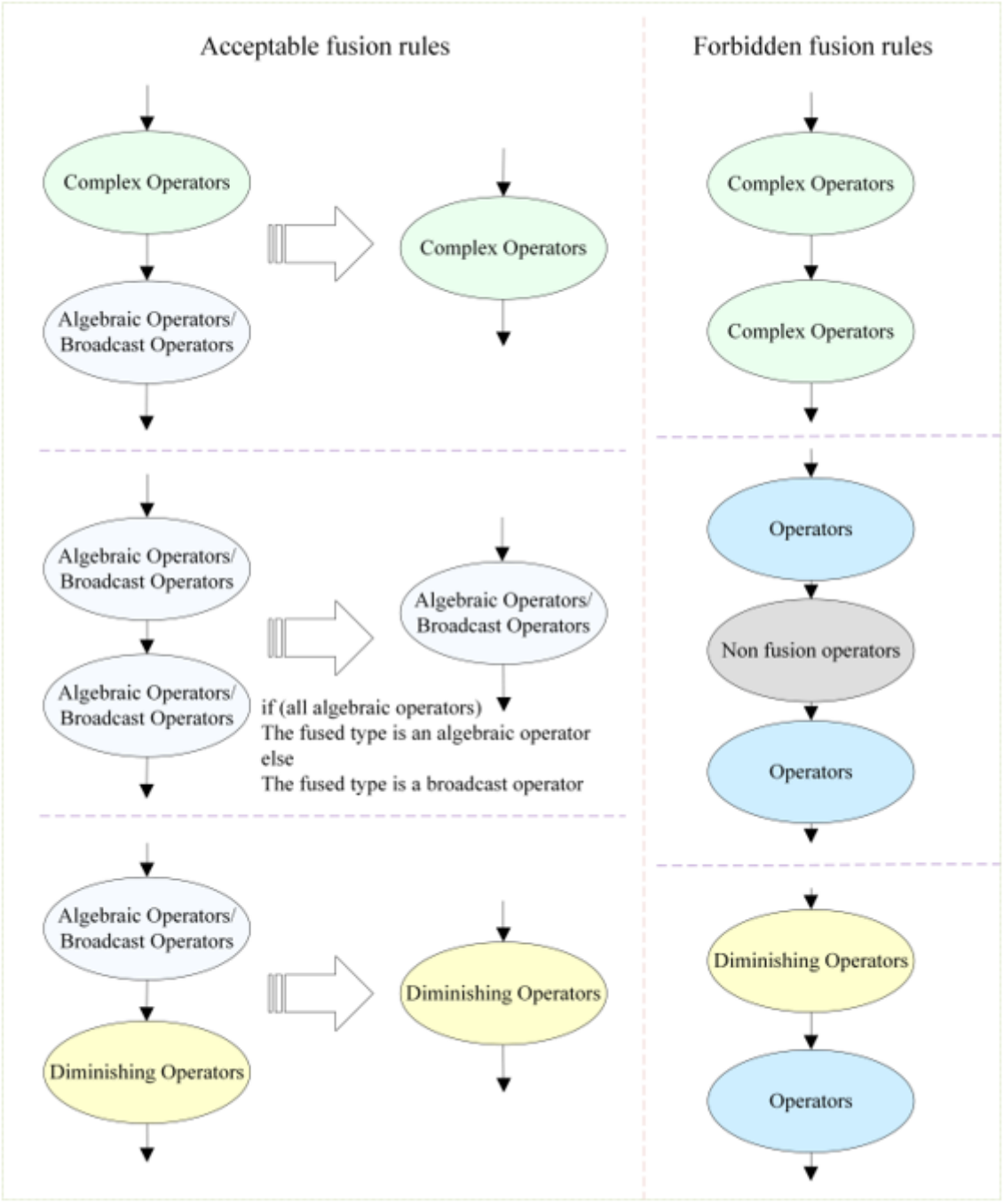

Figure 4

Operator Fusion Rules

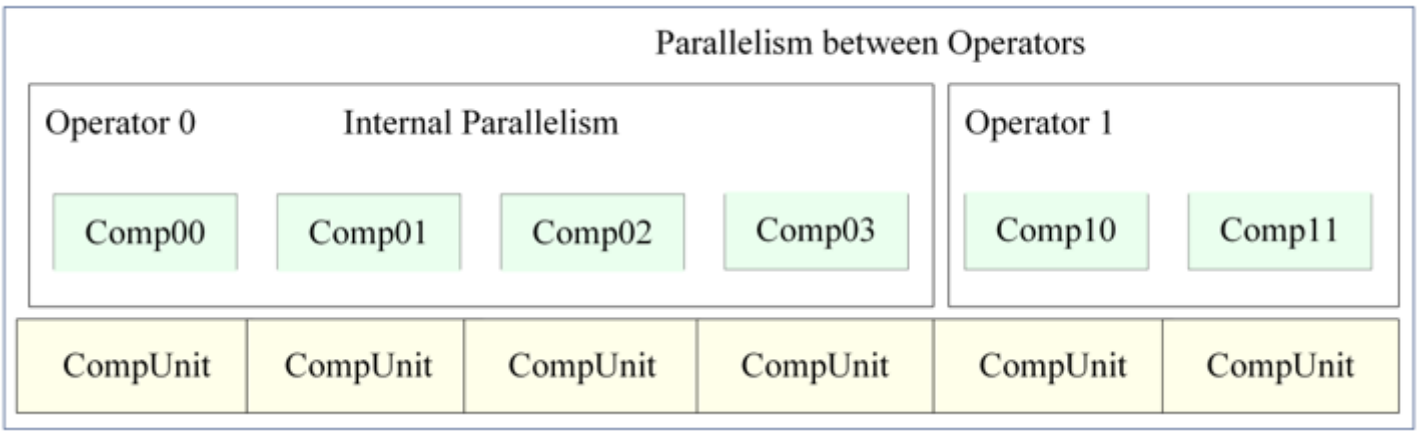

Figure 5

Schematic Diagram of Parallel Operators

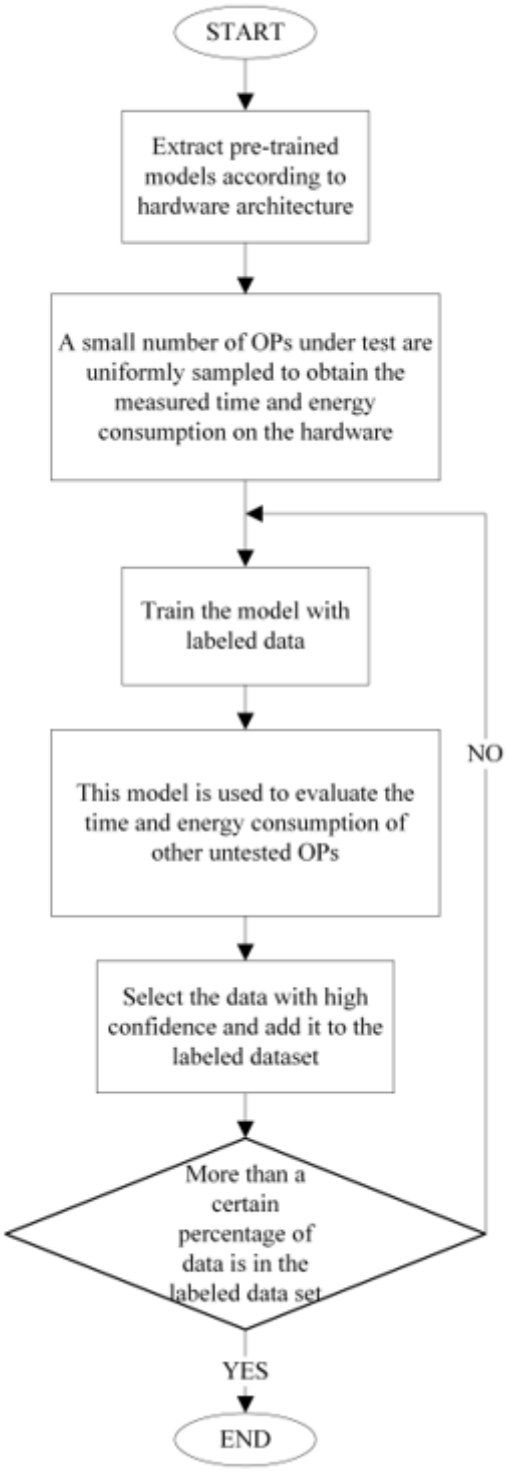

Figure 6

Cost Model Construction

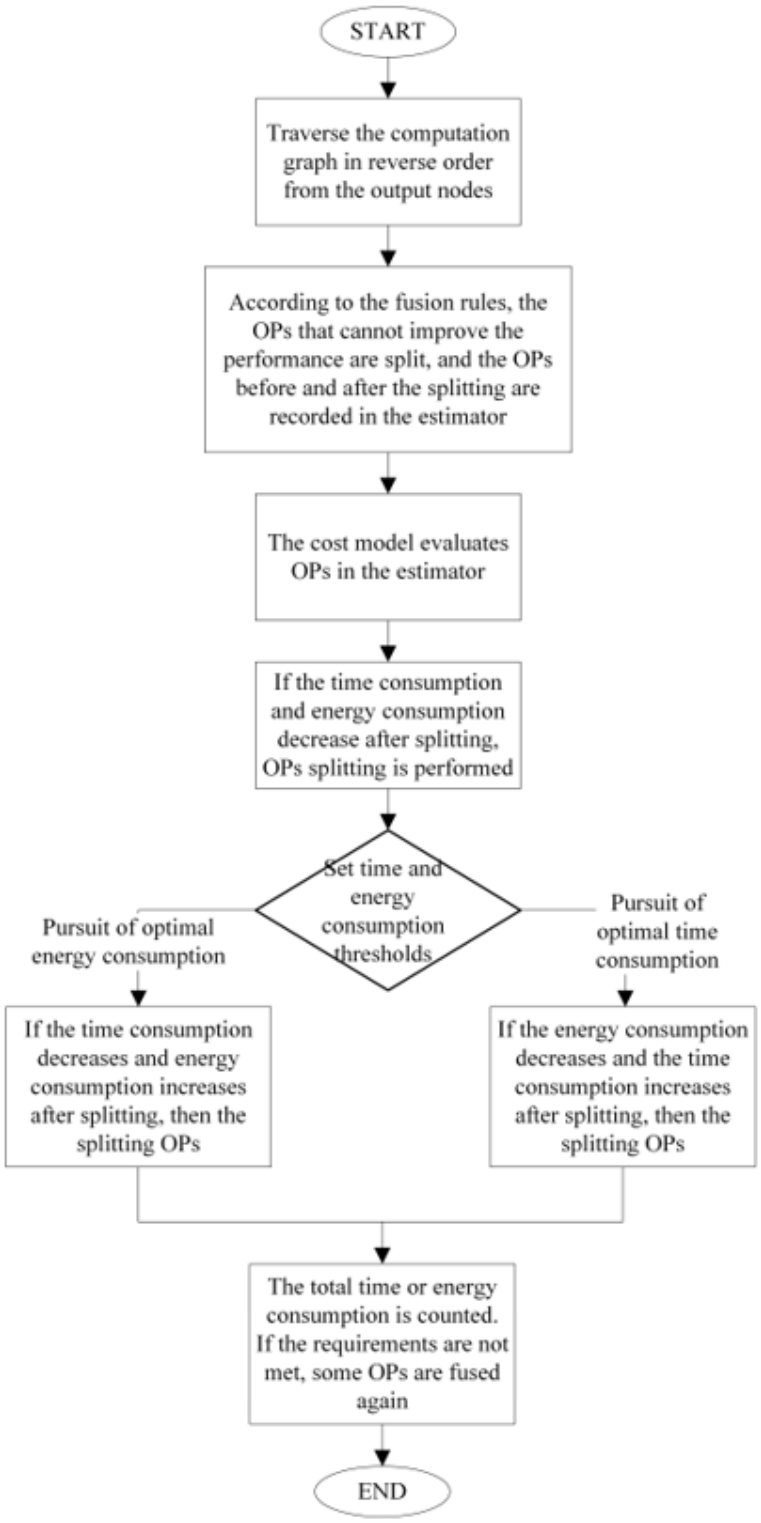

Figure 7

Flow Chart of Dynamic Re-cutting of Subgraph

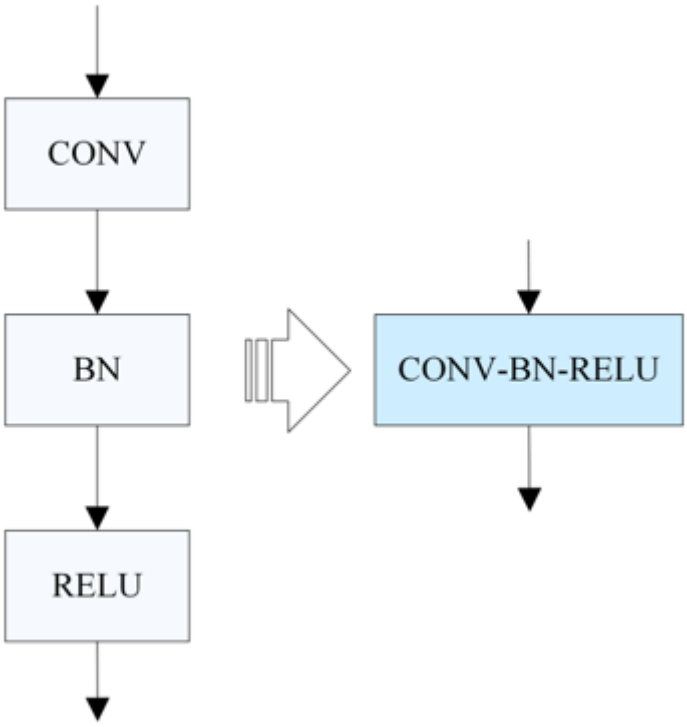

Figure 8

Fusion&Splitting Example 1

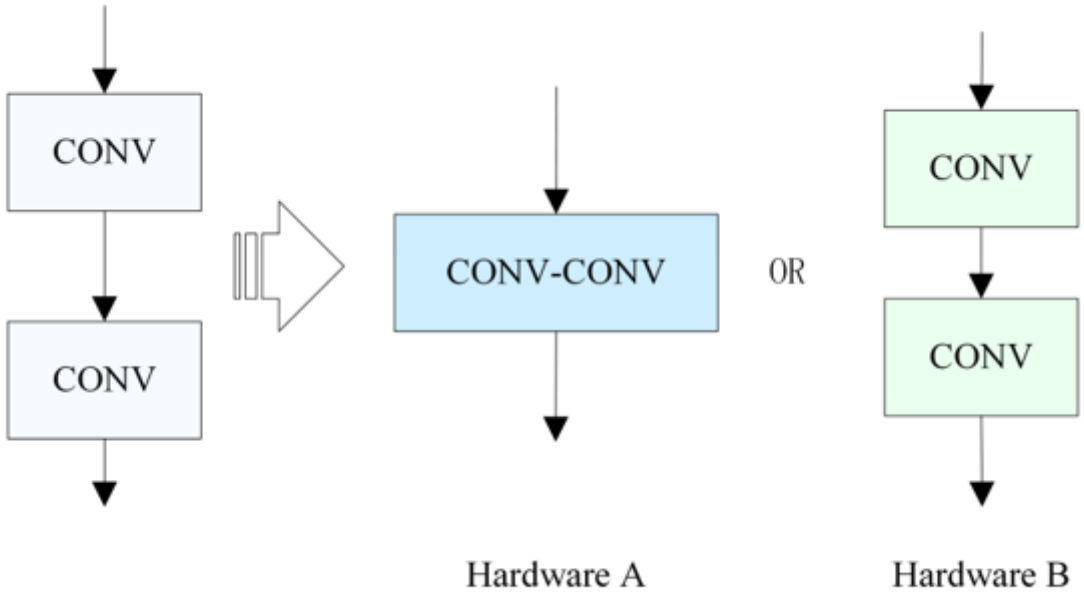

Figure 9

Fusion&Splitting Example 2

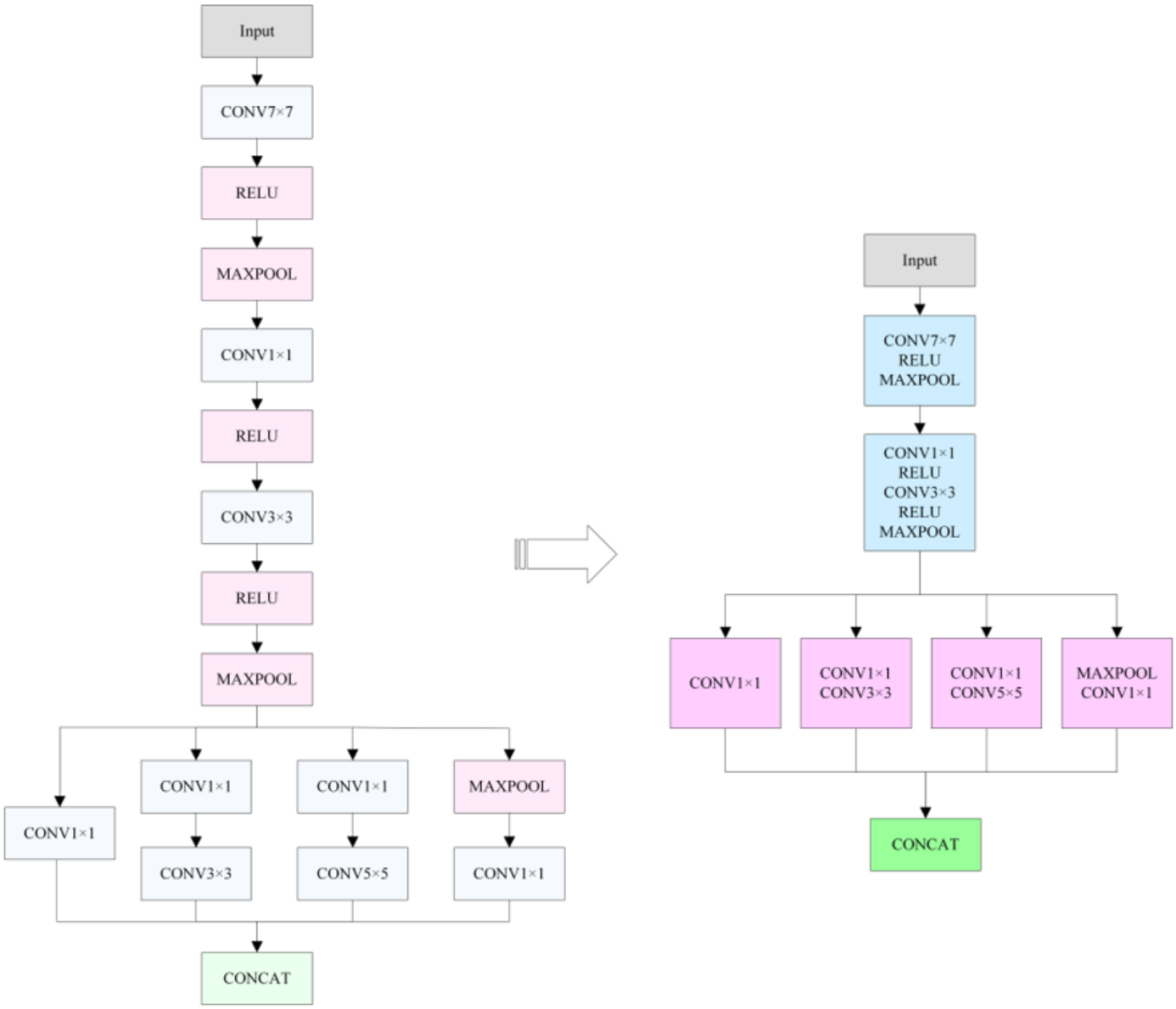

Figure 10

Fusion&Splitting Example 3

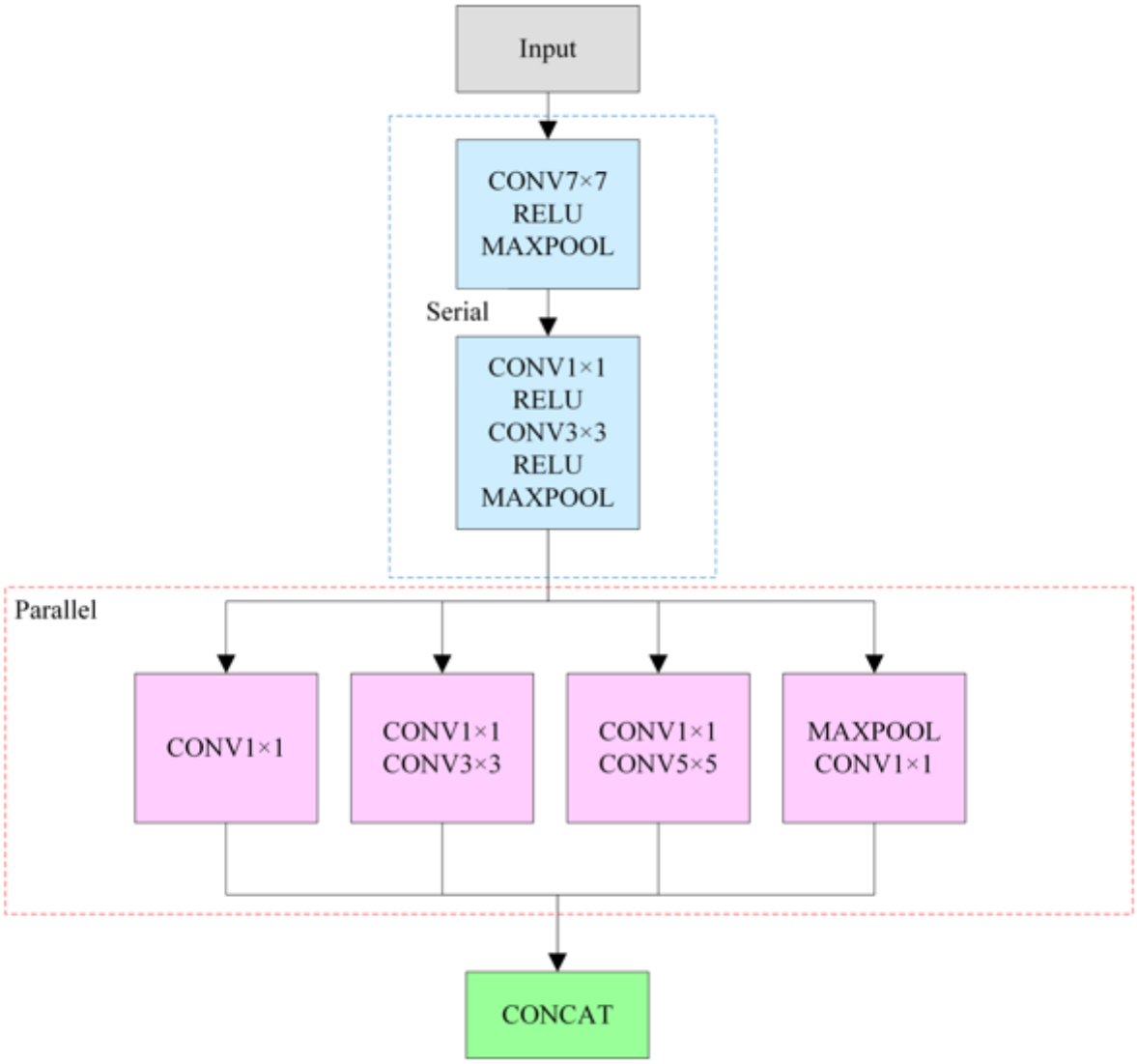

Figure 11

Fusion&Splitting Example 4

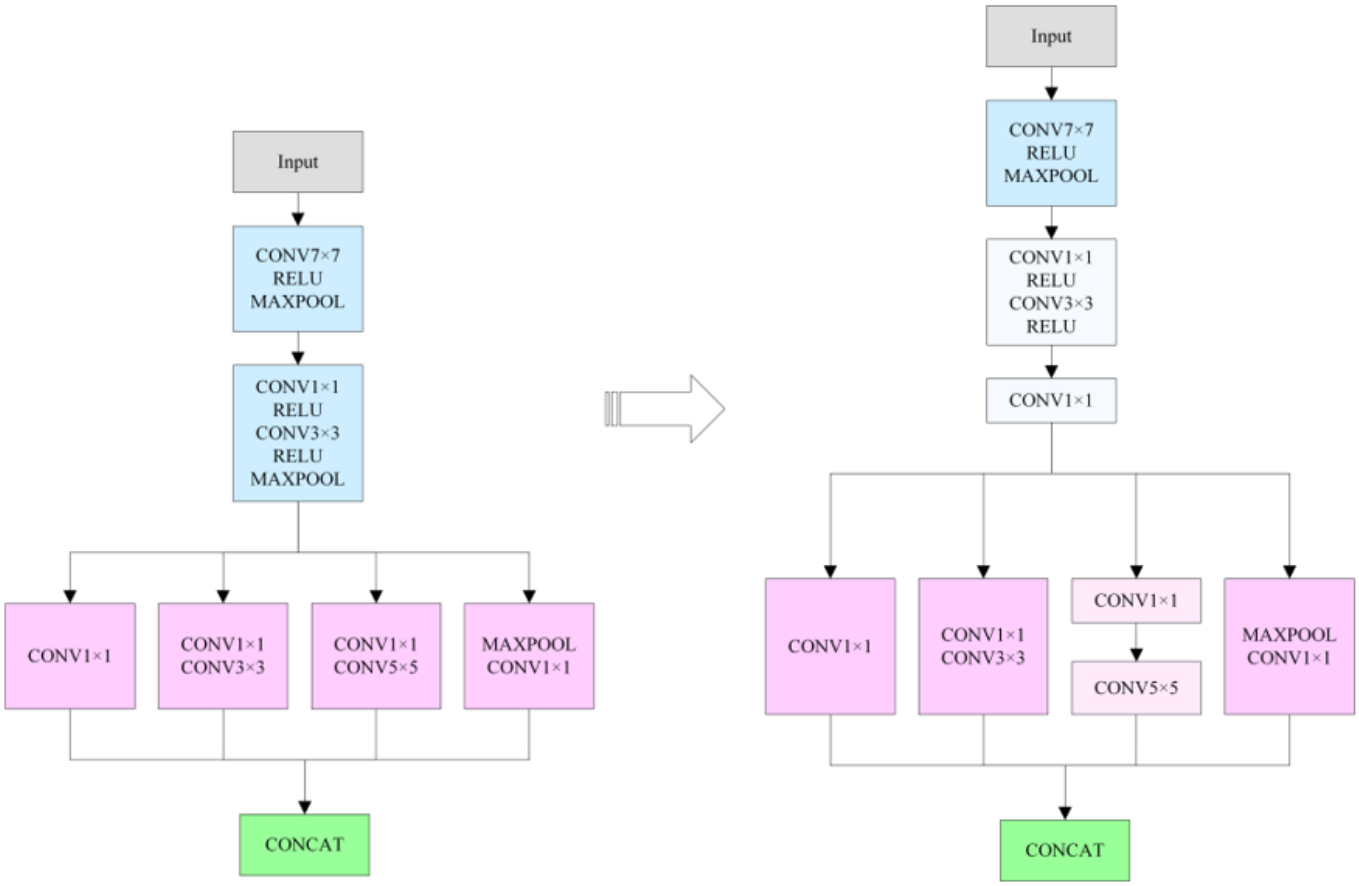

Figure 12

Fusion&Splitting Example 5

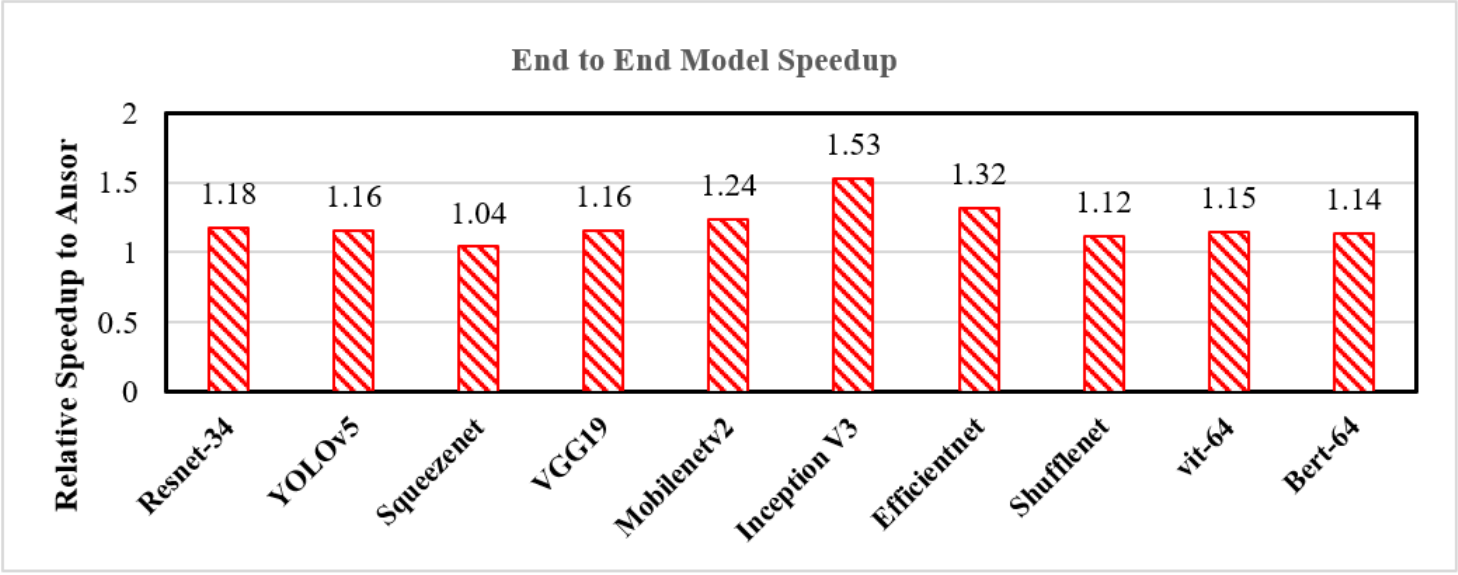

Figure 13

The Speedup Ratio of PCGC Compared to Ansor