Chen XB, Qi H, Peng SH *et al.* Tetris: A heuristic static memory management framework for uniform memory multicore neural network accelerators. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(6): 1255–1270 Nov. 2022. DOI 10.1007/s11390-021-1213-3

Tetris: A Heuristic Static Memory Management Framework for Uniform Memory Multicore Neural Network Accelerators

Xiao-Bing Chen[1,2] (陈小兵), *Student Member*, *CCF*, Hao Qi[3] (齐　豪), Shao-Hui Peng[1,2] (彭少辉)
Yi-Min Zhuang[1,2] (庄毅敏), Tian Zhi[1,*] (支　天), *Member*, *CCF*, and
Yun-Ji Chen[1,2,4] (陈云霁), *Distinguished Member*, *CCF*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China*

[4] *Chinese Academy of Sciences Center for Excellence in Brain Science and Intelligence Technology, Shanghai 200031 China*

E-mail: chenxiaobing@ict.ac.cn; theqihao@mail.ustc.edu.cn; {pengshaohui18z, zhuangyimin, zhitian, cyj}@ict.ac.cn

Received December 10, 2020; accepted May 31, 2021.

Abstract Uniform memory multicore neural network accelerators (UNNAs) furnish huge computing power to emerging neural network applications. Meanwhile, with neural network architectures going deeper and wider, the limited memory capacity has become a constraint to deploy models on UNNA platforms. Therefore how to efficiently manage memory space and how to reduce workload footprints are urgently significant. In this paper, we propose Tetris: a heuristic static memory management framework for UNNA platforms. Tetris reconstructs execution flows and synchronization relationships among cores to analyze each tensor's liveness interval. Then the memory management problem is converted to a sequence permutation problem. Tetris uses a genetic algorithm to explore the permutation space to optimize the memory management strategy and reduce memory footprints. We evaluate several typical neural networks and the experimental results demonstrate that Tetris outperforms the state-of-the-art memory allocation methods, and achieves an average memory reduction ratio of 91.9% and 87.9% for a quad-core and a 16-core Cambricon-X platform, respectively.

Keywords multicore neural network accelerator, liveness analysis, static memory management, memory reuse, genetic algorithm

1 Introduction

Deep neural networks (DNNs) perform well in a spectrum of complex problems. In computer vision, the model proposed by Microsoft Research [1] surpasses humans on the ImageNet classification task. In neural language processing, the state-of-the-art results by BERT [2] outperform human performance on the SQuAD v1.1 questions. In computer games, neural networks designed by DeepMind [3,4] achieve superhuman performance and occupy the top positions of Go's rank list.

The remarkable modeling capabilities of DNNs are inseparable from numerous parameters and tremendous connections of their architectures. For example, 9-layer AlexNet [5] which won the 2012 ILSVRC has six million parameters, while ResNeXt101 32x48d [6] with better image recognition accuracy has over 829 million

Regular Paper

This work is partially supported by the Beijing Natural Science Foundation under Grant No. JQ18013, the National Natural Science Foundation of China under Grant Nos. 61925208, 61732007, 61732002 and 61906179, the Strategic Priority Research Program of Chinese Academy of Sciences (CAS) under Grant No. XDB32050200, the Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI) and Xplore Prize.

*Corresponding Author

parameters. In the machine translation domain, the Sparsely-Gated Mixture-of-Experts layer designed by Google Brain[7] has up to 137 billion parameters.

While neural networks are becoming deeper and wider explosively, the limited memory capacity becomes a critical bottleneck to deploy applications. Thus, it is a deserved research to reduce model footprints and efficiently manage the memory space. Several types of research[8–10] focus on efficiently managing memory on GPU systems by liveness analysis, data recomputation, and data swapping. But these methods are not applicable to uniform memory multicore neural network accelerator (UNNA) platforms. DNN workloads on GPU systems are executed layer by layer through optimized high-performance libraries like cuDNN[11] and cuBLAS[12]. However, UNNAs[13,14] typically generate a kernel for a whole network or a sub-network to eliminate the overhead introduced by launching kernels and achieve inter-layer optimization. This programming paradigm introduces difficulties to existing memory management strategies. Since the kernel is uninterruptable, the intermediate data inside a fused graph cannot be spilled from the UNNA memory to host devices[8–10]. In addition, layer-wise liveness analysis like [9] may miss some potential memory reuse opportunities, because an operation might be split into sub-operations[15] and executed on different cores without a wall clock. It is pretty challenging to manage memory efficiently and reduce footprints of DNNs on UNNA platforms.

In this paper, we propose Tetris, a heuristic memory management framework for UNNA platforms. Tetris enables deploying large-scale neural networks by the exquisite memory reuse strategy and frees programmers from the arduous memory management work. Tetris is composed of two parts: the front-end and the back-end. The front-end takes instructions generated by UNNA compiler stacks as input, analyzes tensor liveness intervals, and generates a sign matrix to record whether two tensors can share the same physical memory space. The back-end takes the sign matrix as input, and leverages a heuristic method to find an optimized memory allocation configuration. Then the back-end calculates the total memory footprint and allocates a uniform memory pool with the minimal size. Finally, for each tensor, the back-end returns an offset to the memory pool as its address. We evaluate Tetris on a quad-core and a 16-core Cambricon platform with multicore configurations, and the results show that Tetris achieves remarkable memory reduction ratios.

The contributions of this work are as follows.

- We develop a framework called Tetris for UNNA platforms to reduce neural network memory footprints. Tetris extracts fine-grained tensor liveness intervals, constructs reusable relationships, and searches for the optimal memory allocation strategy heuristically.
- We propose a static analysis method to realize the fine-grained liveness analysis for multicore systems. This method extracts each core's basic execution patterns and inter-core synchronization relationships, represents relative liveness intervals in a graph, and generates a conflict matrix to indicate whether any two tensors could reuse the same memory space.
- We convert the memory allocation problem into a black-box optimization problem. We propose a customized genetic algorithm based heuristic approach to search for a pretty efficient memory allocation strategy with minimal memory footprints.
- We conduct some experiments in several typical neural networks. The results demonstrate that Tetris achieves an average memory reduction ratio of 91.9% and 87.9% on UNNA platforms, much higher than 56.6% and 50.7% of TensorFlow, with quad-core and 16-core configurations, respectively.

2 Background and Motivation

In this section, we first overview the execution paradigm of DNNs on UNNA platforms. Then we present related memory management researches, and analyze their applicability to UNNA platforms that motivate our work. We further introduce the existing black-box optimization work.

2.1 Execution Paradigm on UNNAs

As Fig. 1 shows, cores in the same UNNA share the uniform off-chip memory space and communicate through the controller. To leverage the parallelism of neural networks and make full use of hardware resources, operations in CNNs are partitioned into several sub-operations[15,16] and sub-operations are mapped to various processing cores. And cores work collaboratively to conduct the workloads. Chen *et al.*[15] designed neural network patition strategies and migrated traditional scheduling algorithms to parallel neural network workload on multicore neural network accelerator platforms. Zhang and Zhi[16] proposed a parallel framework for multicore systems. This framework generates an effective splitting strategy by designing assistant operations, and abstracts the neural network partition into dynamic programming. Zhuang *et al.*[14] proposed to reduce memory traffic by deep fusion on neural network accelerators. In their design, several layers are fused into a composite layer, corresponding to an non-interruptable kernel. Long *et al.*[17] reduced the launch kernel overhead by fusing several operators into an enlarged kernel.

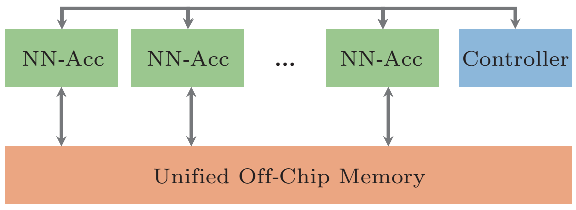


Fig. 1. Uniform memory multicore neural network accelerator architecture. NN-Acc: neural network accelerator core.

For the execution paradigm on UNNA platforms, we have the following observations.

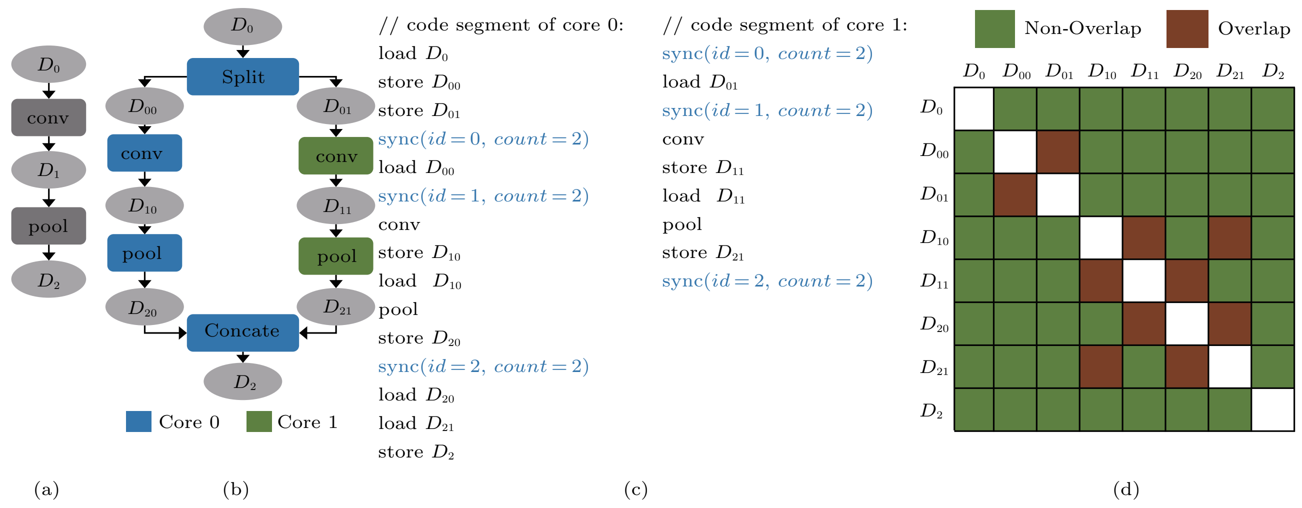
1) The relative liveness intervals cannot be directly analyzed by neural network typologies.

2) Execution flow analysis and inter-core synchronization analysis are sufficient to deduce relative liveness intervals for each tensor.

We take Fig.2 as an example to illustrate the above observations. Fig. 2(a) is the original neural network, consisting of a convolution layer, a pooling layer, and three tensors, D_0, D_1, and D_2. According to the execution paradigm, the original neural network can be transformed into the neural network in Fig.2(b). And the UNNA compiler implements all operations in the transformed neural network in a non-interruptable kernel, and the execution flow is shown in Fig.2(c). From the perspective of the network topology alone, D_{11} and D_{10} belong to different branches, and their liveness intervals are non-overlap. But in reality, they are dispatched by different cores and executed simultaneously. Therefore from the perspective of network typologies, we may get the wrong memory reuse strategies. In case of data pollution, tensors in the same kernel cannot share the memory space by graph-level liveness analysis. But by instruction-level analysis, we can perform fine-grained liveness analysis and explore potential memory sharing opportunities. The synchronization instruction has a count field and an ID field, and the count field indicates the number of cores to be synchronized and only synchronization instructions with the same ID field are identified as the same group. For example, D_{00} and D_{11} only appear in the execution flows of core 0 and core 1, respectively. And their access is separated by the second group of synchronization with the ID of 1. Therefore we can ensure that D_{00} and D_{11} can share the same memory space. As shown in Fig.2(d), we can get the memory reusability between tensors by instruction-level liveness analysis.

2.2 Memory Management Systems

Driven by the constraint of memory space, some pioneering management systems are proposed to reduce



```
// code segment of core 0:
load D0
store D00
store D01
sync(id = 0, count = 2)
load D00
sync(id = 1, count = 2)
conv
store D10
load  D10
pool
store D20
sync(id = 2, count = 2)
load D20
load D21
store D2
```

```
// code segment of core 1:
sync(id = 0, count = 2)
load D01
sync(id = 1, count = 2)
conv
store D11
load  D11
pool
store D21
sync(id = 2, count = 2)
```

	D_0	D_{00}	D_{01}	D_{10}	D_{11}	D_{20}	D_{21}	D_2
D_0	⬜	🟩	🟩	🟩	🟩	🟩	🟩	🟩
D_{00}	🟩	⬜	🟫	🟩	🟩	🟩	🟩	🟩
D_{01}	🟩	🟫	⬜	🟩	🟩	🟩	🟩	🟩
D_{10}	🟩	🟩	🟩	⬜	🟫	🟩	🟫	🟩
D_{11}	🟩	🟩	🟩	🟫	⬜	🟫	🟩	🟩
D_{20}	🟩	🟩	🟩	🟩	🟫	⬜	🟫	🟩
D_{21}	🟩	🟩	🟩	🟫	🟩	🟫	⬜	🟩
D_2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	⬜

Fig. 2. Example of the execution paradigm on UNNAs. (a) Original neural network. (b) Partitioned neural network. (c) Execution flow of each core. (d) Liveness analysis result. "conv" means the convolution layer, "split" means the split layer, and "concate" means the concatenation layer.

the neural network memory footprints on GPU platforms. These systems are divided into two categories: training scenarios oriented techniques [8, 9] and inference scenarios oriented techniques [10, 18]. For the training scenarios, vDNN [9] manages runtime memory by virtualizing the memory usage of neural networks between GPU and CPU memories. The run-time memory manager in vDNN reduces memory footprints in the training phase by swapping inactive intermediate tensors out to CPU memory and vice versa. SuperNeurons [8] profits from three memory optimization strategies, liveness analysis, unified tensor pool, and cost-aware recomputation. Specifically, liveness analysis in SuperNeurons is based on layer-wise data flow analysis. For the inference scenarios, Pisarchyk and Lee [10] proposed the shared object approach and the offset calculation approach to reduce model footprints. They explored various strategies to share memory buffers among intermediate tensors in deep neural networks. Minakova and Stefanov [18] proposed to reduce the CNN memory footprint at the cost of throughput decrease. They converted the CNN into a functionally equivalent Cyclo-Static Dataflow model, and found proper execution order by existing embedded system design tools.

Overall, these methods can be classified into three categories. The first is memory sharing by tensor liveness analysis. And the existing studies are all from the perspective of network topology. As discussed in Subsection 2.1, the coarse-grained analysis may miss some potential memory sharing opportunities on UNNA platforms. The second is memory swapping between the host memory and the device memory. Unlike the executive paradigm in GPU platforms, UNNAs treat a whole neural network as a kernel and it is non-interruptable. Therefore memory swapping is unapplicable to UNNA platforms. The last one is cost-aware recomputing. The last two strategies focus on training scenarios. And the cost-aware recomputing technology [8] is orthogonal to our work.

Besides, there are also memory management studies on neural network accelerators [19, 20]. FP-DNN [19] reuses the DRAM space by allocating tensors whose life spans do not intersect into the same physical buffer. FP-DNN formulates the data buffer reuse problem as a graph coloring problem and solves it by coloring the interval graph with the minimum number of distinct colors and assigning tensors with the same color to the same physical buffer. LCMM [20] is a layer conscious memory management framework for FPGA-based DNN hardware accelerators. LCMM exploits the layer diversity and the disjoint lifespan information of memory buffers to utilize on-chip memory to improve the performance of memory bound layers and thus the entire performance of DNNs. LCMM leverages a customized memory allocation algorithm and buffer sharing and buffer splitting techniques to utilize on-chip memory and reduce off-chip memory traffic. Different from these frameworks, the design philosophy in our work is to decouple the memory access behavior and off-chip memory management, and we only focus on the off-chip memory footprint optimization. We admit that the off-chip memory and on-chip memory co-optimization is significant for both memory footprint and performance, and we leave it as our future study.

2.3 Black-Box Optimization

Searching a superior tensor permutation from the whole permutation space can be viewed as a high-dimensional black-box optimization problem. From the perspective of generalizability, there are several researches emphasizing the black-box optimization problems. One way is training an approximate regressor through sampling and optimizing the fitted regressor instead. Bayesian Optimization (BO) [21] and its variants [22, 23] fall into this route. For high-dimensional problems, BO requires an enormous amount of samples to fit the model. And BO overemphasizes the boundary of the search space which is less efficient. Another way to optimize black-box problems is to partition the exploration space and model local promising spaces. LA-MCTS [24] is a typical representation. LA-MCTS serves as a meta-level algorithm that recursively learns space partition in a hierarchical manner and uses existing black-box optimizers as its local models. In each iteration, LA-MCTS constructs a Monte Carlo search tree, selects the subspace following the upper confidence bound for adaptive exploration, and then uses the local model to propose new samples. The evolutionary algorithm (EA) is also for high dimensional black-box optimizations. CMA-ES [25] is one of the most powerful stochastic numerical optimizers. It uses co-variance matrix adaption to propose new samples with quadratic intrinsic time and space complexity. But all these optimization strategies aim to generic black-box optimization and they totally inquire the block-box system to score samples without any additional information. Compared with these strategies, Tetris leverages superior sub-sequences to construct more efficient samples.

2.4 Motivation: Heuristic Static Memory Management

For simplicity and efficiency, mainstream deep learning frameworks like TensorFlow[26], PyTorch[27] and Caffe[28] are broadly used to deploy neural network applications. Since neural network architectures are going deeper and wider, it is extremely challenging to relieve the constraints of limited memory capacity for UNNA platforms. Rethinking the process of liveness analysis, there are more opportunities to reuse the memory space with fine-grained instruction-level analysis. Since the process of UNNAs' kernels is uninterruptible and the corresponding instructions are generated in the compilation phase, using a static method for liveness analysis is feasible. Meanwhile, we notice that, given the tensor sequence and the constraints of whether every two tensors can use the same physical space, the orders of tensor space allocation can affect the memory reuse ratio by a greedy algorithm. As shown in Fig.3, let us suppose there are three tensors D_0, D_1 and D_2 with the size of 5, 5 and 6, respectively, in which D_0 and D_2 can share the same space while D_0 and D_1, D_1 and D_2 cannot. The memory sizes used with the order of D_0, D_1, D_2 and D_0, D_2, D_1 are 16 and 11 as shown in scheme 1 and scheme 2 of Fig. 3 respectively. Therefore the choice of permutation affects the efficiency of memory allocation. While all permutations are too large to search, we notice that some partial sub-sequences could be reserved as superior partial solutions. Thus we adopt a genetic algorithm to explore the search space to get a superior tensor permutation. We propose Tetris to manage memory resources on UNNA platforms.

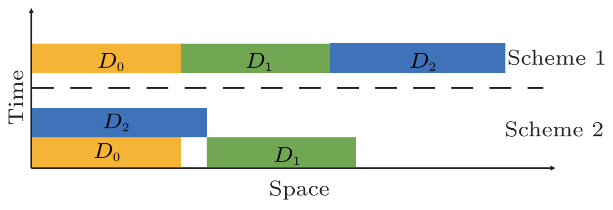


Fig.3. Allocation with different permutations.

3 Framework Overview

Fig.4 provides an overview of Tetris. Tetris retrieves multicore execution information from instructions generated by UNNA compilers and allocates the physical space for each tensor. As shown in Fig.4, the workflow of Tetris consists of two parts: the front-end and the back-end.

1) The front-end takes multicore instructions as input, extracts basic execution flows and synchronization information among cores, and transforms them into a relation graph. Tetris relies on the relation graph to analyze relative liveness intervals for each tensor. Before analyzing the reusabilities among tensors, Tetris removes redundant nodes and edges in the relation graph to simplify the complexity of analysis. Finally, the front-end analyzes reusabilities among tensors based on the connectivity of the relation graph and generates a symmetric boolean conflict matrix indicating whether two tensors can share the same memory space.

2) The back-end allocates the memory space from a memory pool, and tries to find the minimal requirement of the memory pool and each tensor's offset in

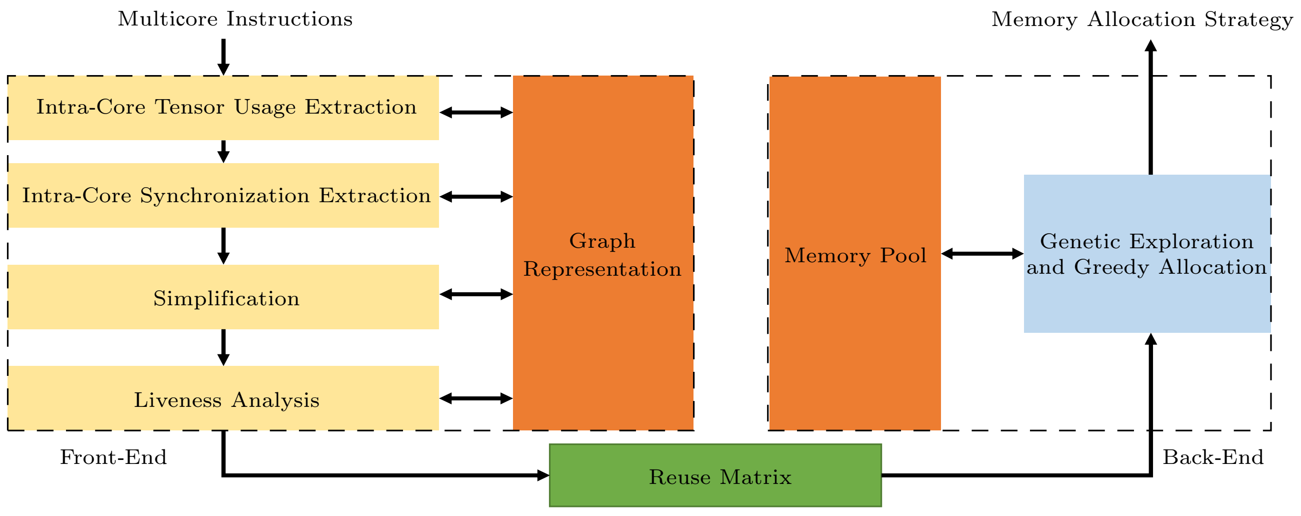

Multicore Instructions
Memory Allocation Strategy
Intra-Core Tensor Usage Extraction
Intra-Core Synchronization Extraction
Simplification
Liveness Analysis
Graph Representation
Memory Pool
Genetic Exploration and Greedy Allocation
Front-End
Reuse Matrix
Back-End

Fig.4. Overview of Tetris.

the memory pool. Under constraints of the conflict matrix, for every permutation of tensors, Tetris achieves a specific memory allocation solution. In the huge permutation space, Tetris explores with a heuristic genetic algorithm[29]. Then, Tetris finds an optimal permutation with a minimal memory pool size.

4 Front-End Conflict Matrix Generation

This section describes the front-end of Tetris. The front-end mainly comprises three parts: relation graph generation, simplification, and liveness analysis. Relation graph generation consists of intra-core tensor usage extraction and inter-core synchronization extraction, representing tensor usage information in a relation graph. Simplification is used to remove unnecessary nodes in the relation graph and liveness analysis finally generates a conflict matrix to indicate whether two tensors can share the same memory space.

4.1 Relation Graph Generation

Intra-Core Tensor Usage Extraction. A relation graph is used to represent the chronological access order of each tensor, in which a node has an attribute with a list of tensors, and an edge represents the relative access order of tensors between adjacent nodes. Note that tensors in the same node cannot reuse the same memory space and the tensor list in an auxiliary node may be empty.

Tetris separates each core's instructions from the whole instructions by physical core ID. A processor core's instruction flow consists of three basic structures: sequential structure, branch structure, and loop structure. Tetris recognizes corresponding structures by matching specific instructions. Fig.5 shows the transformations of these three basic structures. For the sequential structure in Fig.5(a), Tetris transforms it into a sequential relation graph with a tensor for each node, while for the branch structure in Fig.5(b) and the loop structure in Fig.5(c), auxiliary nodes without tensors are added in the instruction flows' entry and exit points.

Local tensors in loop structures are carefully handled. As the instructions shown in Fig.6(a), D_2 and D_3 are initialized in the loop body and they can share the same memory space. To represent local tensors, we propose the hierarchy of tensor initialization which means the level of tensor initialization is conducted in a nested structure. By analyzing the hierarchy of tensor initialization, instructions in Fig.6(a) are marked with the flags of their levels. When conducting hierarchy n in the process of generating a relation graph, nodes with hierarchies larger than n should be excluded from the body, and an additional serial partial graph with nodes whose hierarchies are no more than $n+1$ is added in the periphery of the body. In this way, Tetris generates a relation graph for single-core instructions.

Inter-Core Synchronization Extraction. In UNNA platforms, processor cores run neural network workload collaboratively. They synchronize with others with barrier instructions. Barrier instructions consist of two functions: *barrier_arrive* and *barrier_sync*. A barrier controller is used to record the status of the UNNA

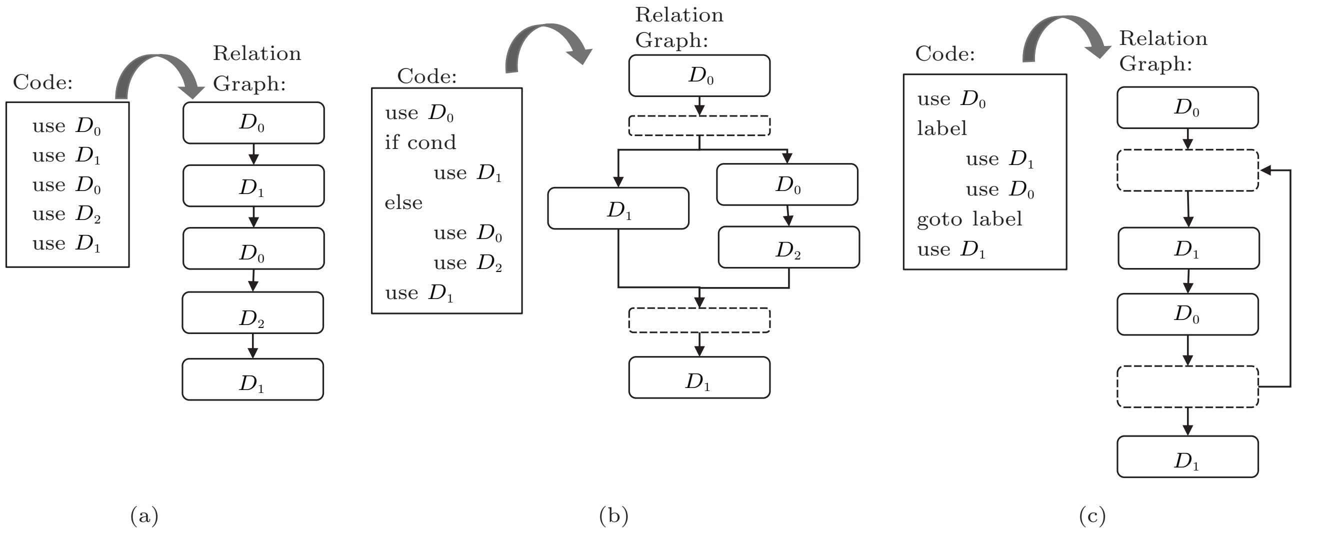
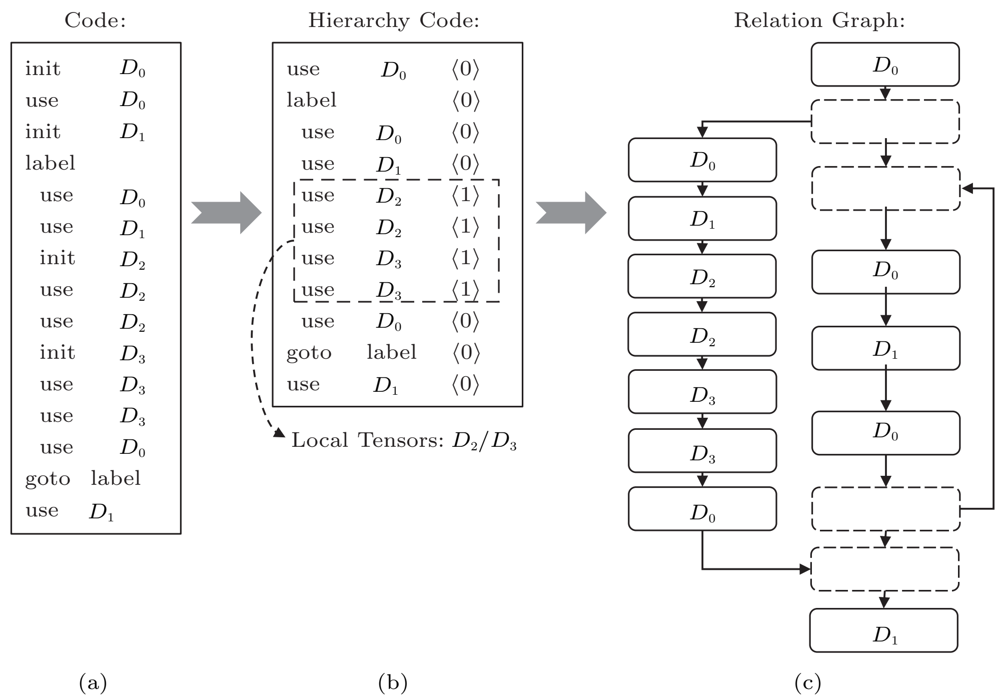


Fig.5. Transformations of three basic structures for instruction flows. (a) Sequential structure. (b) Branch structure. (c) Loop structure.



```
init    D0
use     D0
init    D1
label
  use   D0
  use   D1
  init  D2
  use   D2
  use   D2
  init  D3
  use   D3
  use   D3
  use   D0
goto  label
use    D1
```

```
use        D0     ⟨0⟩
label             ⟨0⟩
  use     D0      ⟨0⟩
  use     D1      ⟨0⟩
  use     D2      ⟨1⟩
  use     D2      ⟨1⟩
  use     D3      ⟨1⟩
  use     D3      ⟨1⟩
  use     D0      ⟨0⟩
goto     label    ⟨0⟩
use       D1      ⟨0⟩
```

Fig.6. Transformation of the loop structure with local tensors. (a) Code with the loop structure and local tensors. (b) Hierarchy code with level flags. (c) Transformed relation graph.

system using a tuple (*id*, *sum*, *counter*, *core id*, ...). Cores with *barrier_arrive* send a message (*id*, *count*) to the barrier controller and continue to execute, while cores with *barrier_sync* instructions send messages to the barrier controller and wait for the response from the barrier controller. When the barrier controller receives a message, it increases the counter variable (*counter*) in the tuple. And the barrier controller responds to cores which send *barrier_sync* instructions. Then, cores which send *barrier_sync* continue to run. Such a mechanism ensures processor cores on a UNNA platform work in concert efficiently.

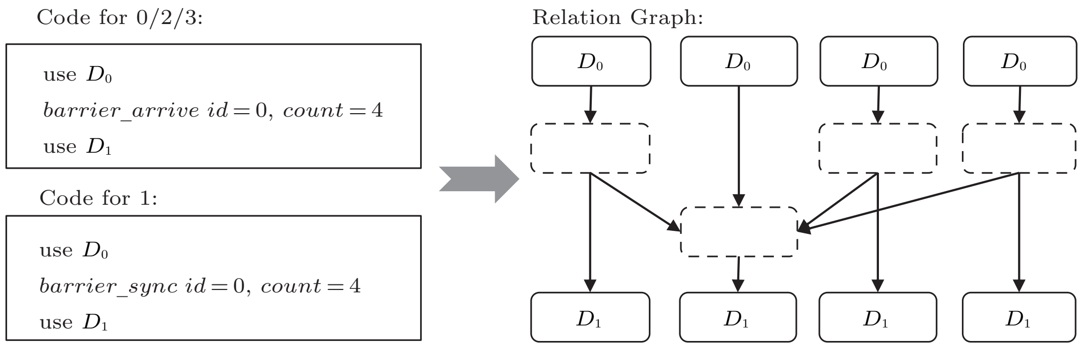
As shown in Fig.7, for a group of synchronizations, Tetris creates an auxiliary empty node for each core and adds edges from all created auxiliary nodes to auxiliary nodes of cores with *barrier_sync* instructions.

4.2 Relation Graph Simplification

Firstly, we introduce Theorem 1 to connect the relation graph with the reusability of every two tensors.

Theorem 1. *For any two tensors, D_i and D_j, that are reusable if and only if that for any node N_{im} which includes D_i and any node N_{jn} which includes D_j, there exists at least one path between N_{im} and N_{jn} and the directions of all paths are the same.*

Proof. *Necessity.* If D_i and D_j are reusable, it means the liveness interval intersection of D_i and D_j



```
use D0
barrier_arrive id = 0, count = 4
use D1
```

```
use D0
barrier_sync id = 0, count = 4
use D1
```

Fig.7. Transformations of synchronization information among different cores.

is empty, that is, all access to D_i must be ahead of D_j, and vice versa. Since the tensor access orders in disconnected nodes are undeterministic, all nodes consisting of D_i and D_j in the relation graph must have a path. Meanwhile, if these exist a path from N_{ia} to N_{jb} and a path from N_{jc} to N_{id}, D_i in N_{ia} is accessed before D_j in N_{jb} and D_j in N_{jd} is accessed before D_i in N_{ic}. And the liveness intervals of D_i and D_j intersect. Therefore the directions of all paths should be the same.

Sufficiency. If all nodes including D_i have at least a path to nodes including D_j, and there is no path from nodes including D_j to nodes including D_i, then the access to D_i is ahead of the access to D_j, and vice versa. Therefore the liveness intervals of D_i and D_j do not overlap, and they can reuse the same space. □

Based on Theorem 1, we summarize the following three operations to simplify the relation graph, and transform it into a directed acyclic graph.

Cycle Elimination. Since nodes in a cycle have the same connectivity with nodes out of the cycle, and tensors belonging to nodes in the cycle cannot reuse the same memory space with each other, we represent cycles in the relation graph with an auxiliary node. As shown in Fig.8, tensors in the replaced auxiliary node of the transformed relation graph are the union of the eliminated cycles. For edges whose start nodes are out of the cycle and end nodes are in the cycle, we replace the end nodes with the auxiliary node. Analogously, for edges whose start nodes are in the cycle and end nodes are out of the cycle, we replace the start nodes with the auxiliary node.

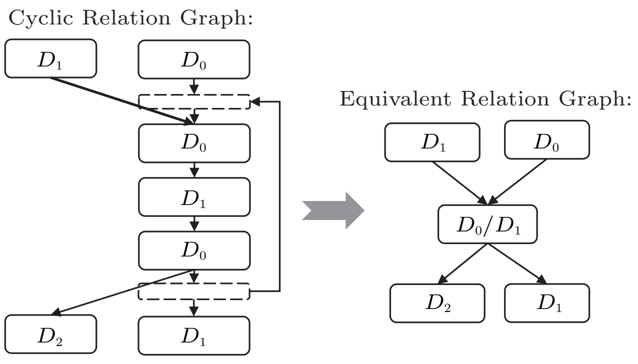


Fig.8. Elimination of cycles in the relation graph.

Redundant Tensor Attributes Elimination. If a tensor appears more than twice on any path, removing tensors other than the start node and the end node can still keep the reusability of the original graph. As shown in the transformation ① of Fig.9, D_1 in intermediate nodes can be removed.

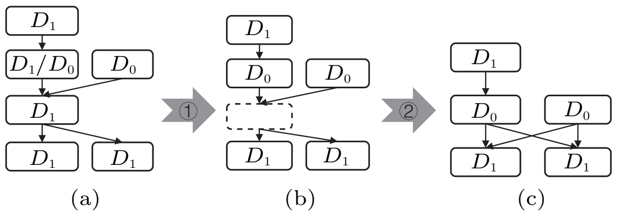


Fig. 9. Elimination of redundant attributes and nodes in the relation graph. (a) Original graph. (b) Redundant attributes elimination. (c) Redundant nodes elimination.

Empty Node Elimination. Nodes with an empty tensor list can be removed. Meanwhile, for edges whose start nodes are empty nodes, we change the start nodes to empty nodes' precursors. For edges whose end nodes are empty nodes, we change the end nodes to empty nodes' successors.

As shown in the transformation ② of Fig. 9, the empty node is eliminated, edges related to this node could be replaced, and the relation graph in Fig.9(b) is transformed into that in Fig.9(c).

4.3 Liveness Analysis

At this stage, Tetris analyzes the simplified relation graph, detects the connectivity of any two nodes, and checks whether the connectivity of any two tensors satisfies Theorem 1. Then, Tetris uses a boolean conflict matrix to represent whether two tensors can share the same memory space. For a workload with N tensors used, the conflict matrix $\boldsymbol{C}_{N\times N}$ is defined as follows:

$$C_{i\times j}=\begin{cases}\text{true}, & \text{if } D_i \text{ and } D_j \text{ are reusable},\\ \text{false}, & \text{otherwise}.\end{cases}$$

5 Back-End Memory Allocation

Tetris uses the First-Fit (FF) algorithm [30] to allocate the memory space. As shown in Algorithm 1, given a permutation of the tensor sequence, the FF algorithm reserves a memory pool with initial size 0. For a tensor to allocate, the space in the memory pool is divided into two kinds of fragments, unreusable and reusable slices. The FF algorithm scans from the beginning to the end to find the first reusable slice with a size no less than the tensor, and allocates the front part to the tensor. If no suitable slice is found, the FF algorithm enlarges the memory pool and allocates the tail space to the tensor. The FF algorithm repeats the previous steps until all tensors are allocated. Finally, the FF algorithm gets the minimal size needed by the memory pool and the offsets in the memory pool as tensors' addresses.

Theorem 2. *For every memory allocation solution, there exists an equivalent or better solution by using the*

FF algorithm with a specific tensor permutation.

Algorithm 1. First-Fit Algorithm

```
Input: $data, \boldsymbol{C}, N$
Output: $addr, max\_size$
$max\_size \Leftarrow 0$, $n \Leftarrow 0$;
while $n < N$ do
    $addr[n] = max\_size$;
    /* partition the allocated memory into blocks
    according to whether they could be reused by $data[n]$
    */
    $blocks = sort\_by\_addr(data, n, \boldsymbol{C})$;
    foreach $block$ in $blocks$ do
        if $is\_valid(block, data[n])$ then
            $addr[n] = block.start$;
            break;
        end
    end
    $max\_size = \max(max\_size, addr[n] + data[n].size)$;
    $n \Leftarrow n + 1$;
end
```

Proof. For a memory allocation solution S with tensor list $(D_1, D_2, \ldots, D_N)$, sorting each tensor's address in ascending order generates a permutation of these tensors $(D'_1, D'_2, \ldots, D'_N)$. Then we prove that in the memory allocation solution S' of the FF algorithm with permutation $(D'_1, D'_2, \ldots, D'_N)$, the address $addr_{FF_i}$ of each tensor D'_i is no more than $addr_{S_i}$ in S.

1) We assume the start address of the memory pool is 0. Then for D'_0, $addr_{FF_0} = 0$ and $addr_{S_0} \geqslant 0$.

2) If the assumption holds when allocating the first k tensors, $D'_1, D'_2, \ldots, D'_k$, e.g.,

$$addr_{FF_i} \leqslant addr_{S_i}, \forall 1 \leqslant i \leqslant k.$$

We mark tensors that are not reusable with D'_{k+1} in $(D'_1, D'_2, \ldots, D'_k)$ as $(D_{r1}, D_{r2}, \ldots, D_{rm})$. We assume $end_{S_i} = addr_{S_i} + size_{S_i}$, and then the following conditions hold:

$$addr_{S_{k+1}} \geqslant \max(end_{S_{r1}}, end_{S_{r2}}, \ldots, end_{S_{rm}}),$$
$$addr_{FF_{k+1}} \leqslant \max(end_{FF_{r1}}, end_{FF_{r2}}, \ldots, end_{FF_{rm}}).$$

Since $\max(addr_{FF_{r1}}, addr_{FF_{r2}}, \ldots, addr_{FF_{rm}}) \leqslant \max(addr_{S_{r1}}, addr_{S_{r2}}, \ldots, addr_{S_{rm}})$ and $size_{S_i} = size_{FF_i}$, $addr_{FF_{k+1}} \leqslant addr_{S_{k+1}}$.

3) By mathematical induction, for each tensor D'_i, $addr_{FF_i} \leqslant addr_{S_i}$.

We mark the size of the memory pool is P, then $P = \max(addr(D_i) + size(D_i))$, and thus $P_{FF} \leqslant P_S$ and Theorem 2 holds. □

Theorem 2 shows that the FF algorithm can find the best solution by enumerating all permutations. The permutations of tensors affect the results of the FF algorithm. The number of n tensors' permutations is $n!$ and these permutations compose the whole searching space. But the searching space is too large to enumerate. The back-end leverages a genetic algorithm to explore the huge searching space heuristically and selects an optimal solution.

Fig. 10 shows the architecture of the genetic algorithm. A tensor permutation is encoded as a chromosome to represent an individual in the genetic algorithm, and parameters in the chromosome are called genes. Many individuals form a population. At the beginning of processing, an initialized population is generated randomly. In the process of evolution, we design a function to evaluate the fitness of individuals. Individuals with higher fitness values have greater opportunities for reproduction. The genetic algorithm uses the crossover operation to recombine two individuals' genes and generate new individuals. Some genes may mutate with a pretty low probability, and mutation is pretty useful to jump out of the local optimum. Details of these phases implemented in Tetris are shown as follows.

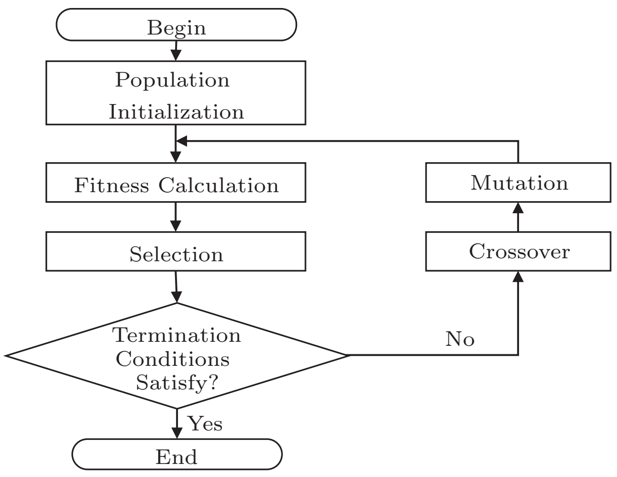


Fig.10. Architecture of the genetic algorithm.

Individual Representation. Each individual is a permutation of tensors to allocate. For example, if there are three tensors, the permutation could be $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$, $(3,2,1)$, where each item means a permutation of tensors to allocate in the FF algorithm.

Population Initialization. We initialize the population by randomly permutating tensors.

Fitness Function. We use the FF algorithm as the fitness function (see Algorithm 1). In a population with m individuals $p_1, p_2, ..., p_m$, for a specific individual p_i, the FF algorithm uses the chromosome as input and calculates the size s_i of the memory pool needed. Then the fitness function $f(p_i)$ is $f(p_i) = 1/s_i$. Individuals with higher fitness values have greater possibilities to pass on their genes to the next generation.

Crossover Strategy. We randomly select two individuals as parents to reproduce an individual. As shown in Fig.11(a) and Fig.11(b), we select a continuous sequence A from the first parent, remove elements of A in the second parent to get B, and then insert A into a random position of B to reproduce a child. A partial sequence in the evolved chromosome is highly likely to have a high memory reuse ratio, and we should pass such high-quality partial permutations onto the offspring.

Mutation Strategy. As shown in Fig.11(c), we select an individual randomly, choose an element from its chromosome, and then move the element to another position to generate a new chromosome. The newly-generated chromosome is a mutated individual.

Selection Strategy. We select the best several individuals based on their fitness values and pass their chromosomes to the next generation.

6 Experiments

In this section, we introduce our experimental setup, and present the results of our experimental studies that evaluate the performance of each component in Tetris.

6.1 Experimental Setup

UNNA Platform. In the experiment, we use a multicore architecture based on Cambricon-X[31] as the hardware platform to run CNNs. Each core in Cambricon-X has 16 processing engines, a 1 MB Synapse Buffer, and a 512 KB Neuron Buffer. And the Cambricon-X platform has a 4 GB DRAM to save model data, input/output data, and intermediate data. The core number in this platform is configurable. We use the upgraded version of DLPlib[32] as the neural network compiler in Cambricon-X. The upgraded version of DLPlib provides API to efficiently get the generated instructions and supports multicore hardware architectures. And the multicore system supports running a neural network collaboratively with various kinds of neural network partition strategies.

Implementation and Hyper-Parameters. We implement Tetris in C++ language, and the genetic algorithm is configured with the maximal number of generations of 100, the maximal population size of 50, the crossover rate of 90%, and the mutation rate of 10%.

Baselines. To evaluate the memory sharing efficiency of Tetris, we evaluate experiments from two hands, i.e., a memory reuse ratio and a convergence speed. To illustrate that Tetris can reduce memory capacity constraint significantly, we compare the memory reuse ratio of the genetic algorithm in Tetris and two other algorithms, i.e., the memory allocation with the First-Fit algorithm as shown in Algorithm 1 and the best-fit with coalescing algorithm (BFC) implemented in TensorFlow[26]. Meanwhile, we take the memory capacity without memory reuse as the baseline, for which the memory capacity equals the sum of all intermediate tensors. To show the efficiency of the exploration strategy used in the genetic algorithm, we take several state-of-the-art black-box optimization algorithms and the random search algorithm as baselines and compare the convergence speed. The black-box optimization algorithms include Bayesian Optimization (BO)[22], LA-MCTS[24] and CMA-ES[25]. For fairness, each algorithm is configured with the same number of permutations in each iteration. We choose the result with the minimal memory pool size as the final solution.

Performance Metrics. We take the memory reduction ratio (MRR) as the memory reuse evaluation met-

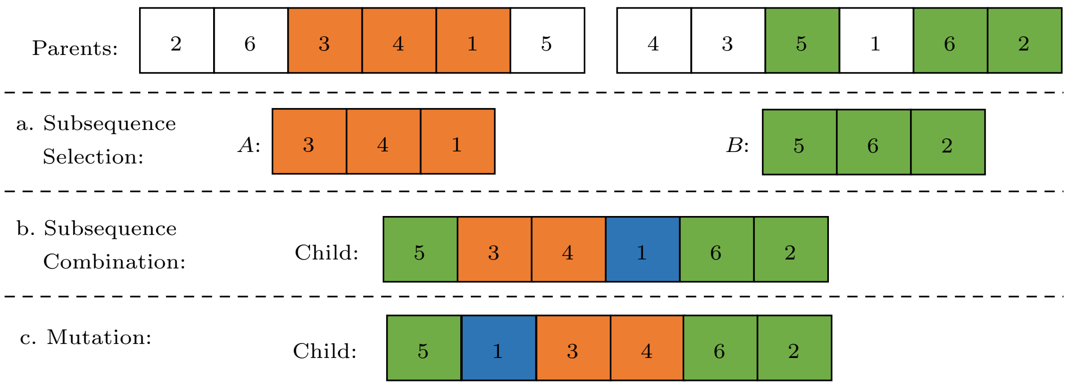


Fig.11. New samples generation strategies in the genetic algorithm.

ric. For a specific task, if the memory space allocated by Tetris is s bytes, and the memory space without memory reuse is n bytes, then we have $MRR_{\text{Tetris}} = (n - s)/n$.

Benchmark. We use several typical neural network models commonly used by previous studies [8–10] as our benchmarks, including ResNet50, MobileNetV1, MobileNetV2, InceptionV3, GoogLeNet, and DenseNet. We run these networks on the Cambricon platforms configured with single-core, quad-core and 16-core, respectively. Through these experiments, we evaluate the memory reuse ratio and the convergence speed of Tetris to evaluate the efficiency of Tetris.

6.2 Components Evaluations

6.2.1 *Relation Graph Simplification*

To facilitate the liveness analysis of tensors, we design several operations to simplify the relation graph. Fig.12 shows the speedup of the liveness analysis process by graph simplification. The speedup on the single-core, quad-core and 16-core platforms increases with the the number of processors. As the number of processors increases on the UNNA platform, the synchronization relationships among cores and the partitioned neural network topologies are more complex. And there are more opportunities to optimize original relation graphs. Overall, on the 16-core Cambricon-X platform, the graph simplification accelerates the liveness analysis procedure by up to 2.5x.

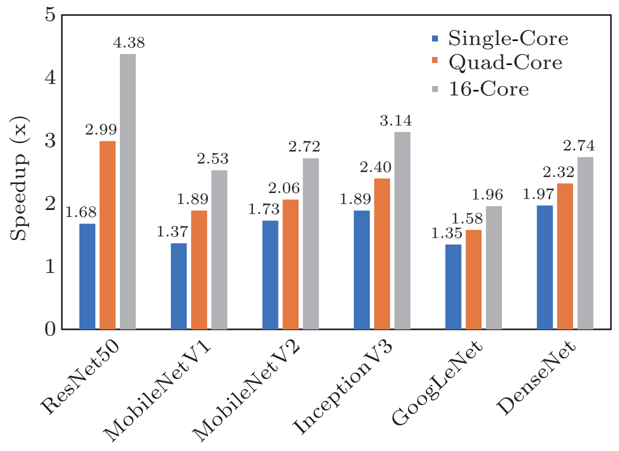


Fig.12. Compilation time optimization by graph simplification.

6.2.2 *Liveness Analysis*

To evaluate the efficiency of our proposed instruction-level tensor liveness analysis, we take the layer-wise liveness analysis as the baseline. For tensors in the same fused subgraph, we have no prior knowledge about behaviors of accessing tensors by the layer-wise liveness analysis. To ensure the correctness, we treat that tensors in a fused subgraph use different memory spaces. For instruction-level liveness analysis and layer-wise analysis, we take the First-Fit algorithm to allocate the DRAM space with the same tensor permutation. Fig.13 shows the normalized memory space for six typical neural networks on Cambricon-X platform with 1 core, 4 cores and 16 cores, respectively. For the single-core platform in Fig. 13(a), the compiler stack does not parition neural network models and the instruction-level and the layer-wise liveness analysis achieve the same results. But for the quad-core and 16-core platforms in Fig.13(b) and Fig.13(c) respectively, the average memory reduction ratios by instruction-layer liveness analysis are 7.57% and 6.85% higher than those by layer-wise liveness analysis, respectively. Since instruction-level liveness analysis has a smaller granularity, it has the potential to find memory reuse opportunities.

6.2.3 *Heuristic Memory Allocation*

The back-end of Tetris uses the customized genetic algorithm (GA) to search tensor permutations. Given a specific permutation, Tetris generates a memory allocation strategy deterministically by the FF algorithm. We take the FF algorithm and the BFC algorithm as references for comparison. We calculate the MRRs of these algorithms relative to the baseline without memory reuse on all evaluated neural networks. The details for our configurations and results are shown in Table 1.

The average MRRs in the Cambricon-X platform with a single core for the genetic algorithm, the FF algorithm, and the BFC algorithm are 91.1%, 89.8%, 89.9%, respectively. Therefore all these algorithms for neural networks running with a single core achieve high MRRs and the genetic algorithm implemented in Tetris achieves slightly higher MRRs. For GoogLeNet with a single-core, and InceptionV3 with a single-core Cambricon-X platform, MRRs of all three algorithms are 84.5%, since they all achieve the optimal result.

In the quad-core and 16-core Cambricon-X platforms, the average MRRs for the BFC algorithm are 56.6% and 50.7%, respectively, whereas these for genetic algorithms are 91.9% and 87.9%, respectively. Our proposed method prompts MRR by 35.3% and 37.2%, respectively. MRR for GoogLeNet with the BFC algorithm running on the 16-core Cambricon-X platform is 27.7%, far less than 79.0% achieved with

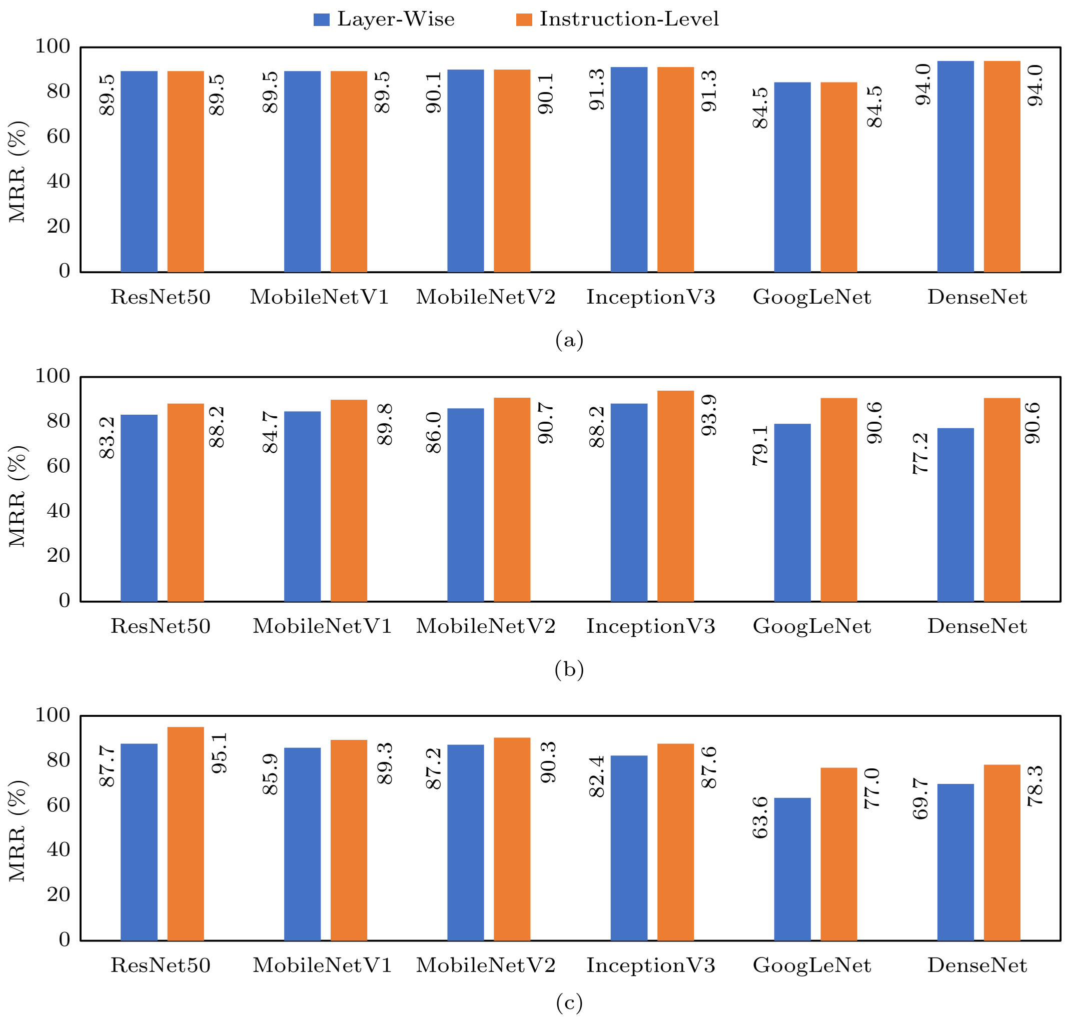


Fig.13. MRR comparison of layer-wise liveness analysis and instruction-level liveness analysis on Cambricon-X platforms with different numbers of cores. (a) Single-core Cambricon-x platform. (b) Quad-core Cambricon-X platform. (c) 16-core Cambricon-X platform.

Table 1. Memory Footprint Comparisons (MRR) for Different Algorithms in Cambricon-X Platform with Different Numbers of Cores

Network	Single-Core			Quad-Core			16-Core		
	GA	FF	BFC	GA	FF	BFC	GA	FF	BFC
ResNet50	0.913	0.895	0.895	0.892	0.882	0.730	0.952	0.951	0.659
MobileNetV1	0.914	0.895	0.895	0.930	0.898	0.650	0.925	0.893	0.701
MobileNetV2	0.910	0.901	0.901	0.922	0.907	0.729	0.919	0.903	0.723
InceptionV3	0.932	0.913	0.917	0.939	0.939	0.499	0.884	0.876	0.353
GoogLeNet	0.845	0.845	0.845	0.906	0.906	0.457	0.790	0.770	0.277
DensetNet	0.951	0.940	0.940	0.924	0.906	0.331	0.804	0.783	0.330
Average	0.911	0.898	0.899	0.919	0.906	0.566	0.879	0.863	0.507

the genetic algorithm implemented in Tetris. Therefore for neural networks running on UNNA platforms, Tetris achieves a much higher memory reuse ratio than the BFC algorithm implemented in TensorFlow. Meanwhile, compared with the FF algorithm, the genetic algorithm achieves higher MRRs, which shows that orders in which intermediate tensors are allocated to affect memory reuse ratio, and exploring permutation space is quite necessary.

In general, Tetris with the genetic algorithm for these six neural networks achieves the memory reduction ratio of 79.0%–95.2%, which alleviates the DRAM capacity demand for deploying neural network models in UNNA platforms.

6.2.4 A Holistic Study

Tetris uses instruction-level liveness analysis to find the reusability among tensors and the customized genetic algorithm to search for an appropriate permutation for allocating the memory space. We take the strategy that uses layer-wise liveness analysis and the First-Fit memory allocation (LW+FF) as the baseline. We futher make a comparison to depict the efficiency of instruction-level (IL) liveness analysis and the customized genetic algorithm (GA) for searching an appropriate permutation. Fig.14 illustrates the optimization effects intuitively. The instruction-level liveness analysis can find more opportunities with a fine-grained granularity for the tensors between processing cores and the same kernel. The strategy with the instruction-level liveness analysis and the first-fit memory allocation (IL+FF) reduces 24.1%–60.5% memory footprints. By exploring the permutation of the tensor allocation sequence, (IL+GA) further reduces memory footprints by 34.3%–60.8%. The genetic searching process can be seen as an exploration to reduce memory fragmentation.

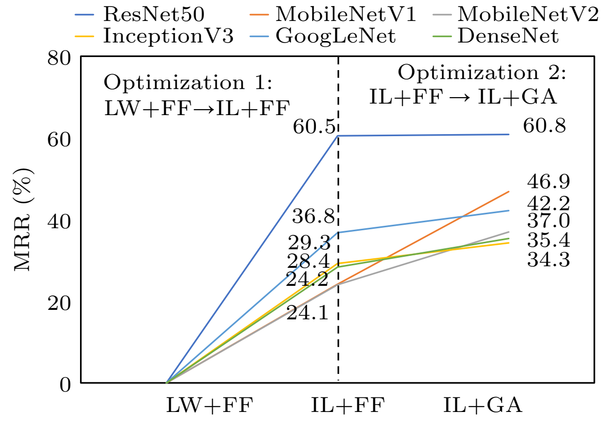


Fig.14. Holistic study on memory footprint reduction optimizations.

6.3 Evaluation on Original Neural Networks

Since layer-by-layer execution is also a common execution paradigm for UNNAs [19, 20], we experiment with this execution method to test the versatility of Tetris. We take the First-Fit algorithm as the baseline, and present the memory footprint reduction of Tetris. As shown in Fig.15, Tetris outperforms the First-Fit algorithm in five out of six neural networks with an average of 13.57% memory footprint reduction. The memory footprint reduction is due to the fact that the tensor permutation explored by Tetris can effectively reduce memory fragmentation.

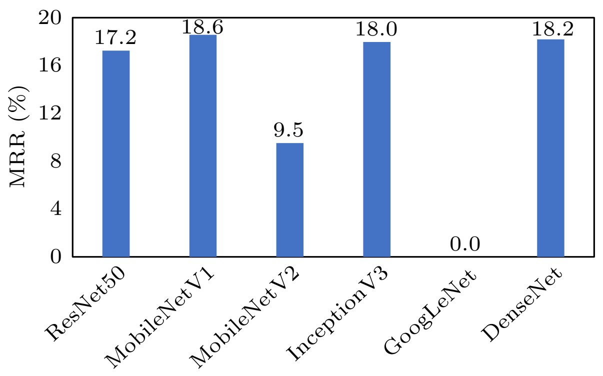


Fig.15. Memory footprint comparison for Tetris on original neural networks.

6.4 Convergence Comparison

To evaluate the efficiency of the customized genetic algorithm in Tetris (GA), we compare GA with the state-of-the-art baselines from different algorithm categories ranging from MCTS (LA-MCTS), Evolutionary Algorithm (EA), and Bayesian Optimization (BO). We use the open-source implementations [24] as baselines and ensure the same number of samples in each iteration for fairness. As shown in Fig.16, in a 16-core Cambricon-X platform, we compute the MRRs of six typical neural network models for each iteration. We have the following observations.

• In total, GA consistently outperforms the other optimization algorithms in the memory reduction ratio. Four in six neural networks GA achieves a higher memory reduction ratio. For MobileNetV1 and MobileNetV2, the optimal solutions are pretty simple and all these exploration algorithms find optimal strategies with same memory reduction ratios.

• In the huge permutation space, several permutations correspond to the same memory reduction ratio value. And the searching space has several long and flat valleys, which makes optimization hard. In these exploration curves, there exist several jumping points which represent the jump from one valley to another.

• GA performs better in rapidity of convergence. In Fig.16, all the other curves are below the curves of GA for all models, which indicates the customized genetic algorithm in Tetris converges faster than the other algorithms. The other exploration algorithms simply treat the exploration problem as a black-box optimization problem. But the customized genetic algorithm exploits partial superior results to generate high-quality samples efficiently. In case of trapping into local optimum, the customized genetic algorithm jumps out of local optimum by introducing random partial samples in the mutation phase.

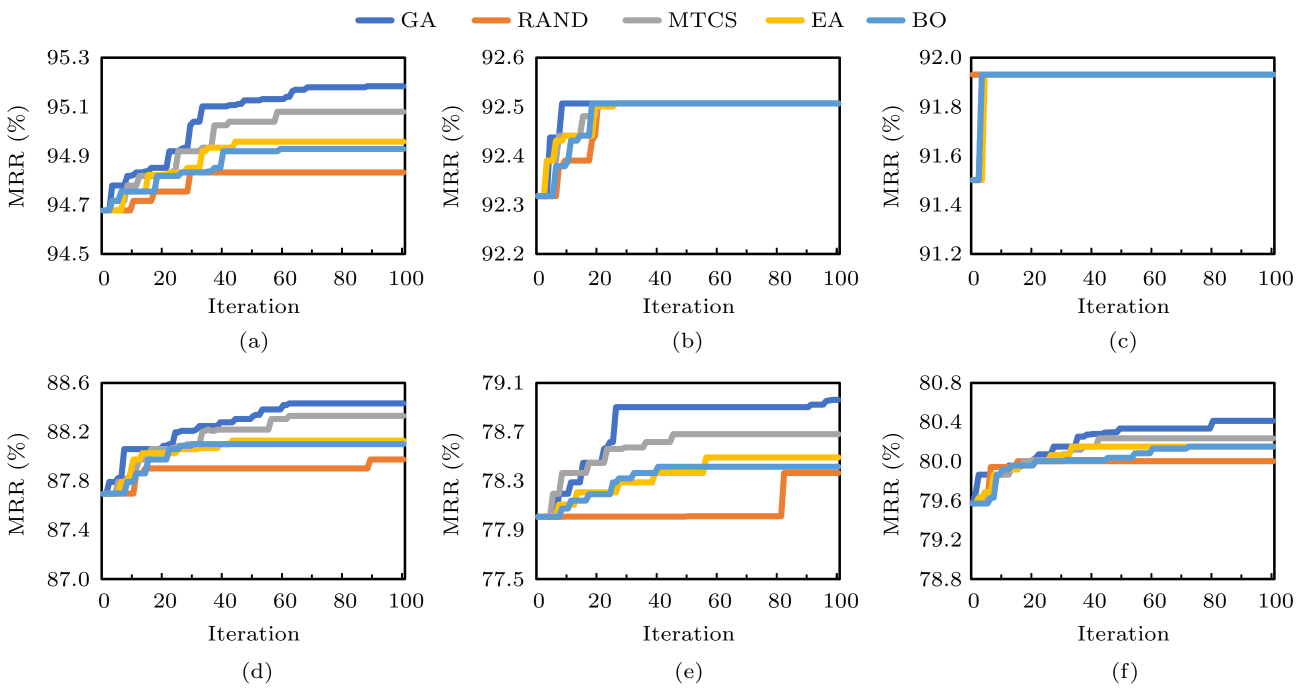


Fig. 16. Convergence comparisons on a 16-core Cambricon-X platform. (a) ResNet50. (b) MobileNetV1. (c) MobileNetV2. (d) InceptionV3. (e) GoogLeNet. (f) DenseNet.

6.5 Performance Evaluation

We further evaluate the impact of Tetris on performance. We treat layer-wise liveness analysis and the First-Fit allocation algorithm as the baseline, and make a comparison between the baseline and Tetris. The result shows that Tetris has a negligible impact on performance, with the performance change in the range of −1.54%–3.17%. The design philosophy of Tetris is to decouple performance optimization in the compilation procedure and off-chip memory management. And we conduct optimizations like on-chip memory reuse and off-chip memory traffic reduction in the compilation procedure. Thus, we mainly focus on memory footprints reduction and treat the performance optimization as an orthogonal work.

7 Conclusions

Neural network architectures become deeper and wider, and the connections between operations become denser. Memory capacity becomes the bottleneck in UNNA platforms to deploy neural network applications, especially in edge devices with small memory capacity. In this paper, we proposed an automatic memory management framework called Tetris to improve the DRAM reuse ratio. Tetris uses instruction-level information to make fine-grained liveness analysis, and uses the genetic algorithm to allocate the memory space heuristically. Experiments on typical neural networks showed that Tetris promotes memory reuse ratio dramatically and the heuristic method converges fast. Tetris achieves an average memory reduction ratio of 91.9% and 87.9% for quad-core and 16-core Cambricon-X platforms, respectively.

The instruction-level liveness analysis and the heuristic memory allocation algorithm in Tetris alleviate memory space bottleneck effectively. And in the future, we will focus on the off-chip memory and on-chip memory co-optimization, and take both the memory footprint and performance into consideration.

References

[1] He K, Zhang X, Ren S, Sun J. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proc. the 2015 IEEE International Conference on Computer Vision*, Dec. 2015, pp:1026-1034. DOI: 10.1109/ICCV.2015.123.

[2] Devlin J, Chang M W, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805, 2018. https://arxiv.org/abs/1810.04805, April 2021.

[3] Silver D, Huang A, Maddison C J *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016, 529(7587): 484-489. DOI: 10.1038/nature16961.

[4] Silver D, Schrittwieser J, Simonyan K *et al.* Mastering the game of Go without human knowledge. *Nature*, 2017, 550(7676): 354-259. DOI: 10.1038/nature24270.

[5] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. In *Proc. the 25th International Conference on Neural Information Processing Systems*, Dec. 2012, pp.1097-1105.

[6] Xie S, Girshick R, Dollár P, Tu Z, He K. Aggregated residual transformations for deep neural networks. In *Proc. the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, Jul. 2017, pp.1492-1500. DOI: 10.1109/CVPR.2017.634.

[7] Shazeer N, Mirhoseini A, Maziarz K, Davis A, Le Q, Hinton G, Dean J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv:1701.06538, 2017. https://arxiv.org/abs/1701.06538, Jan. 2021.

[8] Wang L, Ye J, Zhao Y, Wu W, Li A, Song S L, Xu Z, Kraska T. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proc. the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2018, pp.41-53. DOI: 10.1145/3178487.3178491.

[9] Rhu M, Gimelshein N, Clemons J, Zulfiqar A, Keckler S W. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016, Article No. 18. DOI: 10.1109/MICRO.2016.7783721.

[10] Pisarchyk Y, Lee J. Efficient memory management for deep neural net inference. arXiv:2001.03288, 2020. https://arxiv.org/abs/2001.03288, Jan. 2021.

[11] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759, 2014. https://arxiv.org/abs/1410.0759, April 2021.

[12] Barrachina S, Castillo M, Igual F D, Mayo R, Quintana-Orti E S. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proc. the 2008 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008. DOI: 10.1109/IPDPS.2008.4536485.

[13] Mahmoud M, Siu K, Moshovos A. Diffy: A Déjà vu-free differential deep neural network accelerator. In *Proc. the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018, pp.134-147. DOI: 10.1109/MICRO.2018.00020.

[14] Zhuang Y, Peng S, Chen X, Zhou S, Zhi T, Li W, Liu S. Deep fusion: A software scheduling method for memory access optimization. In *Proc. the 16th IFIP WG 10.3 International Conference on Network and Parallel Computing*, Aug. 2019, pp.277-288. DOI: 10.1007/978-3-030-30709-7_22.

[15] Chen X, Peng S, Jin L, Zhuang Y, Song J, Du W, Liu S, Zhi T. Partition and scheduling algorithms for neural network accelerators. In *Proc. the 13th International Symposium on Advanced Parallel Processing Technologies*, Aug. 2019, pp.55-67. DOI: 10.1007/978-3-030-29611-7_5.

[16] Zhang X, Zhi T. Machine learning inference framework on multicore processor. *Journal of Computer Research and Development*, 2019, 56(9): 1977-1987. DOI: 10.7544/issn1000-1239.2019.20180786. (in Chinese)

[17] Long G, Yang J, Zhu K, Lin W. Fusion stitching: Deep fusion and code generation for tensorflow computations on GPUs. arXiv:1811.05213, 2018. https://arxiv.org/abs/1811.05213, April 2021.

[18] Minakova S, Stefanov T. Buffer sizes reduction for memory-efficient CNN inference on mobile and embedded devices. In *Proc. the 23rd Euromicro Conference on Digital System Design*, Aug. 2020, pp.133-140. DOI: 10.1109/DSD51259.2020.00031.

[19] Guan Y, Liang H, Xu N, Wang W, Shi S, Chen X, Sun G, Zhang W, Cong J. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proc. the 25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, April 30-May 2, 2017, pp.152-159. DOI: 10.1109/FCCM.2017.25.

[20] Wei X, Liang Y, Zhang P, Yu C H, Cong J. Overcoming data transfer bottlenecks in DNN accelerators via layer-conscious memory managment. In *Proc. the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2019, pp.120-120. DOI: 10.1145/3289602.3293947.

[21] Frazier P I. A tutorial on Bayesian optimization. arXiv:1807.02811, 2018. https://arxiv.org/abs/1807.02811, April 2021.

[22] Eriksson D, Pearce M, Gardner J R, Turner R, Poloczek M. Scalable global optimization via local Bayesian optimization. arXiv:1910.01739, 2019. https://arxiv.org/abs/1910.01739, April 2021.

[23] Nayebi A, Munteanu A, Poloczek M. A framework for Bayesian optimization in embedded subspaces. In *Proc. the 36th International Conference on Machine Learning*, June 2019, pp.4752-4761.

[24] Wang L, Fonseca R, Tian Y. Learning search space partition for black-box optimization using Monte Carlo tree search. arXiv:2007.00708, 2020. https://arxiv.org/abs/2007.00708, April 2021.

[25] Varelas K, Auger A, Brockhoff D, Hansen N, ElHara O A, Semet Y, Kassab R, Barbaresco F. A comparative study of large-scale variants of CMA-ES. In *Proc. the 15th International Conference on Parallel Problem Solving from Nature*, Sept. 2018, pp.3-15. DOI: 10.1007/978-3-319-99253-2_1.

[26] Abadi M, Barham P, Chen J *et al.* TensorFlow: A system for large-scale machine learning. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, November 2016, pp.265-283.

[27] Paszke A, Gross S, Massa F *et al.* PyTorch: An imperative style, high-performance deep learning library. arXiv:1912.01703, 2019. https://arxiv.org/abs/1912.01703, April 2021.

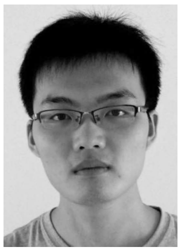
[28] Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In *Proc. the 22nd ACM International Conference on Multimedia*, Nov. 2014, pp.675-678. DOI: 10.1145/2647868.2654889.

[29] Whitley D. A genetic algorithm tutorial. *Statistics and Computing*, 1994, 4(2). DOI: 10.1007/BF00175354.

[30] Knuth D. The Art of Computer Programming, Volume I: Fundamental Algorithms. Addison-Wesley, 1968.

[31] Zhang S, Du Z, Zhang L, Lan H, Liu S, Li L, Guo Q, Chen T, Chen Y. Cambricon-X: An accelerator for sparse neural networks. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016. DOI: 10.1109/MICRO.2016.7783723.

[32] Lan H Y, Wu L Y, Zhang X, Tao J H, Chen X Y, Wang B R, Wang Y Q, Guo Q, Chen Y J. DLPlib: A library for deep learning processor. *Journal of Computer Science and Technology*, 2017, 32(2): 286-96. DOI: 10.1007/s11390-017-1722-2.



Xiao-Bing Chen received his B.E. degree in information security from Wuhan University, Wuhan, in 2016. He is currently a Ph.D. candidate in University of Chinese Academy of Sciences, Beijing. His research interests include deep learning and compilation optimization.

Hao Qi received his B.E. degree in computer science from Huazhong University of Science and Technology, Wuhan, in 2018. He is currently a Master student at University of Science and Technology of China, Hefei. His research interests include compilation optimization.

Shao-Hui Peng received his B.E. degree in computer science from University of Chinese Academy of Sciences, Beijing, in 2018. He is currently a Ph.D. candidate in University of Chinese Academy of Sciences, Beijing. His research interests include reinforcement learning.

Yi-Min Zhuang received his B.E. degree in electronic engineering from University of Science and Technology of China, Hefei, in 2016. He is currently a Ph.D. candidate in University of Chinese Academy of Sciences, Beijing. His research interests include deep learning and compiler of neural network accelerator.

Tian Zhi received her B.E. degree in biomedical engineering from Zhejiang University, Hangzhou, in 2009, and her Ph.D. degree in information engineering from Institute of Electrics (IE), Chinese Academy of Sciences, Beijing, in 2014. She is currently an associate professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include computer architecture and computational intelligence.

Yun-Ji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, in 2002. He received his Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, in 2007. He is currently a professor at ICT, Beijing. His research interests include parallel computing, microarchitecture, hardware verification, and computational intelligence.