doi:10.3772/j.issn.1006-6748.2022.04.004

HOPE: a heterogeneity-oriented parallel execution engine for inference on mobiles①

XIA Chunwei(夏春伟)[*,**], ZHAO Jiacheng②[*], CUI Huimin[*,**], FENG Xiaobing[*,**]
([*] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China)
([**] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100190, P. R. China)

Abstract

It is significant to efficiently support artificial intelligence (AI) applications on heterogeneous mobile platforms, especially coordinately execute a deep neural network (DNN) model on multiple computing devices of one mobile platform. This paper proposes HOPE, an end-to-end heterogeneous inference framework running on mobile platforms to distribute the operators in a DNN model to different computing devices. The problem is formalized into an integer linear programming (ILP) problem and a heuristic algorithm is proposed to determine the near-optimal heterogeneous execution plan. The experimental results demonstrate that HOPE can reduce up to 36.2% inference latency (with an average of 22.0%) than MOSAIC, 22.0% (with an average of 10.2%) than StarPU and 41.8% (with an average of 18.4%) than μLayer respectively.

Key words: deep neural network (DNN), mobile, heterogeneous scheduler, parallel computing

0 Introduction

Deep neural networks (DNNs) are increasingly adopted in mobile applications and have become the core building blocks of top apps[1]. Typically, for these applications, the inference latency is a significant issue for users. Meanwhile, mobile platforms are emerging towards heterogeneous embedded systems[2], which integrate a variety of computing devices into a mobile system-on-chip (SoC). These mobile computing devices exhibit huge diversity in performance, power consumption, memory capacity and programming interface[3].

To efficiently exploit the heterogeneity and support artificial intelligence (AI) applications on heterogeneous mobile platforms, several frameworks are proposed. For example, TFLite[4] could run inference workload on graphics processing unit (GPU) through GPU delegate or other accelerators through the Android neural networks API (NNAPI). MNN[5] supported running DNN models on mobile GPU through OpenCL, OpenGL, Vulkan or Apple Metal. These mobile frameworks provide fundamental support for running a DNN model across different platforms. However, there still lacks heterogeneity models to automatically distribute a DNN model across the on-chip processing units. To address this challenge, MOSAIC[6] used a heterogeneity-, communication-, and constraint-aware model slicing approach to distribute a DNN model across computing devices on a heterogeneous platform. However, MOSAIC considers the DNN model as a linear model without exploring the inter-operation parallelism. On traditional CPU-GPU hybrid servers, researchers have proposed a number of approaches for dynamically scheduling parallel tasks to heterogeneous hardware[7-11], and a representative work is StarPU[12]. However, these runtime heuristics are not applicable to mobile platforms. The reason is that DNN models have some special directed acyclic graph (DAG) topology structures, which are built from layers or blocks with similar structure and sizes, especially for large models. But StarPU is unaware of the DAG topology characteristics, thus missing the opportunity of globally determining the scheduling policy and causing performance loss. Ref. [13] proposed μlayer which accelerates each NN layer by simultaneously utilizing heterogeneous computing devices on mobile SoCs.

To materialize the optimization, there are several major challenges. Challenge 1: trade-off between performance and scheduling time. Scheduling on mobile platform requires an acceptable scheduling time to obtain the execution plan. StarPU is lightweight but the performance is degraded, leaving the optimization op-

① Supported by the General Program of National Natural Science Foundation of China (No. 61872043).
② To whom correspondence should be addressed. E-mail: zhaojiacheng@ict.ac.cn.
Received on Jan. 18, 2022

portunity submerged. Challenge 2: communication overhead. Running DNN operators to heterogenous computing devices will introduce communication overhead which requires careful consideration. For DNN frameworks with CPU and GPU kernels using different data layout, μlayer will introduce significant data layout translation overhead.

In this paper, HOPE, an end-to-end lightweight heterogeneous inference framework is proposed to distribute a DNN model coordinately on different computing devices of a platform. The key insight is that many DNN models exhibit inter-operation parallelism and enable different operations to be executed in parallel on multiple computing devices. Meanwhile, the topology characteristics of DNN models can be used to seek for an optimal scheduling solution. HOPE first profiles the computation latency for each DNN operator, together with the communication cost of each tensor between every two computing devices. Then, the problem is formalized as an integer linear programming (ILP), and for complex DNN models that require an extremely long time for ILP solver, HOPE partitions a graph into multiple subgraphs and solves ILP for each subgraph individually.

For challenge 1, two distinct algorithms are proposed. The ILP based scheduling method is proposed for best performance but longer scheduling time, while the heuristic method is designed to get the execution plan much faster with moderate performance. Typically, it takes several seconds for the ILP based algorithm to get the execution plan while it takes less than one second for the heuristic method. A heuristic algorithm is also proposed to partition the DNN model into multiple modules, with each module containing several operators. Finally, HOPE includes an execution engine for simultaneously launching modules to different computing devices according to the execution plan. Experimental results demonstrate that HOPE can reduce up to 36.2% inference latency (with an average of 22.0%) than MOSAIC, 22.0% (with an average of 10.2%) than StarPU, respectively, and 41.8% (with an average of 18.4%) than μlayer, compared with the state-of-the-art work.

1 Problem formalization

The problem of scheduling a dataflow graph model across multiple heterogeneous computing devices can be formalized as the following.

A dataflow graph can be described with $G(V, E)$, where V represents the set of vertexes and E represents the set of edges. Typically, a vertex $v_i \in V$ represents an operator like convolution in DNN models. While an edge (v_i, v_j) represents that operator v_j depends on the output of operator v_i. Therefore, v_j cannot start executing until v_i is finished. Given a dataflow graph model $G(V,E)$, and a set D containing all computing devices of the target platform, for each $v_i \in V$, its execution time on device d_j is $cl_{i,j}$; for each (d_i, d_j) pair $(d_i, d_j \in D)$, the communication cost is $C_{(d_i, d_j)}(T)$ if a tensor T is required to transfer from d_i to d_j. The algorithm's objective is to find out an execution plan R determining the execution device d_j for each node v_i, which minimizes the end-to-end latency $\tau(R, G)$. An execution plan $R(b,\psi)$ contains two parts, i. e., a mapping matrix b indicating the target device for each v_i, and an order matrix array ψ indicating the execution order of the operators mapped to each device. In $R(b, \psi)$, b is a two-valued mapping matrix recording the selected device for each operator, with each element $b_{i,j}$ representing whether the node v_i is scheduled to the device d_j, i. e.,

$$b_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ is scheduled on } d_j \\ 0 & \text{if } v_i \text{ is not scheduled on } d_j \end{cases}$$

ψ is a matrix array, and the j-th element is a matrix representing the execution order of the operators mapped to d_j, i. e.,

$$\psi_{i,k,j} = \begin{cases} 1 & \text{if } v_i \text{ is scheduled before } v_k \text{ on device } d_j \\ 0 & \text{if } v_i \text{ is scheduled after } v_k \text{ on device } d_j \end{cases}$$

2 ILP-based theoretically optimal solution

In this section, firstly, the parallel DAG execution problem is formalized as an ILP, by considering the computation time of each node and communication cost between computing devices. Then a graph partitioning algorithm is proposed to reduce the graph complexity for the ILP solver.

2.1 ILP formulation

2.1.1 Node latency

The overall processing latency of a node v_i on a device d_j comes from two parts. The first part is the kernel computation latency $cl_{i,j}$ of node v_i on the computing device d_j. If node v_i is not supported by d_j, $cl_{i,j}$ is set to infinite ($+\infty$). The second part is the communication latency of receiving tensors from v_i's predecessors. In particular, for the predecessor v_j, the communication latency from v_j to v_i is denoted as $C_{(d(v_j), d(v_i))}(T)$, where $d(v_j)$ and $d(v_i)$ represent the computing device for v_j and v_i respectively, T represents the tensor transferred from v_j to v_i. Furthermore, when v_i has multiple predecessors, its total communication

cost is computed by aggregating the tensors from all its predecessors, i. e. ,

$$comm_i = \sum_{v_j \in pred(v_i)} C_{d(v_j),\, d(v_i)}(T)$$

Therefore, the processing latency $t_{i,j}$ of node v_i on device d_j can be formalized to the following expression.

$$t_{i,j} > b_{i,j} \times cl_{i,j} + \sum_{(v_k,\, v_i)}^{\forall v_k \in pred(v_i)} (b_{i,j} - b_{k,j}) \times C_{(d(v_k),\, d(v_i))}(T) \quad (1)$$

$(b_{i,j} - b_{k,j}) \times C_{(d(v_k),\, d(v_i))}(T)$ describes whether there is communication overhead between the predecessor v_i and node v_k. If node v_i and node v_k are distributed to different devices (for example node v_k is dispatched on CPU and node v_i is dispatched on GPU), then the output tensor produced by node v_k needs to be transferred from CPU memory space to GPU memory space. If $(b_{i,j} - b_{k,j})$ is less than zero, then $b_{i,j}$ must be zero and $b_{k,j}$ must be one. The right side of the inequation is less than zero and the equation is always true. In other words, on device $d(v_j)$ vertex v_i does not bring any constraint on vertex v_j. Note that the communication from the predecessors can overlap with the execution of operators that have no precedence constraints with v_i.

2.1.2 Objective function

The objective is to minimize the execution time of the computation graph, thus an auxiliary variable $st_{i,j}$ is introduced to describe that node v_i starts its execution on device d_j at time $st_{i,j}$. The end-to-end computation latency of the DNN model is the interval from the starting time of the first node to the completion time of the last node. Therefore, the objective is to minimize the end-to-end latency.

$$\text{Minimize } \tau(\Re, G) \quad (2)$$

st.

$$\tau(R, G) > b_{i,j} \times (st_{i,j} + t_{i,j}) \quad (\forall v_i \in V), (\forall d_j \in D)$$

2.1.3 Graph topology constraints

For node v_i and node v_k, if there is an edge $(v_i, v_k) \in E$, the graph topology constraint must be satisfied:

$$st_{k,l} > st_{i,j} + t_{i,j} + (b_{i,j} - 1) \times M \quad (3)$$
$$(\forall (v_i, v_k) \in E), (\forall d_l, d_j \in D)$$

This constraint describes that for any node v_k on any device d_j it cannot start execution until all its predecessor(s) v_i are finished. If $b_{i,j}$ is 0, the constraint will always hold true. Note that M is a sufficient large positive number.

2.1.4 Device constraints

In this paper, it is assumed that a computing device will not be shared by multiple operators simultaneously, thus at any time there can be at most one operator running on each device. And the execution of a task cannot be interrupted. If there is a path from node v_i to node v_j, the device constraint would be naturally satisfied since the graph topology constraint guarantees that v_j would start after v_i. Therefore, only pairs of nodes that have no path introduce the following constraint.

$$st_{i,j} > st_{k,j} + t_{k,j} \quad \forall d_j \in D \quad (4)$$
$$\text{or} \quad st_{k,j} > st_{i,j} + t_{i,j} \quad \forall d_j \in D$$

This constraint illustrates that v_i and v_k can be executed in any order but cannot execute in parallel on device d_j. An auxiliary variable $\psi_{i,k,j}$ and a sufficiently large number M are introduced for the ILP formalization, as

$$st_{i,j} > st_{k,j} + t_{k,j} - \psi_{i,k,j} \times M \quad \forall d_j \in D \quad (5)$$
$$st_{k,j} > st_{i,j} + t_{i,j} - (1 - \psi_{i,k,j}) \times M \quad \forall d_j \in D$$

where $\psi_{i,k,j}$ is a binary variable which describes the execution order of node v_i and v_k. If it equals to 0, v_i would execute after v_k, and vice versa.

2.1.5 Node constraints

Each node is expected to be executed only once:

$$\sum_{j}^{\forall d_j \in D} b_{i,j} = 1, \quad \forall v_i \in V \quad (6)$$

2.1.6 Summary

With constraint Eqs(1) - (6), the problem can be solved by using a standard ILP solver, e. g., GLPK[13]. Finally, the execution plan is expressed using two variables, $b_{i,j}$ describing the device placement of the nodes, and $\psi_{i,k,j}$ describing the execution order of the node pair v_i and v_k.

2.2 Graph partitioning problem

The graph partitioning problem is formalized as follows. Given a DAG $G = (V, E)$ and an imbalance parameter ϵ, find an acyclic k-way partition $P = V_0, V_1$ of V such that the balance constraint

$$w(V_i) \leqslant (1 + \epsilon) \frac{\sum_{v \in V_i} w(v)}{k} \quad (7)$$

is satisfied and the vertex cut is minimized. The set $\{V_0, V_1\}$ represents the two subgraphs' vertexes of the graph G. $w(v)$ is the weight of vertex and is set to 1 for all the vertices. $w(V_i)$ is the sum of $w(v)$ for all vertex v in V_i. To find a proper partitioning point, the concept of upward rank is included, and is defined as

$$uprank(v_j) = 1 + \max_{v_i \in pred(v_j)}(uprank(v_i)) \quad (8)$$

Intuitively, the upward rank of v_i is the longest path from the input vertex(es) of G to v_i. The upward rank

of all the vertexes in a DAG can be computed in a single traversal of G in the topology order with a time complexity $O(|V|+|E|)$. The upward rank value of vertexes is used to partition the DAG into subgraphs by recursively partitioning the subgraphs until all the subgraphs have the number of vertices less than NT. NT is set to 12 in the evaluation and can be configured.

Algorithm 1 shows the pseudo-code of the partitioning approach with up rank. The algorithm tries to find an appropriate upward rank value that partitions G to two parts with approximate number of vertexes and minimize the vertex cut. The algorithm first computes the upward rank value of each vertex in the graph using equation (line 1). Then the minimum and maximum upward rank value is computed (lines 2 – 3). After that the algorithm counts the number of vertexes with every upward rank value by traversing G (lines 5 – 6). Then by the increasing order of up rank, the accumulated number of vertexes is summed up (lines 7 – 8). The variable accUprank[rank] represents how many vertexes there are whose upward rank value is less or equal than rank. Then the algorithm traverses all the upward rank values and finds in which upward rank value the balance constraint can be satisfied and the vertex cut can be minimized. Then the partitioning result with the minimum vertex cut is obtained (lines 9 – 18). The computational complexity of this partitioning algorithm is $O(|V|\log(|V|))$. The balance factor ϵ is set to 0.2 in this paper and if no balanced partition can be obtained, the algorithm will increase the ϵ by 0.1 until a valid partition scheme is found.

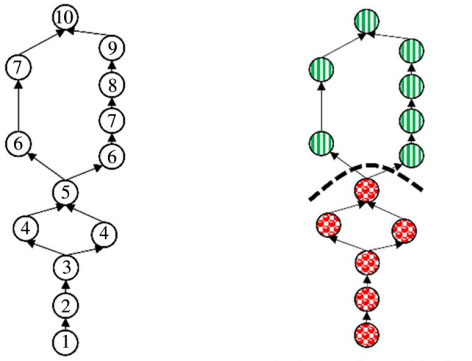
Algorithm 1: Graph partitioning with up rank.

```
   Input: Directed graph G=(V, E)
   Result: A feasible partitioning (V0, V1) of G
 1 compute upward rank value for each vertex in G;
 2 min U prank ← minimum upward rank value in G;
 3 max U prank ← maximum upward rank value in G;
 4 alloc uprankCount[maxUprank], accUprank[maxUprank];
 5 for v in V do
 6     uprankCount[uprank(v)]+ = 1
 7 for rank ← minUprank to maxUprank do
 8     accUprank[rank]= uprankCount[rank]+ accUprank[rank−1]
 9 minVertexCutCount ← |V|;
10 minBlance ← |V|;
11 V0,V1 ← ∅;
12 for rank ← minUprank to maxUprank do
13     w(V0) ← accUprank[rank];
14     w(V1) ← |V|− accUprank[rank];
15     if balance constraint is satisfed and uprankCount[rank]<= minVertexCutCount and
        minBlance< abs(|V0|−|V1|) then
16         V0 ← {v ∀v ∈V if uprank(v)≤rank};
17         V1 ← {v ∀v ∈V if uprank(v)>rank};
18         minVertexCutCount ← uprankCount[rank]
19 return (V0, V1)
```

Fig. 1 shows an example of graph partitioning using upward rank. The number in the vertex is the upward rank. There are 6 vertexes whose upward rank value is less than or equal to 5 and 7 vertexes whose upward rank value is greater than 5. If partitioning the graph with upward rank 5, the balance constraint is satisfied as $(1+0.2)\frac{13}{2}=7.8$, and the vertex cut is 1, which is also minimized. The coarse graph with V_0 and V_1 being the vertexes is also acyclic.

3 Heuristic scheduling algorithm

In this section, a greedy algorithm is proposed to rapidly obtain a near-optimal execution plan Re, including the device placement of each operator and the execution order of the nodes on each device. The key point



(a) An example graph from Inception-V4 with up rank (b) An acyclic partitioning

Fig. 1 Graph partitioning example with up rank

is that the algorithm schedules a batch of operators for which the optimal execution plan is obtained. Furthermore, operators having the minimal starting execution time st would be scheduled first, where the minimal starting execution time of an operator is defined as the maximal completion time of all its predecessor nodes.

3.1 Greedy search algorithm

First, the algorithm greedily selects the top-K nodes that have the minimal starting execution time, by sorting the starting execution time of all operators in the increasing order, and these top-K nodes would be the candidate operators for scheduling. Second, the algorithm enumerates all possible operator-to-device mappings and selects the mapping leading to the minimal completion time of these K operators. If a DAG has y nodes and the target mobile platform has x devices, the heuristic-based algorithm's computation complexity is $O(yk \times x^k)$. The algorithm empirically set k to 4 for two computing devices and 3 for three computing devices. Finally, the algorithm updates the successor operators of these K nodes for their starting execution time using the computed completion time of the K nodes. Algorithm 2 shows the pseudo-code.

Algorithm 2: Greedy searching

Input: DAG with profle data; computing device
Result: Execution plan
1 opsQueue ← inputNodes;
2 **while** opsQueue! = ∅ **do**
3 sort opsQ eue by starting execution time;
4 kOps ← opsQueue.getTopK ();
5 searchMiniEndTime (kOps, devices);
6 update Successor Starting Execution Time (kOps);
7 opsQueue ← kOps.successors
8 **end**

3.2 Node merging for heuristic algorithm

In DNN models, there exist several short running operators that have very low computation latency comparing with the computation-intensive operators. For example, on the Redmi, the latency of ReLU is less than 0.1 ms while that of Conv2D may be more than 3 ms. This observation motivates us to introduce node merging algorithm. For a short running operator, if it has only one predecessor, the algorithm prefers to dispatch it to the same computing device as its predecessor. The merged node is called super-op. The algorithm first runs the node merging algorithm then calls the heuristic scheduler to generate the execution plan. The merged nodes within a super-op will be scheduled to the same device. For frameworks that support kernel fusion, HOPE's scheduler will schedule the fused kernels as a whole.

4 Framework

Fig. 2 shows the HOPE framework. First, a DNN model is fed to the benchmarking engine, which collects the execution time of each node on each computing device of the target platform, together with the communication cost between each pair of devices for the tensors in the model. Second, graph pre-processing partitions the DAG into several smaller modules, with each module consisting of a number of super-ops, as discussed in subsection 4.2. Third, the heterogeneity-aware scheduler leverages the scheduling algorithm in subsection 2.1 (ILP-Scheduler) or subsection 3.1 (GS scheduler) to generate an execution plan that determines the execution order of all nodes and the target device for each node. Finally, the HOPE heterogeneous execution engine uses the generated execution plan and launches the parallel execution across different computing devices.

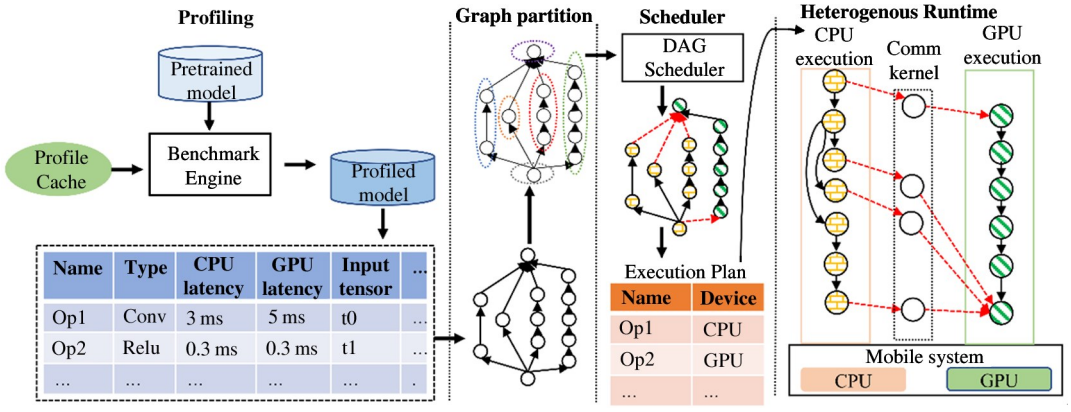


Fig. 2 System overview

4.1 Benchmarking engine

The benchmarking engine serves to collect all the profiles that are needed by the scheduler. First, it runs the DNN model, launches each node to each available computing device, and records the corresponding execution time. Second, it collects the information for all tensors passed between adjacent nodes, including their shape and layout information, synthesizes benchmarks to mandatorily transfer each tensor between any pair of computing devices, applies the corresponding layout transformations when necessary, and records these execution time as the communication cost. In some cases, communication will not introduce extra cost, e. g., transferring a tensor between a big core and a little core of one unique CPU, thus, such communication cost is set to zero. Typically, a variety of computing devices support different tensor layouts via vendor-provided or third-party libraries. For convolution operations, CPU implementation uses the data layout of NC4HW4[5], while GPU with OpenCL implementation uses the layout of OpenCL image object with the shape being ($C/4 \times W$, $N \times H$, 4) in the MNN framework. The profiling overhead of the communication cost and layer-wise computation latency is limited, less than 10 min for one DNN model. And performance variation is low (around 1%) thus this paper only reports the average results. The number of tensors with different shapes is much less than the number of operators, there are 1049 operators but only 39 tensors with different shapes in NASNET-large.

4.2 Heterogenous execution engine

The heterogeneous execution engine reads an executionplan and executes the DNN model by launching each operator to its target computing devices. Before execution, the execution engine first inserts necessary data transfer and layout transformation statements according to the determined execution plan. Fig. 3 shows an example of inserting a communication operator to transfer and transform the tensor 1 from CPU memory space to GPU memory space.

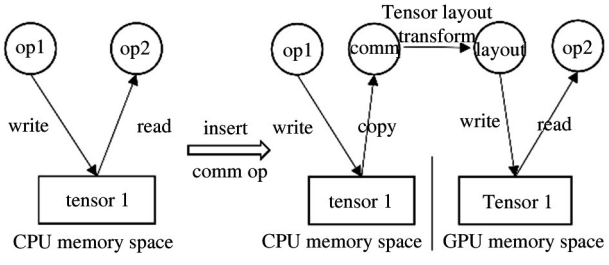


Fig. 3 Example of inserting communication operator

The execution engine creates an individual thread for each computing device, with the main thread serving for the CPU big core. To avoid interference across the threads of each computing device, HOPE uses sched _ set _ affinity API to bind each thread to a dedicated CPU core. Users can configure the number of threads for operators running on CPU. The execution engine introduces a queue for each computing device to keep the nodes that are placed on the device according to the execution plan. Each operator has one flag executed to indicate whether the operator has been executed. After one operator v_i is executed, the engine traverses all its successor operators v_k in the order determined by the execution plan. For a given successor v_k, if it is ready for execution and is mapped to the same device, it will be launched; if it is ready for execution but mapped to another device, the engine would notify the thread of the target device and launch the operator on the corresponding device. Otherwise, the thread would keep sleeping.

4.3 Tensor caching

Motivated by the observation that a tensor would be used multiple times on a device, HOPE introduces a tensor-oriented optimization named tensor-caching. After a tensor has been transferred and transformed to a device, it would be cached on the target device, so that when the tensor is reused the corresponding communication cost can be eliminated. As soon as no operator will read it, the tensor will be freed.

4.4 Optimizing CPU and GPU communication

For the mobile systems in which CPU and GPU share the same physical memory and support shared virtual memory, HOPE provides an optimization to eliminate the memory copy in the communication between CPU and GPU. Specifically, OpenCL provides a few mechanisms to avoid costly memory copy between CPU and GPU. HOPE utilizes the CL _ MEM _ ALLOC _ HOST _ PTR flag provided by OpenCL to create OpenCL buffers and the clEnqueueMapBuffer API to map the OpenCL buffer to a host pointer. Then the CPU can update the mapped OpenCL buffers to transform tensor data layout directly without additional memory copy.

5 Evaluation

5.1 Platform and benchmark

Mobile systems. The experiments are conducted on 3 commercial mobile phones including Redmi Note 4x (low-end (LE) with Snapdragon 625), Xiaomi 9SE (medium-end (ME) with Snapdragon 712),

HUAWEI P40(high-end (HE) with Kirin 990), ranging from low-end to high-end mobile platforms.

DNNs and datasets. HOPE is evaluated on seven most popular CNNs including Inception-v3 (IN-V3)[14], Inception-v4(IN-V4)[15], PNASNET-mobile (PN-M)[16], PNASNET-large(PN-L)[16], NASNET-large(NAS-L)[17], and SqueezeNet(SQZ)[18], and one long short-term memory (LSTM)[19] model which follows the configuration in DeepSpeech[20]. All the CNN models are trained on the ImageNet dataset. The LSTM has one layer with 1024 hidden states and 10 time steps. The workloads include the DNN models both highly optimized for mobile systems (e.g., PN-M and SQZ) and models for highest accuracy (e.g., IN-V4 and PN-L). Further, the evaluated CNN models exhibit widely different number of operators ranging from 40 to 1049, which indicates that efficient scheduling algorithms are required to place the numerous operators to computing devices for heterogeneous parallel execution.

Implementation. The latest version of the GNU linear programming kit (GLPK) is used to solve the ILP formulation. The heterogeneous execution engine is written in 5000 LOC in C++. The graph partitioning and scheduling algorithms are written with 4000 LOC in Python.

5.2 Overall performance

To investigate the effectiveness of HOPE in terms of inference latency with different computing device combinations and configurations, a series of comparative experiments have been conducted. MOSAIC, the state-of-the-art heterogeneous execution engine for DNN model inference, and StarPU's dequeue model data aware ready (DMDAR) policy, a most suitable and classical heuristic that is used a reference heuristic in many recent scheduling studies used in comparison with the proposed approaches. On the LE mobile system 4 CPU cores are used, and all the 2 big cores on the ME and HE mobile systems and their GPU for evaluation. Note that StarPU does not use all the CPU cores on the big LITTLE architecture mobile platforms. The reason is that the framework is unaware of the heterogeneous processing architecture and will distribute workloads evenly to big and little cores. The little cores would slow down the computation without carefully scheduling. HOPE is aware of the heterogeneity of CPU clusters, and the detailed experimental results is shown in subsection 6.5. Fig. 4 shows the normalized inference latency of the seven DNN models on the three mobile systems. For each DNN model, six different versions are evaluated: CPU only (CPU), GPU only (GPU), CPU and GPU with MOSAIC as the scheduler (MOSAIC), CPU and GPU with StarPU's (DMDAR) policy (StarPU), CPU and GPU with HOPE's heuristic scheduler (HOPE:GS), and HOPE's ILP scheduler HOPE:LP version. All the inference latency of each version is normalized to the GPU version.

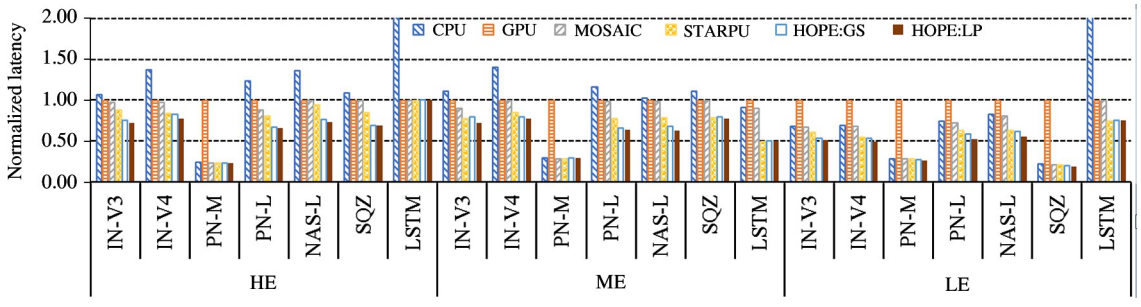


Fig. 4 Normalized inference latency on the three mobile systems

The tensor cache optimization is enabled by default for all versions. Fig. 4 demonstrates the effectiveness of HOPE's ILP and heuristic scheduler. First, HOPE significantly outperforms the performance of MOSAIC. Specifically, HOPE with ILP scheduler exhibits 23.4%, 24.1%, 2.9%, 29.4%, 31.1%, 21.3%, 23.3% lower latency than the MOSAIC version of the seven DNN models respectively of the three mobile systems on average. This is because HOPE can schedule the DNN operators to different computing devices and run them in parallel. Second, HOPE with ILP scheduler exhibits 13.7%, 9.1%, 2.9%, 17.9%, 18.0%, 10.0% and 0% lower latency than the STARPU version. HOPE:LP significantly reduces inference latency by finding the optimal solution for each subgraph and globally determining the scheduling policy. Third, HOPE:GS also reduces the inference latency by 8.2%, 3.2%, 0.5%, 12.9%, 11.7%, 0.0% and 0.0% compared with StarPU version. The performance gain comes from two aspects. The first is that HOPE:GS merges nodes before scheduling and the potential communication cost is reduced. The second is that HOPE:GS schedules nodes in a larger search window (with K nodes) while StarPU considers only one node.

Note that for LSTM there is no computation reduction compared with StarPU.

The reason is that the cell has only four gates and the structure is rather simple. The experimental results clearly demonstrate the effectiveness of HOPE in that it can effectively reduce the inference latency than MOSAIC and StarPU. HOPE exhibits a similar inference latency to that of the CPU alone version with PNAS-M on all the three mobile systems. This is mainly because the PNAS-M is specifically optimized for CPU inference. For instance, it takes 43 ms running PNAS-M with CPU while 183 ms with GPU on the HE.

5.3 Scalability with CPU performance

Next, consider the peak performance ratio between CPUs and GPUs across mobile SoCs varies[3], the frequency of the CPU cores varies to investigate the performance scalability of HOPE scheduler. The LE is chosen as the mobile system and the available CPU frequency ranges from 652 MHz to 2016 MHz. The maximum, medium and minimum CPU frequencies in the CPU's scaling _ avaliable _ frequencies list and GPU are used for evaluation. Fig. 5 shows the normalized inference latency of each version on the DNN models. HOPE:LP reduces up to 26.4% (with an average of 19.2%), 25.2% (with an average of 17.5%) and 29.1% (with an average of 22.1%) inference latency than the MOSAIC version, and 17.2% (with an average of 10.8%), 20.4% (with an average of 9.3%) and 17.0% (with an average of 10.3%) inference latency than the StarPU version, with the minimum, medium and maximum CPU frequencies respectively. Similar observations can be found for the HOPE:GS version. Experimental results show that HOPE still outperforms the MOSAIC version and the StarPU version when the peak performance ratio between CPU and GPU varies.

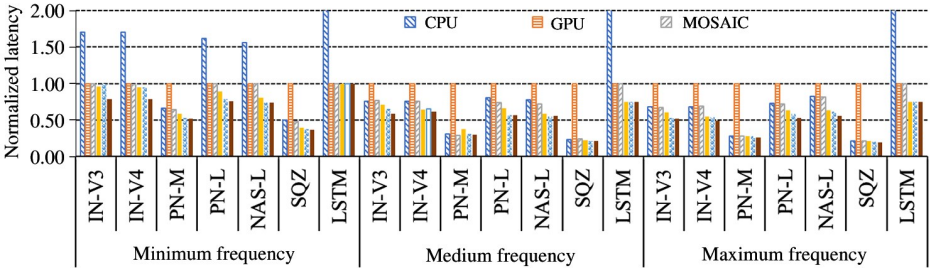


Fig. 5 Normalized inference latency with CPU frequency scaling on LE

5.4 Comparison with μlayer

This work targets at full-precision DNN model inference thus HOPE is compared against μlayer using FP32. μlayer runtime is implemented on ARM compute library (ACL) as described in Ref. [13]. One benefit of using ACL rather than MNN is that NEON- and OpenCL- based kernels in ACL utilize the same data layout, and no data layout conversion overhead is introduced when CPU and GPU need synchronization.

Fig. 6 shows the normalized execution latency of μlayer and two scheduling algorithms of HOPE. The results show that HOPE:GS can reduce the computation latency by up to 32.9% (HE), 32.8% (ME) and 25.4% (LE) over single-layer acceleration of μlayer. HOPE:LP can reduce the computation latency by up to 40.6% (HE), 35.9% (ME}) and 28.0% (LE). The reason is that single-layer acceleration of μlayer needs fine-grained CPU-GPU synchronization for each layer while HOPE does not. Note that for LSTM HOPE does not reduce the latency over μlayer as HOPE and μlayer partition the LSTM cell with the same split ratio.

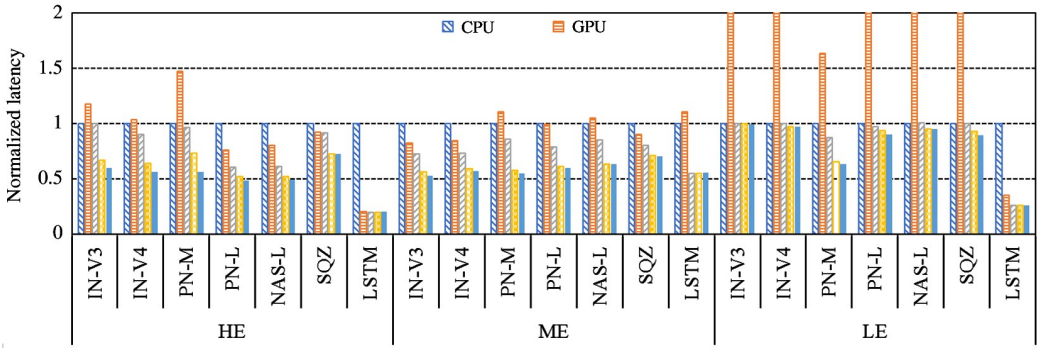


Fig. 6 comparison with μlayer

HOPE can reduce 25.8% and 21.3% computation latency on average for the HE and ME platforms, respectively, while only 8.0% for LE. The reason is that the performance of the CPU and the GPU on the LE platform is severely imbalanced.

5.5 CPU big + LITTLE

Mobile SoCs adopted big LITTLE technology to balance the performance and power efficiency. The 'LITTLE' processors are designed for maximum power efficiency while 'big' processors are designed to provide maximum compute performance. HOPE can schedule the DNN models considering the heterogeneous processing architecture of CPU. The awareness of big LITTLE of HOPE is evaluated by using 2 big cores +4 little cores on the HE mobile system. The communication overhead can be eliminated when using big and little clusters as they share the same memory space and tensor layout for computation. Fig. 7 shows the result. Specifically, HOPE with HOPE: LP reduces 15.7%, 2.2%, 2.6%, 9.8%, 8.3%, 3.2% and 0% inference latency than StarPU. HOPE can efficiently reduce the inference latency and improve the QoS when using only CPU. HOPE: GS exhibits similar performance as STARPU for there is no communication cost and HOPE: GS cannot obtain benefit from merging nodes.

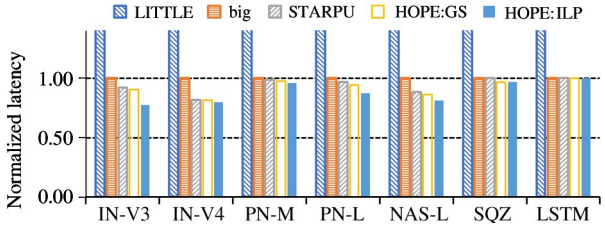


Fig. 7 Normalized latency with big LITTLE cores

6 Conclusion

In this paper, HOPE, a heterogeneity-oriented parallel execution engine for DNN model inference on mobile systems is proposed. HOPE profiles operator's computation latency of the accurate models, pre-processes the DAG to modules, and schedules the DAG with an ILP-based or a greedy-based algorithm to determine the near-optimal heterogeneous execution plan. The experimental results show that HOPE can significantly reduce the computation latency compared with the state-of-the-art work including MOSAIC, StarPU and μlayer. In the future, DNN models with control-flow will be supported and policies will be proposed to dynamically schedule operators. Furthermore, the potential benefits of collaborative execution on accelerators like NPU and DSP will also be explored.

Reference

[1] CHEN Z, CAO Y, LIU Y, et al. A comprehensive study on challenges in deploying deep learning-based software [C]// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, USA, 2020: 750-762

[2] IGNATOV A, TIMOFTE R, KULIK A, et al. AI benchmark: all about deep learning on smartphones [C]// 2019 IEEE/CVF International Conference on Computer Vision Workshop, Seoul, Korea, 2019: 3617-3635

[3] WU C, BROOKS D, CHEN K, et al. Machine learning at facebook: understanding inference at the edge [C]// The 25th IEEE International Symposium on High Performance Computer Architecture, Washington, USA, 2019: 331-344

[4] Google. Tensorflow lite [EB/OL]. https://www.tensorflow.org/lite: Google, [2022-01-18]

[5] JIANG X, WANG H, CHEN Y, et al. MNN: a universal and efficient inference engine [C]// Proceedings of Machine Learning and Systems, Austin, USA, 2020: 1-13

[6] HAN M, HYUN J, PARK S, et al. Mosaic: Heterogeneity-, communication-, and constraint aware model slicing and execution for accurate and efficient inference [C]// The 28th International Conference on Parallel Architectures and Compilation Techniques, Seattle, USA, 2019: 165-177

[7] KEDAD-SIDHOUM S, MONNA F, TRYSTRAM D. Scheduling tasks with precedence constraints on hybrid multicore machines [C]// IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, India, 2015: 27-33

[8] TOPCUOGLU H, HARIRI S, WU M. Performance-effective and low-complexity task scheduling for heterogeneous computing [J]. *IEEE Transactions on Parallel and Distributed Systems*, 2002, 13: 260-274

[9] ARABNEJAD H, BARBOSA J G. List scheduling algorithm for heterogeneous systems by an optimistic cost table [J]. *IEEE Transactions on Parallel and Distributed Systems*, 2014, 25(3): 682-694

[10] WITT C, WHEATING S, LESER U. LOS: level order sampling for task graph scheduling on heterogeneous resources [C]// 2018 IEEE/ACM Workflows in Support of Large-Scale Science, Dallas, USA, 2018: 20-30

[11] KANEMITSU H, HANADA M, NAKAZATO H. Clustering based task scheduling in a large number of heterogeneous processors [J]. *IEEE Transactions on Parallel and Distributed Systems*, 2016, 27(11): 3144-3157

[12] AUGONNET C, THIBAULT S, NAMYST R, et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures [J]. *Concurrency Computation Practice & Experience*, 2011, 23(2): 187-198

[13] KIM Y, KIM J, CHAE D, et al. μlayer: low latency on-device inference using cooperative single layer acceleration and processor-friendly quantization [C]// Proceedings of the Fourteenth EuroSys Conference, New York, USA,

2019: 1-15
[14] SZEGEDY C, VANHOUCKE V, IOFFE S, et al. Rethinking the inception architecture for computer vision [C]// 2016 IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, USA, 2016: 2818-2826
[15] SZEGEDY C, IOFFE S, VANHOUCKE V, et al. Inception-v4, inception-resnet and the impact of residual connections on learning[C]// The 31st AAAI Conference on Artificial Intelligence, San Francisco, USA, 2017: 4278-4284
[16] ZOPH B, LE Q V. Neural architecture search with reinforcement learning[C]// The 5th International Conference on Learning Representations, Toulon, France, 2017: 1-16
[17] Google, Tensorfow-slim nasnet-a implementation/checkpoints[EB/OL]. https://github.com/tensorflow/models/blob/master/research/slim/nets/nasnet/README.md: Google, [2022-01-18]
[18] IANDOLA F N, HAN S, MOSKEWICZ W, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size[EB/OL]. https://arxiv.org/abs/1602.07360: arXiv, (2016-02-24), [2022-01-18]
[19] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. *Neural Computation*, 1997, 9(8): 1735-1780
[20] AWNI Y H, CARL C, JARED C, et al. Deep speech: scaling up end-to-end speech recognition[EB/OL]. https://arxiv.org/abs/1412.5567: arXiv, (2014-12-17), [2022-01-18]

XIA Chunwei, born in 1994. He received his B. S. degree from Tianjin University in 2016. He is currently a Ph. D candidate at Institute of Computing Technology, Chinese Academy of Sciences. His research interests include the mobile cloud computing systems, heterogenous computing and program optimization.